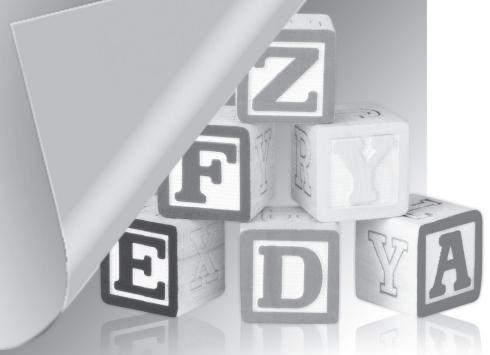


# SAS® Macro Programming Made Easy

*Third Edition*

Michele M. Burlew



[support.sas.com/bookstore](http://support.sas.com/bookstore)

The correct bibliographic citation for this manual is as follows: Burlew, Michele M. 2014. *SAS® Macro Programming Made Easy, Third Edition*. Cary, NC: SAS Institute Inc.

**SAS® Macro Programming Made Easy, Third Edition**

Copyright © 2014, SAS Institute Inc., Cary, NC, USA

ISBN 978-1-61290-791-8

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.

The scanning, uploading, and distribution of this book via the Internet or any other means without the permission of the publisher is illegal and punishable by law. Please purchase only authorized electronic editions and do not participate in or encourage electronic piracy of copyrighted materials. Your support of others' rights is appreciated.

**U.S. Government License Rights; Restricted Rights:** The Software and its documentation is commercial computer software developed at private expense and is provided with RESTRICTED RIGHTS to the United States Government. Use, duplication or disclosure of the Software by the United States Government is subject to the license terms of this Agreement pursuant to, as applicable, FAR 12.212, DFAR 227.7202-1(a), DFAR 227.7202-3(a) and DFAR 227.7202-4 and, to the extent required under U.S. federal law, the minimum restricted rights as set out in FAR 52.227-19 (DEC 2007). If FAR 52.227-19 is applicable, this provision serves as notice under clause (c) thereof and no other notice is required to be affixed to the Software or documentation. The Government's rights in Software and documentation shall be only those set forth in this Agreement.

SAS Institute Inc., SAS Campus Drive, Cary, North Carolina 27513-2414.

June 2014

SAS provides a complete selection of books and electronic products to help customers use SAS® software to its fullest potential. For more information about our offerings, visit [support.sas.com/bookstore](http://support.sas.com/bookstore) 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 trademarks of their respective companies.

# **Contents**

<b>About This Book .....</b>	<b>ix</b>
<b>About The Author .....</b>	<b>xiii</b>
<b>Acknowledgments .....</b>	<b>xv</b>
<b>Part 1: Understanding the Concepts and Features of Macro Programming .....</b>	<b>1</b>
<b>Chapter 1 Introduction .....</b>	<b>3</b>
What Is the SAS Macro Facility?.....	4
What Are the Advantages of the SAS Macro Facility? .....	5
Where Can the SAS Macro Facility Be Used? .....	11
Examples of the SAS Macro Facility .....	12
<b>Chapter 2 Mechanics of Macro Processing .....</b>	<b>21</b>
Introduction .....	21
The Vocabulary of SAS Processing.....	21
SAS Processing without Macro Activity .....	22
Understanding Tokens .....	23
Tokenizing a SAS Program .....	24
Comparing Macro Language Processing and SAS Language Processing.....	26
Processing a SAS Program That Contains Macro Language.....	27
<b>Chapter 3 Macro Variables .....</b>	<b>37</b>
Introduction .....	37
Basic Concepts of Macro Variables.....	38
Referencing Macro Variables.....	39
Understanding Macro Variable Resolution and the Use of Single and Double Quotation Marks.....	41

<b>Displaying Macro Variable Values .....</b>	<b>42</b>
Using the %PUT Statement .....	43
Displaying Macro Variable Values As They Resolve by Enabling the SYMBOLGEN System Option .....	46
<b>Understanding Automatic Macro Variables .....</b>	<b>48</b>
<b>Understanding User-Defined Macro Variables.....</b>	<b>51</b>
Creating Macro Variables with the %LET Statement.....	52
<b>Combining Macro Variables with Text.....</b>	<b>55</b>
Placing Text before a Macro Variable Reference .....	55
Placing Text after a Macro Variable Reference .....	57
Concatenating Permanent SAS Data Set Names and Catalog Names with Macro Variables.....	58
<b>Referencing Macro Variables Indirectly .....</b>	<b>60</b>
<b>Chapter 4 Macro Programs.....</b>	<b>67</b>
Introduction .....	67
Creating Macro Programs .....	68
Executing a Macro Program.....	74
Displaying Notes about Macro Program Compilation in the SAS Log.....	75
Displaying Messages about Macro Program Processing in the SAS Log .....	77
Using MPRINT to Display the SAS Statements Submitted by a Macro Program.....	77
Using the MLOGIC Option to Trace Execution of a Macro Program.....	78
Passing Values to a Macro Program through Macro Parameters.....	79
Specifying Positional Parameters in Macro Programs .....	80
Specifying Keyword Parameters in Macro Programs .....	82
Specifying Mixed Parameter Lists in Macro Programs.....	86
Defining a Macro Program That Can Accept a Varying Number of Parameter Values .	89
<b>Chapter 5 Understanding Macro Symbol Tables and the Processing of Macro Programs .....</b>	<b>95</b>
Understanding Macro Symbol Tables .....	95
Understanding the Global Macro Symbol Table .....	97
Understanding Local Macro Symbol Tables .....	101
Working with Global Macro Variables and Local Macro Variables .....	108
Defining the Domain of a Macro Variable by Using the %GLOBAL or %LOCAL Macro Language Statements.....	110
Processing of Macro Programs .....	114

How a Macro Program Is Compiled .....	114
How a Macro Program Executes .....	120
<b>Chapter 6 Macro Language Functions.....</b>	<b>129</b>
Introduction .....	129
Macro Character Functions .....	130
Macro Evaluation Functions .....	133
Macro Quoting Functions.....	135
Macro Variable Attribute Functions .....	137
Other Macro Functions.....	141
SAS Supplied Autocall Macro Programs Used Like Functions .....	147
<b>Chapter 7 Macro Expressions and Macro Programming Statements.....</b>	<b>151</b>
Introduction .....	151
Macro Language Statements.....	151
Constructing Macro Expressions .....	154
Understanding Arithmetic Expressions.....	155
Understanding Logical Expressions.....	156
Understanding the IN Operator As Used in Macro Language Statements .....	157
Conditional Processing with the Macro Language .....	158
Iterative Processing with the Macro Language .....	167
Writing Iterative %DO Loops in the Macro Language.....	167
Conditional Iteration with %DO %UNTIL.....	170
Conditional Iteration with %DO %WHILE .....	172
Branching in Macro Processing .....	174
<b>Chapter 8 Masking Special Characters and Mnemonic Operators .....</b>	<b>177</b>
Introduction .....	177
Why Are Quoting Functions Called Quoting Functions?.....	178
Illustrating the Need for Macro Quoting Functions .....	179
Describing the Commonly Used Macro Quoting Functions .....	180
Understanding How Macro Quoting Functions Work .....	181
Applying Macro Quoting Functions.....	182
Specifying Macro Program Parameters That Contain Special Characters or Mnemonic Operators .....	189
Unmasking Text and the %UNQUOTE Function.....	198
Using Quoting Versions of Macro Character Functions and Autocall Macro Programs ...	198

<b>Chapter 9 Interfaces to the Macro Facility.....</b>	<b>201</b>
Introduction .....	201
Understanding DATA Step Interfaces to the Macro Facility .....	201
Understanding the SYMGET Function .....	202
Understanding the SYMPUT and SYMPUTX Call Routines .....	209
Understanding the CALL EXECUTE Routine.....	216
Understanding the RESOLVE Function.....	227
Using Macro Facility Features in PROC SQL.....	231
Creating and Updating Macro Variables with PROC SQL .....	232
Using the Macro Variables Created by PROC SQL .....	238
Displaying Macro Option Settings with PROC SQL and Dictionary Tables .....	241
<b>Part 2: Applying Your Knowledge of Macro Programming .....</b>	<b>245</b>
<b>Chapter 10 Storing and Reusing Macro Programs.....</b>	<b>247</b>
Introduction .....	247
Saving Macro Programs with the Autocall Facility .....	248
Creating an Autocall Library .....	248
Making Autocall Libraries Available to Your Programs .....	250
Maintaining Access to the Autocall Macro Programs That Ship with SAS.....	251
Using the Autocall Facility under UNIX and z/OS Systems .....	252
Saving Macro Programs with the Stored Compiled Macro Facility .....	254
Setting SAS Options to Create Stored Compiled Macro Programs .....	254
Creating Stored Compiled Macro Programs.....	255
Saving and Retrieving the Source Code of a Stored Compiled Macro Program .....	257
Encrypting a Stored Compiled Macro Program.....	258
Resolving Macro Program References When Using the Autocall Facility and the Stored Compiled Macro Facility .....	258
<b>Chapter 11 Building a Library of Utilities .....</b>	<b>261</b>
Introduction .....	261
Writing a Macro Program to Behave Like a Function .....	261
Programming Routine Tasks.....	266

<b>Chapter 12 Debugging Macro Programming and Adding Error Checking to Macro Programs .....</b>	<b>273</b>
Introduction .....	273
Understanding the Types of Errors That Can Occur in Macro Programming .....	273
Minimizing Errors in Developing SAS Programs That Contain Macro Language .....	274
Categorizing and Checking for Common Problems in Macro Programming.....	275
Understanding the Tools That Can Debug Macro Programming .....	278
Using SAS System Options to Debug Macro Programming .....	278
Using Macro Language Statements to Debug Macro Programming.....	279
Using Macro Functions to Debug Macro Programming.....	280
Using Automatic Macro Variables to Debug Macro Programming.....	280
Examples of Solving Errors in Macro Programming .....	281
Improving Your Macro Programming by Including Error Checking .....	298
<b>Chapter 13 A Stepwise Method for Writing Macro Programs.....</b>	<b>307</b>
Introduction .....	307
Building a Macro Program in Four Steps .....	307
Applying the Four Steps to an Example.....	308
Step 1: Write, test, and debug the SAS program(s) that you want the macro program to build.....	309
Step 2: Remove hard-coded programming constants from the program(s) in Step 1 and replace these constants with macro variables.....	317
Step 3: Create macro program(s) from the program(s) in Step 2 .....	321
Step 4: Refine and generalize the macro program(s) in Step 3 by adding macro language statements like %IF-%THEN and %DO groups .....	325
Executing the REPORT Macro Program .....	328
Enhancing the Macro Program REPORT .....	338
<b>Part 3 Appendixes .....</b>	<b>339</b>
<b>Appendix A Sample Data Set.....</b>	<b>341</b>
<b>Appendix B Reference to Examples in This Book .....</b>	<b>343</b>
<b>Index .....</b>	<b>355</b>



# About This Book

---

## Purpose

The purpose of this book is to teach you how to incorporate macro programming features in your SAS programs. By learning how to use the SAS macro facility, you can accomplish repetitive programming tasks quickly and efficiently and your code may become easier to reuse and to understand.

---

## Is This Book for You?

This book is for SAS programmers who want to learn about SAS macro programming and its advantages.

---

## Prerequisites

This book assumes that you have beginning to intermediate experience writing SAS language programs. It does not review SAS language and SAS programming concepts. You should know how to write DATA steps and PROC steps and have some familiarity with Base SAS procedures.

---

## Scope of This Book

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

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.

This book describes the technical aspects of macro processing and these concepts are illustrated with examples that you can adapt for your applications. While understanding the technical aspects is not necessary to begin to reap the benefits of SAS macro programming, this knowledge might help you more wisely apply macro programming techniques.

Because macro facility features are interrelated, this book does not have to be read linearly. Work through sections as appropriate to your needs. Return to earlier sections when that information becomes pertinent.

The book is grouped into three sections. The first section explains the elements and mechanics of the macro programming language. The second part shows ways to apply the knowledge that you gained in the first part. The third part contains two appendices. One appendix lists the PROC CONTENTS for the data set used to illustrate the concepts. The second appendix provides a reference and description to the programs in this book.

---

## About the Examples

---

### Software Used to Develop the Book's Content

The examples in this book were tested using Base SAS 9.4 under Windows 7. The book describes a few features new in SAS 9.4. Most programs will also run in Base SAS 9.3.

---

### Example Code and Data

The examples in this book are illustrated with sales data from a fictitious bookstore. Appendix A presents a PROC CONTENTS of this data set. The DATA step to create this data set can be obtained from the link listed in the next paragraph. Appendix B provides a reference list to the programs in this book.

You can access the example code and data for this book by linking to its author page at <http://support.sas.com/publishing/authors>. Select the name of the author then select the “Example Code and Data” link under the book title to display the SAS programs that are included in this book.

For an alphabetical listing of all books for which example code and data is available, see <http://support.sas.com/bookcode>. Select a title to display the book’s example code.

If you are unable to access the code through the website, e-mail [saspress@sas.com](mailto:saspress@sas.com).

---

## Additional Resources

---

Check <http://support.sas.com> for additional information about and updates of the SAS macro facility and for papers contributed by SAS and SAS users.

The SAS Support Communities website is a forum where you can exchange information about SAS programming with other users: <https://communities.sas.com>.

Look on this website for other books published by SAS and SAS Press that discuss macro programming features: <https://support.sas.com/publishing/>.

---

## Keep in Touch

We look forward to hearing from you. We invite questions, comments, and concerns. If you want to contact us about a specific book, please include the book title in your correspondence to [saspress@sas.com](mailto:saspress@sas.com).

---

### To Contact the Author through SAS Press

By e-mail: [saspress@sas.com](mailto:saspress@sas.com)

Via the Web: [http://support.sas.com/author\\_feedback](http://support.sas.com/author_feedback)

---

## SAS Books

For a complete list of books available through SAS, visit <http://support.sas.com/bookstore>.

Phone: 1-800-727-3228

Fax: 1-919-677-8166

E-mail: [sasbook@sas.com](mailto:sasbook@sas.com)

---

## SAS Book Report

Receive up-to-date information about all new SAS publications via e-mail by subscribing to the SAS Book Report monthly eNewsletter. Visit <http://support.sas.com/sbr>.

---

## Publish with SAS

SAS is recruiting authors! Are you interested in writing a book? Visit <http://support.sas.com/saspress> for more information.



## About The Author



Michele M. Burlew designs and programs SAS applications for data management, data analysis, report writing, and graphics for academic and corporate clients. A SAS user since 1980, she has expertise in many SAS products and operating systems. Burlew is the author of seven SAS Press books: *SAS Hash Object Programming Made Easy*; *Combining and Modifying SAS Data Sets: Examples, Second Edition*; *Output Delivery System: The Basics and Beyond* (coauthor); *SAS Guide to Report Writing: Examples, Second Edition*; *SAS Macro Programming Made Easy, Third Edition*; *Debugging SAS Programs: A Handbook of Tools and Techniques*; and *Reading External Data Files Using SAS: Examples Handbook*.

Learn more about this author by visiting her author page at:

<http://support.sas.com/burlew>. There you can download free book excerpts, access example code and data, read the latest reviews, get updates, and more.



## **Acknowledgments**

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

Thanks to my editors at SAS, John West and Julie Platt, for their guidance. Thanks to Shelley Sessoms for review of my proposal to write a third edition. I appreciate the opportunity to write a third edition on a subject that I really like.

Thanks to the technical reviewers, Kay Alden, Art Carpenter, Pat Garrett, Ginny Piechota, Kevin Russell, and Russ Tyndall for their careful review of the material and useful suggestions.

Thanks to the SAS Publishing copyedit and production team for their hard work in the layout, formatting, figure redesign, copyediting, and marketing of this book. The team members contributing to this book include Sallie Fiore for copyediting, Robert Harris for the update of the figures for the third edition, Denise Jones for formatting, Aimee Rodriguez for marketing, and Stacy Suggs for cover design.

As in the first and second editions, 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. They were a great group to work with, and the programming skills I learned from them have helped me over and over again.



# **Part 1: Understanding the Concepts and Features of Macro Programming**

<b>Chapter 1</b>	<b>Introduction</b>	<b>3</b>
<b>Chapter 2</b>	<b>Mechanics of Macro Processing</b>	<b>21</b>
<b>Chapter 3</b>	<b>Macro Variables</b>	<b>37</b>
<b>Chapter 4</b>	<b>Macro Programs</b>	<b>67</b>
<b>Chapter 5</b>	<b>Understanding Macro Symbol Tables and the Processing of Macro Programs</b>	<b>95</b>
<b>Chapter 6</b>	<b>Macro Language Functions</b>	<b>129</b>
<b>Chapter 7</b>	<b>Macro Expressions and Macro Programming Statements</b>	<b>151</b>
<b>Chapter 8</b>	<b>Masking Special Characters and Mnemonic Operators</b>	<b>177</b>
<b>Chapter 9</b>	<b>Interfaces to the Macro Facility</b>	<b>201</b>



# **Chapter 1 Introduction**

<b>What Is the SAS Macro Facility? .....</b>	<b>4</b>
<b>What Are the Advantages of the SAS Macro Facility? .....</b>	<b>5</b>
<b>Where Can the SAS Macro Facility Be Used? .....</b>	<b>11</b>
<b>Examples of the SAS Macro Facility .....</b>	<b>12</b>

Imagine you have an assistant who helps 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.

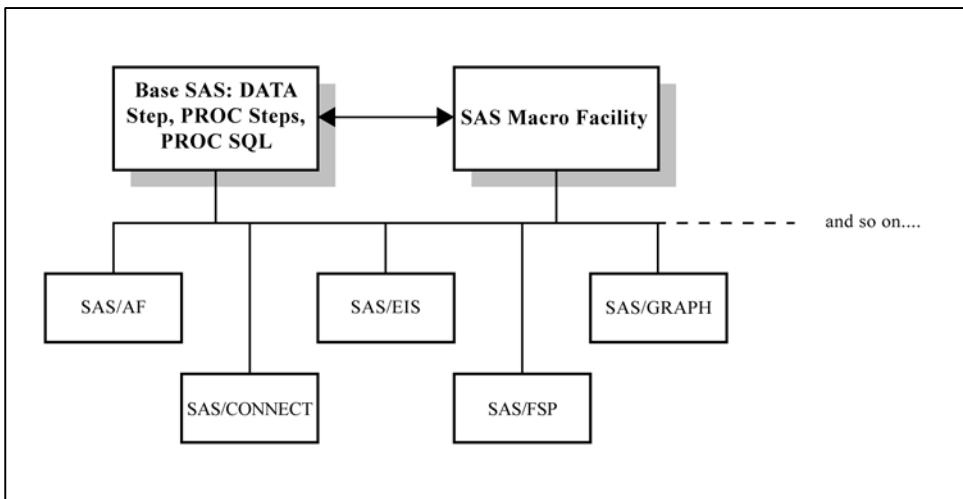
## What Is the SAS Macro Facility?

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 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. Additionally, if you use SAS Enterprise Guide, you can add macro facility features when specifying projects and tasks. Indeed, many of the SAS products that you license contain programs that make use of 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 Example 1.1 is used to select a subset of a data set and to place information in the PROC PRINT report title. Macro language and macro variable references are in bold.

```
%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 Example 1.2 executes, it submits a PROC MEANS step three times: once for each of the years 2014, 2015, and 2016. Each time, PROC MEANS processes a different data set. The macro language and references that generate the three steps are in bold.

```
%macro sales;
%do year=2014 %to 2016;
    proc means data=books.sold&year;
        title "Sales Information for &year";
        class section;
        var listprice saleprice;
    run;
%end;
%mend sales;
%sales
```

SAS first released the macro facility in SAS 82.0 in 1982. SAS macro language has relatively few statements, 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 and versatile 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, when you incorporate 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 your SAS program can reference that macro program. The reference to the macro program is similar to calling a subroutine. Your 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 a 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, the bank initiates 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 utility 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 retailer.

Example 1.3 produces two reports for the computing-related book sales. The first is a monthly sales report. The second is a panel 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.

```
%let repmonth=4;
%let repyear=2014;
%let repmword=%sysfunc(mdy(&repmonth,1,&repyear),monname9.);

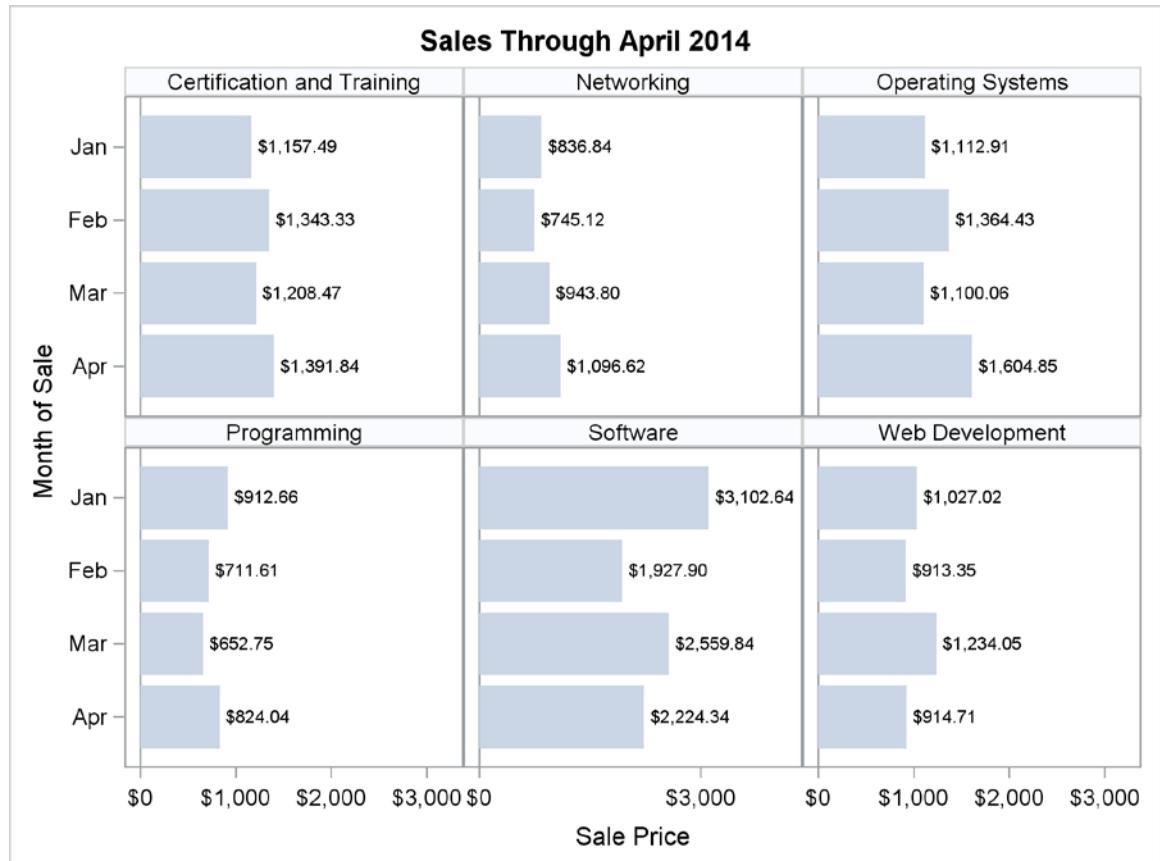
proc tabulate data=books.ytdsales;
  where month(datesold)=&repmonth and year(datesold)=&repyear;
  title "Sales During &repmword &repyear";
  class section;
  var saleprice listprice cost;
  tables section all='**TOTAL**',
    (saleprice listprice cost)*(n*f=4. sum*f=dollar10.2);
run;

proc sgpanel data=books.ytdsales;
  where month(datesold)<=&repmonth and
    year(datesold)=&repyear;
  title "Sales Through &repmword &repyear";
  panelby section;
  hbar datesold / stat=sum fill datalabel response=saleprice;
  format datesold monname3.;
run;
```

Output 1.3 presents the output from Example 1.3.

**Output 1.3 Output from Example 1.3****Sales During April 2014**

	Sale Price		List Price		Wholesale Cost	
	N	Sum	N	Sum	N	Sum
<b>Section</b>						
Certification and Training	44	\$1,391.84	44	\$1,637.40	44	\$654.96
Networking	45	\$1,096.62	45	\$1,433.31	45	\$573.32
Operating Systems	57	\$1,604.85	57	\$2,359.95	57	\$943.98
Programming	28	\$824.04	28	\$1,077.16	28	\$430.86
Software	64	\$2,224.34	64	\$2,616.80	64	\$1,046.72
Web Development	43	\$914.71	43	\$1,195.53	43	\$478.21
<b>**TOTAL**</b>	281	\$8,056.40	281	\$10,320.15	281	\$4,128.06



Changing just the first line of the program from

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

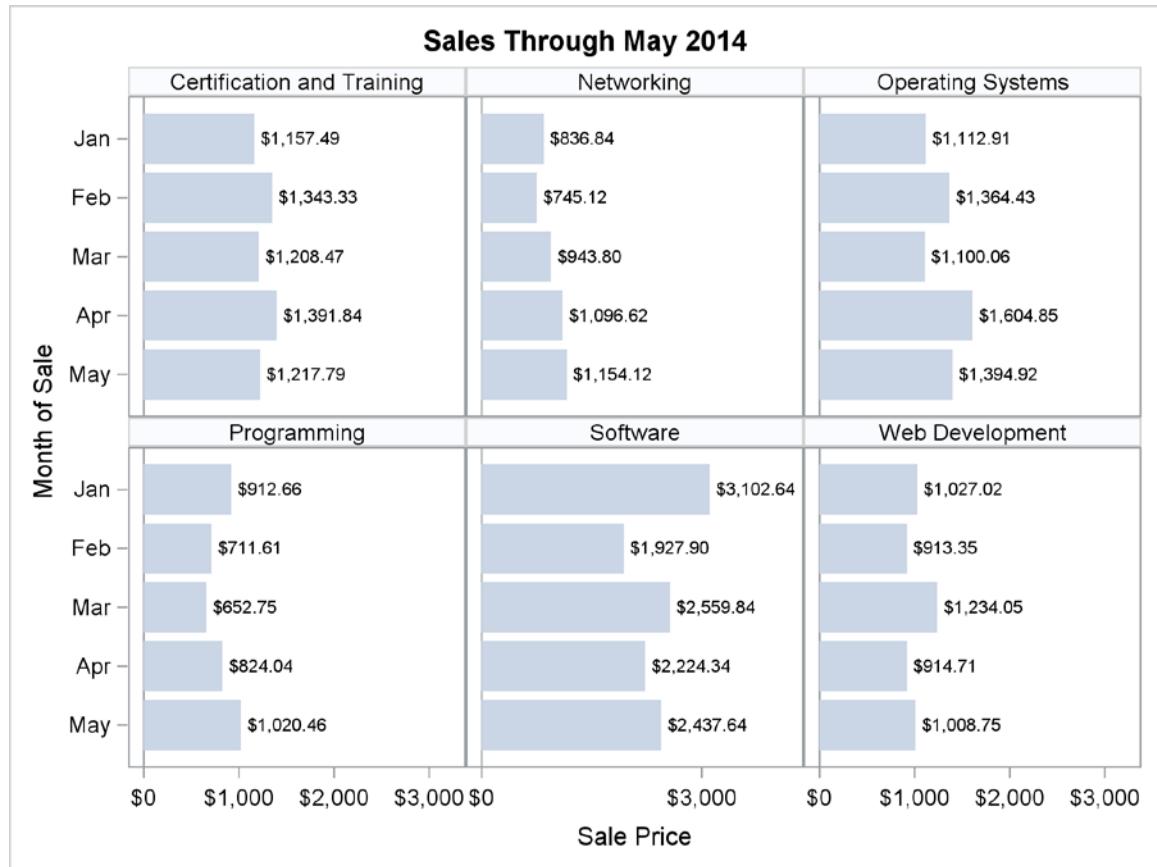
to

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

runs the same program, but now the program processes the data collected for May. No other editing of the program is required to process this subset. Output 1.4 presents the output for May.

**Output 1.4 Output from revised Example 1.3****Sales During May 2014**

	Sale Price		List Price		Wholesale Cost	
	N	Sum	N	Sum	N	Sum
<b>Section</b>						
Certification and Training	39	\$1,217.79	39	\$1,432.65	39	\$573.06
Networking	50	\$1,154.12	50	\$1,508.46	50	\$603.38
Operating Systems	50	\$1,394.92	50	\$2,051.26	50	\$820.50
Programming	34	\$1,020.46	34	\$1,333.90	34	\$533.56
Software	72	\$2,437.64	72	\$2,867.76	72	\$1,147.10
Web Development	48	\$1,008.75	48	\$1,318.44	48	\$527.38
<b>**TOTAL**</b>	293	\$8,233.68	293	\$10,512.47	293	\$4,204.99




---

## Where Can the SAS Macro Facility Be Used?

The macro facility can be used with most 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/GRAFH	Contains libraries of macro routines for annotating SAS/GRAFH output

## 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 you can delegate to the macro processor.

In addition to the examples that follow, Example 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.

Example 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, 2014.

```
title "Sales Report";
title2 "As of &sysdate &sysday &sysdate9";
title3 "Using SAS Version: &sysver";
proc means data=books.ytdsales n sum;
    var saleprice;
run;
```

Output 1.5 presents the output from Example 1.4.

#### Output 1.5 Output from Example 1.4

<b>Sales Report</b>	
<b>As of 15:51 Thursday 20FEB2014</b>	
<b>Using SAS Version: 9.4</b>	
<b>The MEANS Procedure</b>	
<b>Analysis Variable : saleprice</b>	
<b>Sale Price</b>	
<b>N</b>	<b>Sum</b>
470	13620.70

#### 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 that a SAS program should execute. These conditional processing macro language statements are similar in syntax and structure to SAS language statements.

Macro program DAILY in Example 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 22, 2014.

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

```
%macro daily;
  proc means data=books.ytdsales maxdec=2 sum;
    where datesold=today();
```

```

title "Daily Sales Report for &sysdate";
class section;
var saleprice;
run;
%if &sysday=Friday %then %do;
  proc means data=books.ytdsales maxdec=2 sum;
    where today()-6 le datesold le today();
    title "Weekly Sales Report Week Ending &sysdate9";
    class section;
    var saleprice;
  run;
%end;
%mend daily;

%daily

```

Output 1.6 presents the output from Example 1.5.

#### **Output 1.6 Output from Example 1.5**

### **Daily Sales Report for 22AUG2014**

#### The MEANS Procedure

<b>Analysis Variable : saleprice Sale Price</b>		
<b>Section</b>	<b>N Obs</b>	<b>Sum</b>
Certification and Training	2	69.62
Networking	1	19.12
Operating Systems	2	53.68
Software	2	67.10
Web Development	3	64.95

## Weekly Sales Report Week Ending 22AUG2014

### The MEANS Procedure

Analysis Variable : saleprice Sale Price		
Section	N Obs	Sum
Certification and Training	8	278.48
Networking	8	191.06
Operating Systems	14	411.04
Programming	2	48.94
Software	13	430.64
Web Development	12	253.71

### Example 1.6: Iterative Processing of SAS Steps

Coding each iteration of a programming process that contains multiple iterations can be 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.

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 1.6 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.

```
%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. With this technique, you can avoid the tedious task of entering all the data set names and all the ELSE statements.

### **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 and call routines that interact with the macro facility can transfer information between steps in your SAS programs.

Example 1.7 calculates total sales for two sections in the retailer's inventory. That value is then inserted in the TITLE statement of the PROC SGLOT output. The SYMPUTX SAS language routine instructs the macro processor to retain the total sales value in macro variable SOFTPROGSALES after the DATA step finishes. The total sales value is then available to subsequent steps in the program.

```
data temp;
  set books.ytdsales end=lastobs;
  retain sumsoftprog 0;
  if section in ('Software','Programming') then
    sumsoftprog=sumsoftprog + saleprice;
```

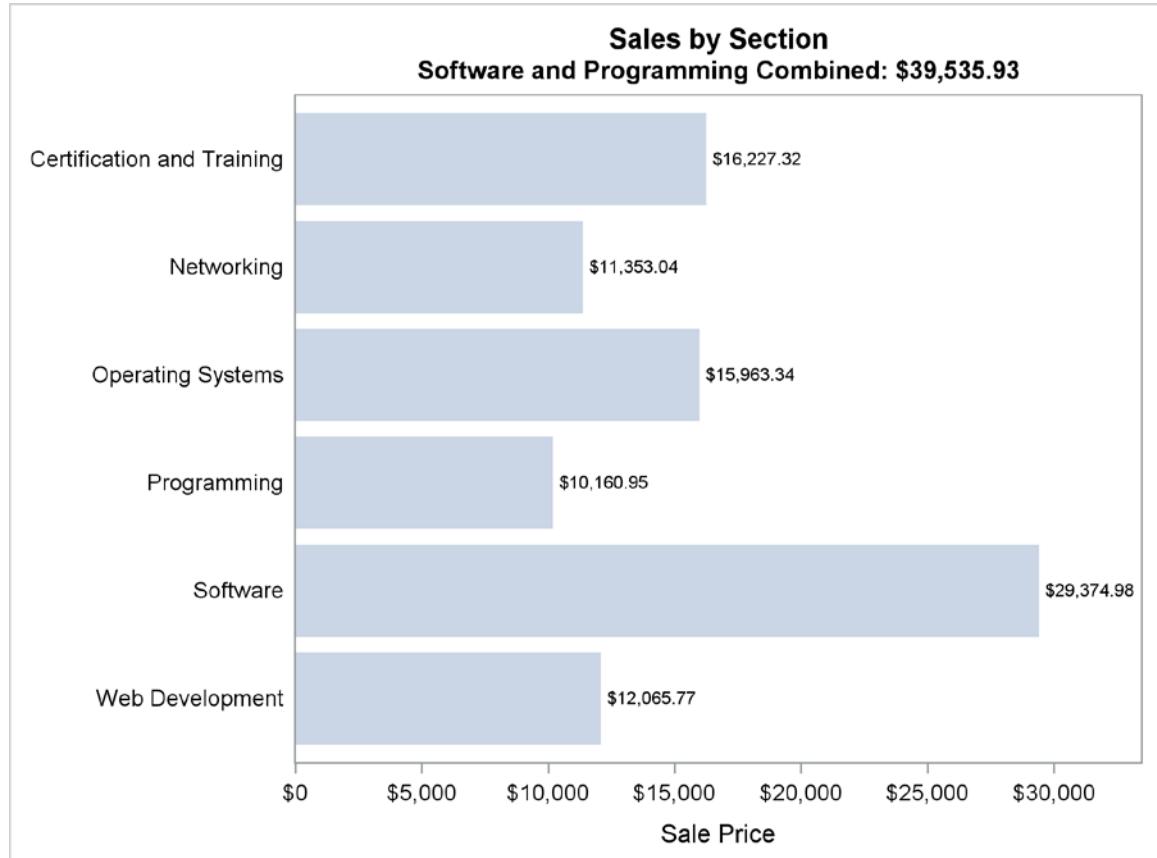
```

if lastobs then
  call symputx('softprogsales',put(sumsoftprog,dollar10.2));
run;
proc sgplot data=temp;
  title "Sales by Section";
  title2 "Software and Programming Combined: &softprogsales";
  hbar section / stat=sum fill datalabel response=saleprice;
  yaxis display=(nolabel noticks);
  format saleprice dollar10.2;
run;

```

Output 1.7 presents the output from Example 1.7.

#### Output 1.7 Output from Example 1.7



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.

Alternatively, you can use PROC SQL to compute the same sum of sales for the software and programming sections and save that sum in a macro variable for use later. The following PROC SQL step uses the INTO clause to create macro variable SUMSOFTPROG.

```
proc sql noprint;
  select sum(saleprice) format=dollar10.2 into :sumsoftprog
  from books.ytdsales
  where section in ('Software','Programming');
quit;
```

The PROC SQL step does not create a table so the DATA= option on the PROC SGPLOT step should specify BOOKS.YTDSALES instead of TEMP.

### **Example 1.8: Interfacing Macro Language and SAS Language Functions**

SAS has libraries of functions that you can also use in your macro language code. 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 Example 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 SAS inserts these attributes in the report title. The PUTN SAS language function formats the date value. Macro program DSREPORT opens and closes the data set with the OPEN and CLOSE SAS language functions. The SAS language functions are underlined in Example 1.8.

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

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

  %*----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: &nlobs 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 Example 1.8.

#### **Output 1.8 Output from Example 1.8**

**Report on Data Set books.ytdsales**  
**Num Obs: 3346 Date Created: 10JAN2015**

**The MEANS Procedure**

Analysis Variable : saleprice Sale Price		
Section	N Obs	Sum
Certification and Training	524	16227.32
Networking	479	11353.04
Operating Systems	578	15963.34
Programming	337	10160.95
Software	857	29374.98
Web Development	571	12065.77

#### **Example 1.9: Building and Saving a Library of Utility Routines**

Your work might frequently require that you 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 even to save the compiled code in permanent locations. For example, perhaps some of your reports 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 macro programs in a special location called an *autocall* library. Then when you want to submit one of these saved macro programs for compilation and execution during a later SAS session, you only need to tell SAS to look for macro programs in your autocall library, and you do not have to explicitly submit the code in the SAS session. Further, you could make the library

available to your workgroup, and thus your workgroup only needs to maintain single copies of macro programs that you all need to use.

The macro program STANDARDOPTS in Example 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. (If you were using UNIX, make sure the filename is in lowercase.)

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

In a later SAS session, you do not need to explicitly submit the macro program definition. Instead, when you submit the following OPTIONS statement and call macro program STANDARDOPTS, SAS finds the definition of STANDARDOPTS in the c:\mymacroprograms directory, compiles it for you, and executes 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

```
Book Sales Report
```

If STANDARDOPTS was submitted on February 22, 2014, from a SAS session that started at 8:36 using SAS 9.4, the footnote text on subsequent reports would be:

```
Prepared Friday 22FEB2014 08:36 using SAS 9.4
```

# **Chapter 2 Mechanics of Macro Processing**

<b>Introduction .....</b>	<b>21</b>
<b>The Vocabulary of SAS Processing.....</b>	<b>21</b>
<b>SAS Processing without Macro Activity.....</b>	<b>22</b>
<b>Understanding Tokens .....</b>	<b>23</b>
<b>Tokenizing a SAS Program .....</b>	<b>24</b>
<b>Comparing Macro Language Processing and SAS Language Processing.....</b>	<b>26</b>
<b>Processing a SAS Program That Contains Macro Language.....</b>	<b>27</b>

---

## **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**

This chapter uses several terms to describe SAS processing. Table 2.1 reviews these terms.

**Table 2.1 Terms commonly used to describe SAS processing**

Term	Description
<b>input stack</b>	Holds a SAS program after it is submitted and before it is processed by the word scanner.
<b>word scanner</b>	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.
<b>token</b>	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.
<b>compiler</b>	Checks the syntax of tokens received from the word scanner. After the compiler completes checking the syntax, it translates the tokens into a form for execution.
<b>macro processor</b>	Processes macro language references and statements.
<b>macro trigger</b>	The symbols & and %, when followed by a letter or underscore, that signal the word scanner to transfer what follows to the macro processor.
<b>macro symbol table</b>	The area in memory where macro variables and their associated values are stored. (If option MSYMTABMAX is set to 0, SAS writes macro symbol tables to disk.)

---

## SAS Processing without Macro Activity

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

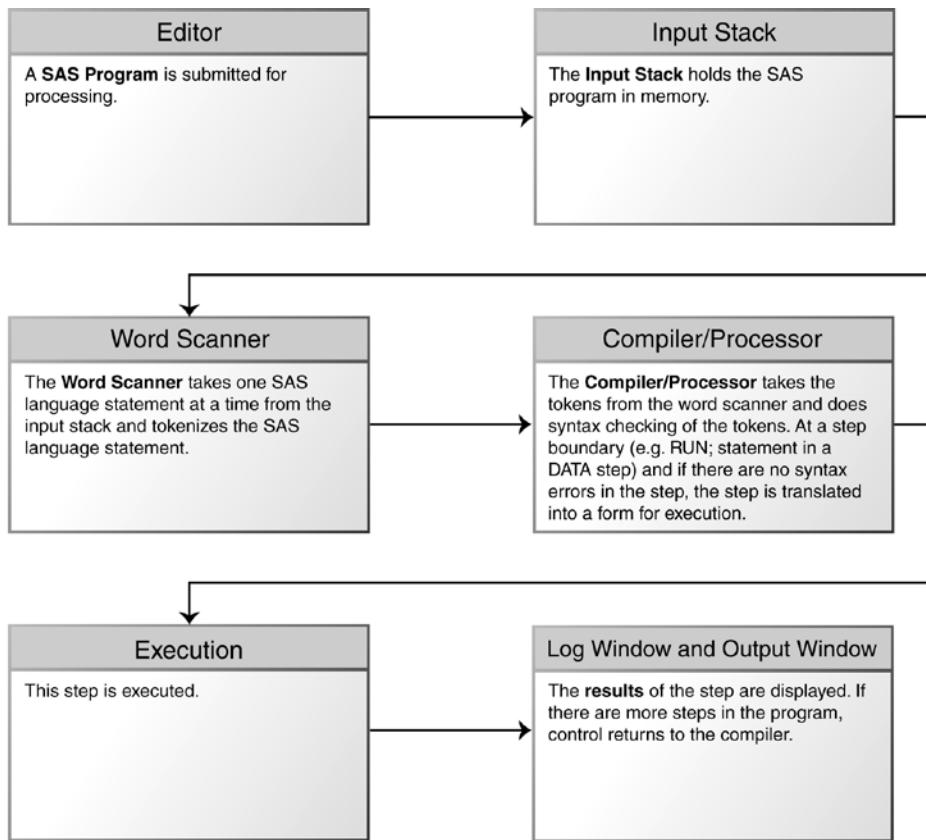
- an interactive SAS session from the Editor
- a batch program
- a noninteractive program
- SAS Enterprise Guide
- from the command line in the SAS windowing environment
- an SCL SUBMIT block
- the SCL COMPILE command

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
<b>literal</b>	A string of characters enclosed in single or double quotation marks.	'My program text' "My program text"
<b>numbers</b>	A string of digits including integers, decimal values, and exponential notation. SAS dates, SAS times, and hexadecimal constants are also number tokens.	123456 '30APR1982'D 3. '01'x 6.023E23
<b>names</b>	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_ if ssn. mmddyy10. descending var1 var2 var3 var4 var5
<b>special</b>	Characters other than a letter, number, or underscore that have a special meaning to SAS.	; + - * / ** ( ) {} & %

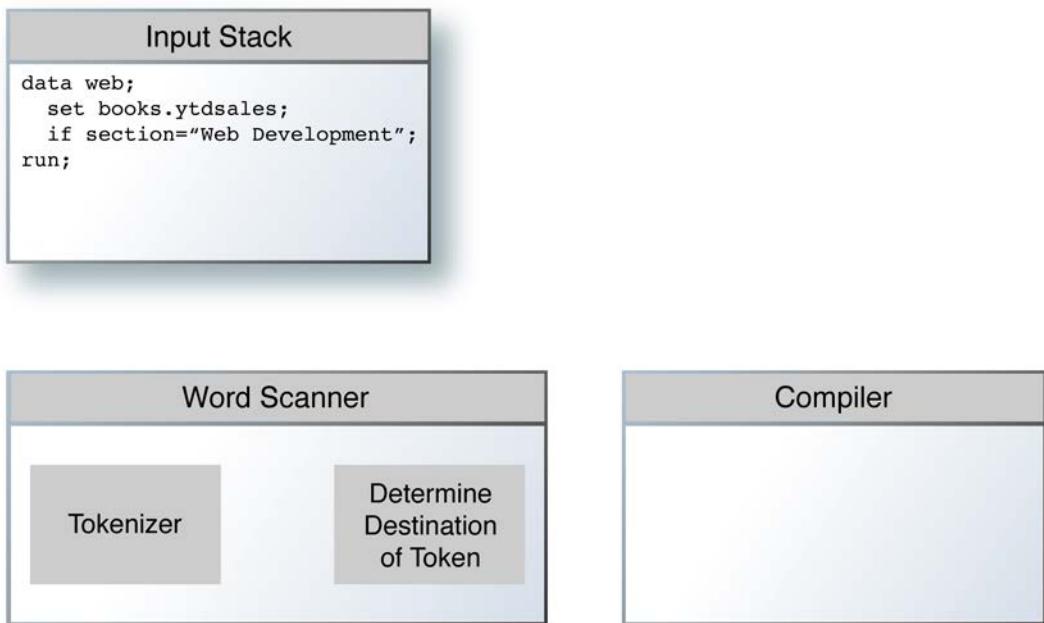
The importance of understanding tokenization is evident if you have ever dealt with unmatched quotation marks in your SAS language statements. Matched quotation marks delimit a literal token. When you omit a closing quotation mark, SAS continues to add text to your literal token beyond what you intended. The SAS programming statements added to the literal token never get tokenized by the word scanner. SAS issues a warning once a character string reaches 262 characters. 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.

---

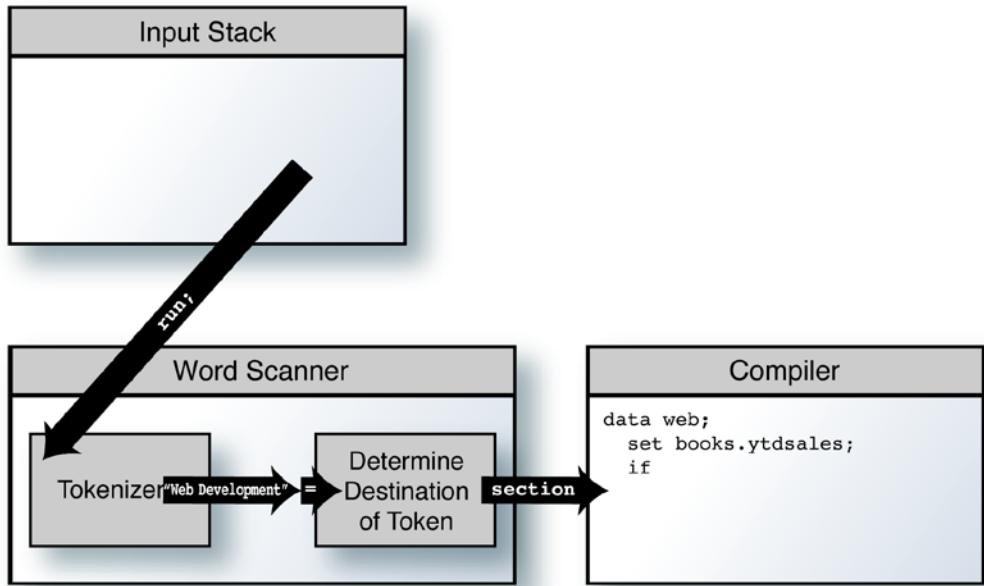
## Tokenizing a SAS Program

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 Word Scanner receives the semicolon following the RUN statement, it stops sending tokens to the compiler. 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 SAS will tokenize, compile, and execute.

Recall that SAS compiles and executes programs the same way. After you submit a SAS program, 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.

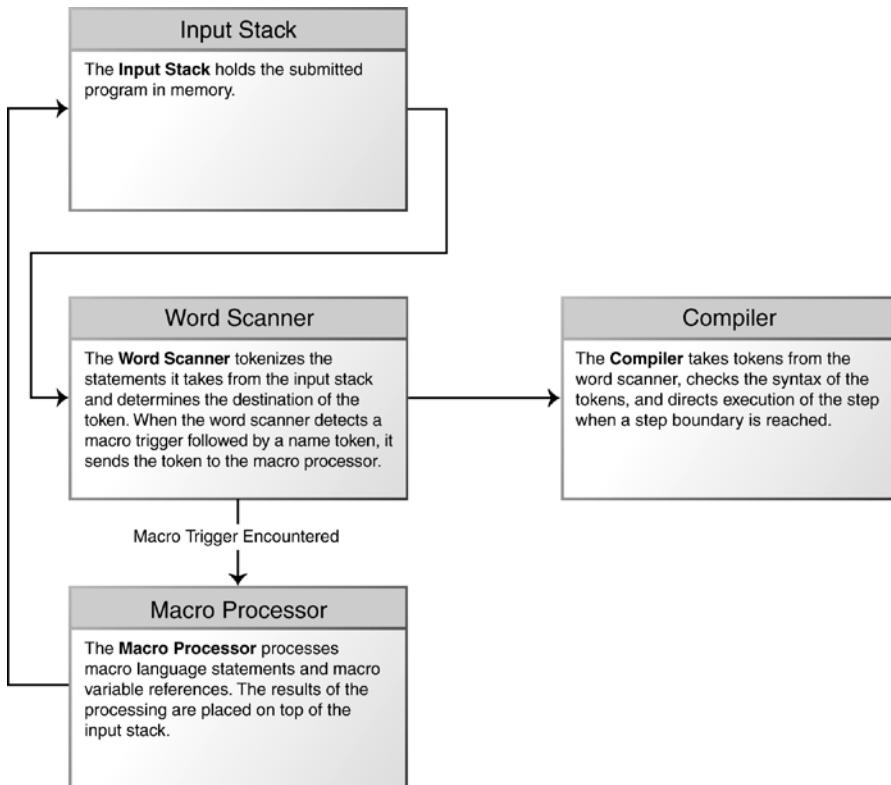
When the word scanner detects a macro trigger followed by a name token, it sends what follows to the macro processor and temporarily turns processing over to the macro processor. The word scanner suspends tokenization while the macro processor completes its job. Therefore, processing of a macro language 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 sends non-macro text, such as SAS language statements, back to the input stack. The macro processor puts the non-macro text that it builds back on top of the input stack. The word scanner then resumes its work by tokenizing the non-macro text it received 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 it detects that the macro reference has been 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 Development" are written to the output data set.

The value of a macro variable that you create is stored in the macro symbol table for the duration of the SAS session or until you explicitly delete the macro variable. When a SAS session starts, SAS automatically defines several macro variables. These automatic macro variables are also stored in the macro symbol table. You cannot delete these macro variables that SAS defines. 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**

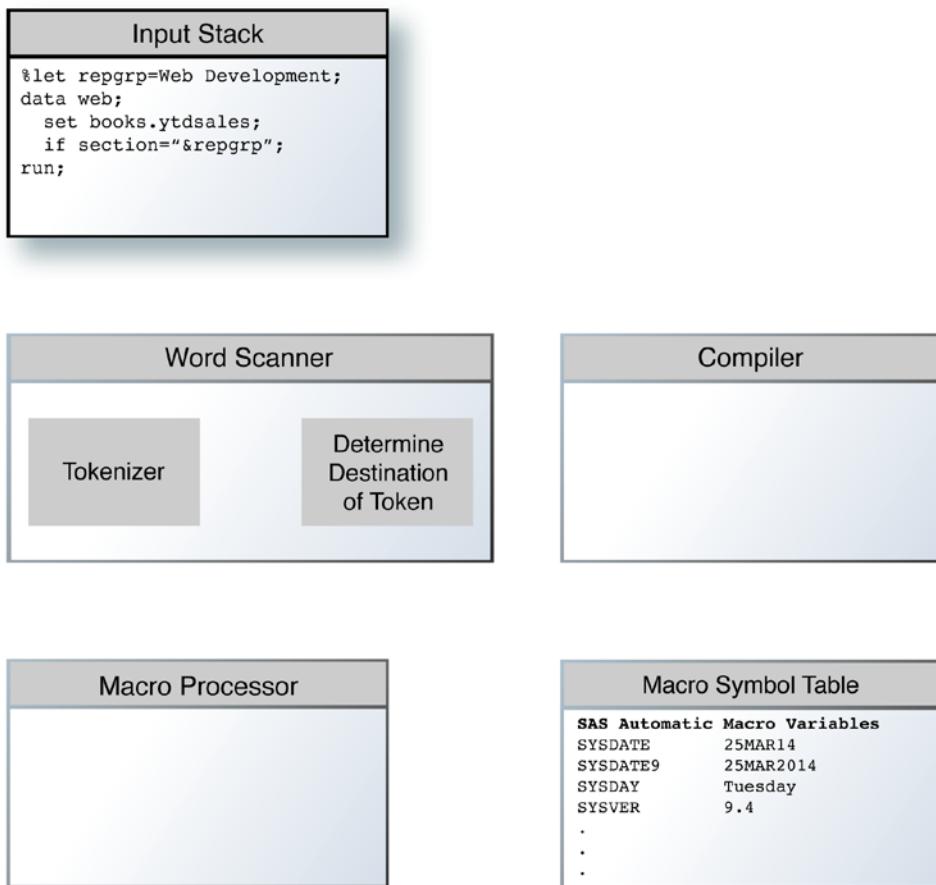
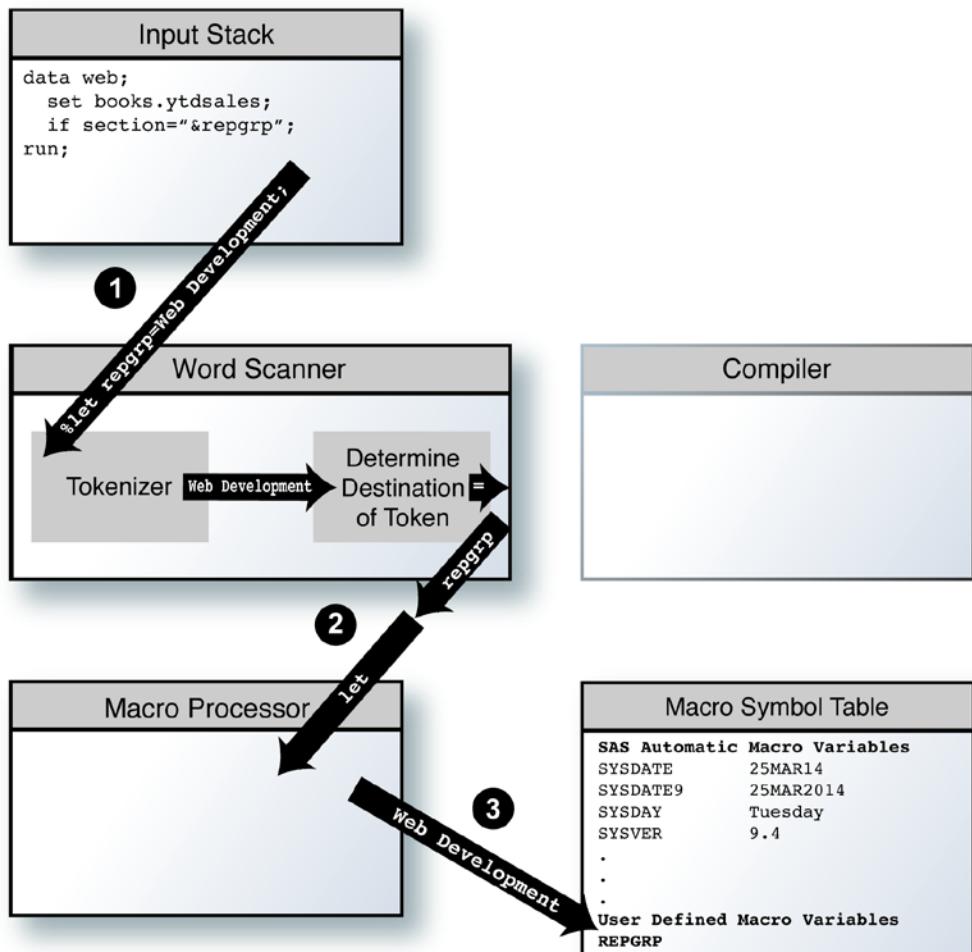


Figure 2.6 shows how a macro language statement is taken from the input stack, tokenized by the word scanner, and passed to the macro processor.

**Figure 2.6 A macro language statement is processed**



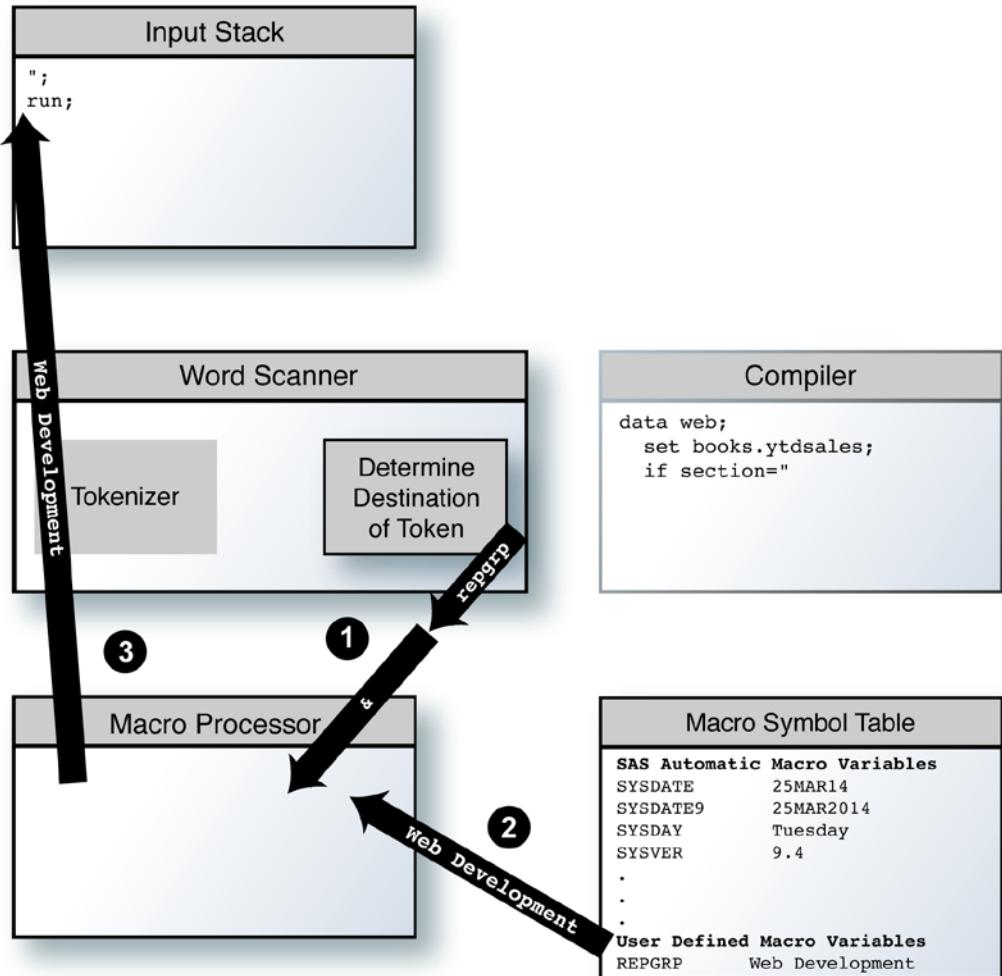
The three steps in Figure 2.6 are:

- ❶ Statements are taken from the input stack one at a time. The %LET statement is the first one to be tokenized 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 macro processor takes over and keeps requesting tokens until it encounters the semicolon (;) that terminates the %LET statement. The word scanner then stops sending tokens to the macro processor.
- ❸ Last, the macro processor executes the %LET statement, and it places the macro variable REPGRP and its associated text, *Web Development*, in the macro symbol table.

No quotation marks enclose the literal token *Web Development*. 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

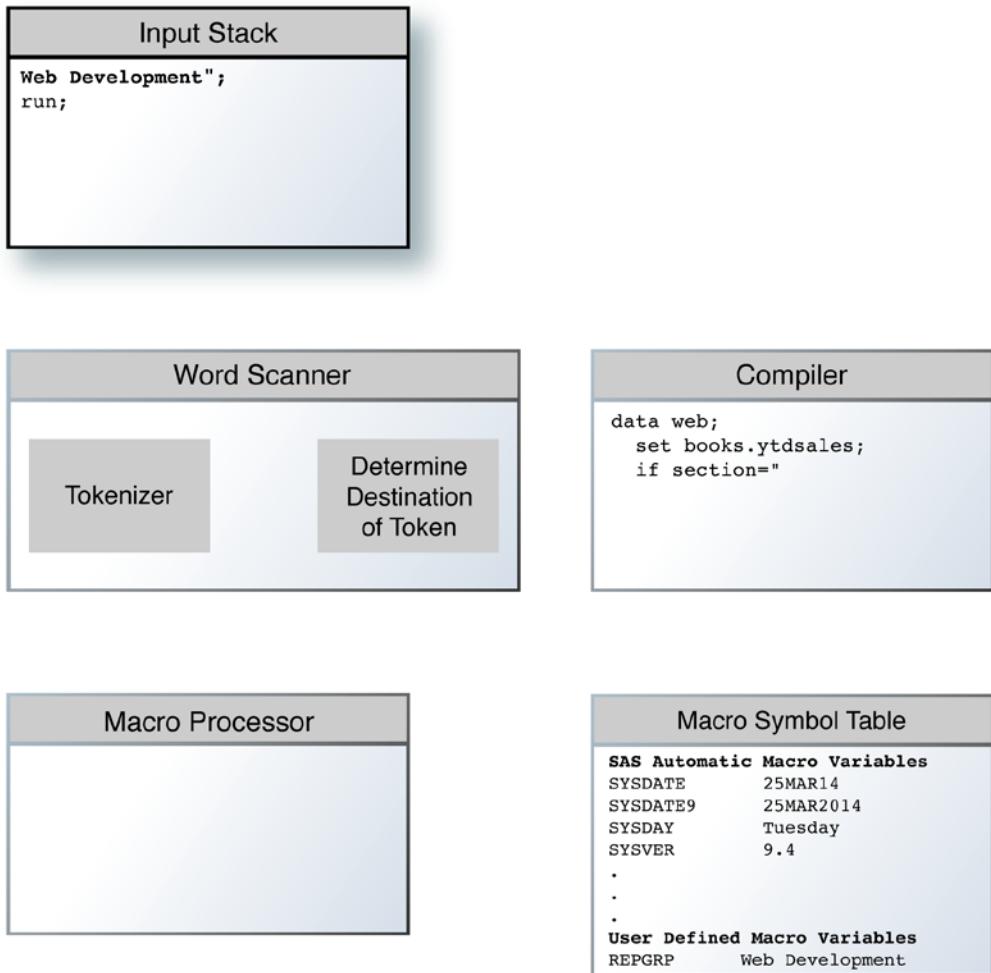


The three steps in Figure 2.7 are:

- ❶ When the word scanner encounters the ampersand (&) followed by REPGRP, it sends tokens 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.

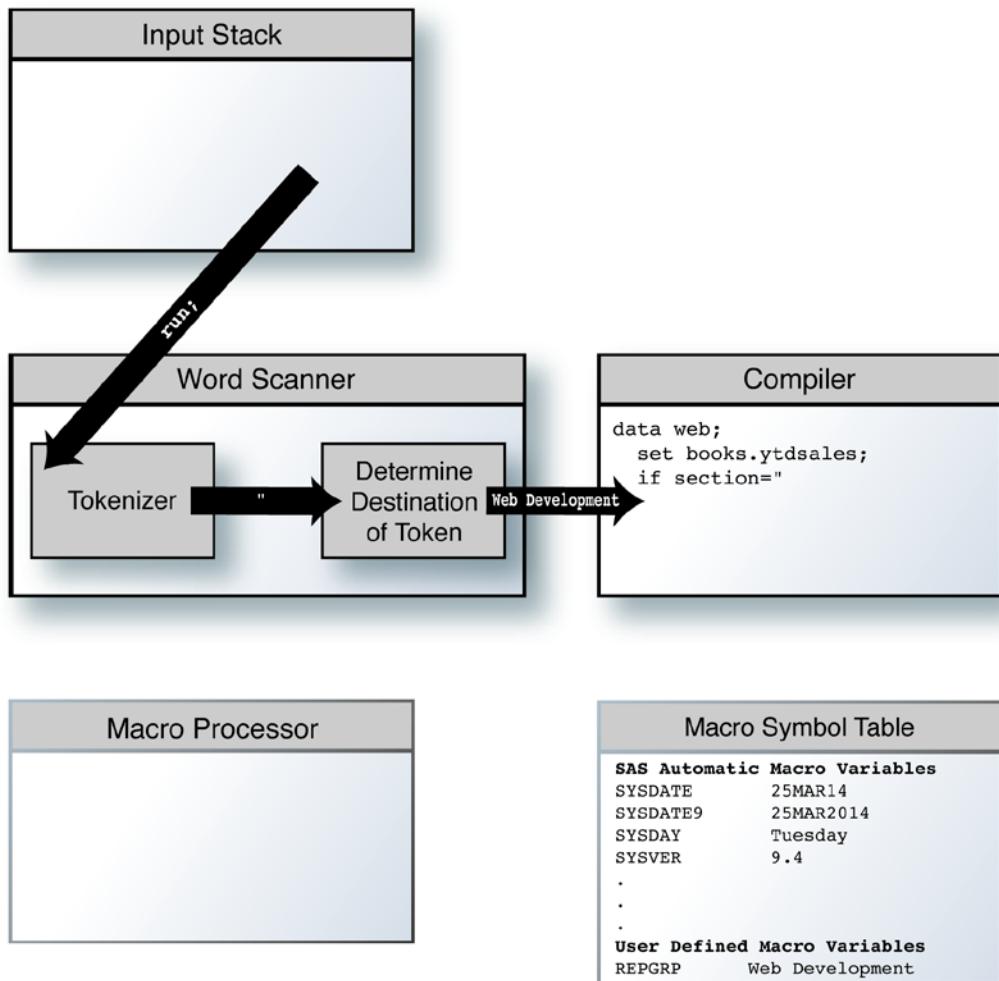
The value of the macro variable REPGRP is now on top of the input stack as shown in Figure 2.8. Remember that the macro variable reference in the SAS language IF statement was enclosed in double quotation marks. Therefore, the value of the macro variable is treated as one literal token in the IF statement.

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



Next, the value of macro variable REPGRP, Web Development, is transferred from the input stack 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 2.9.

**Figure 2.9 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 that terminates the DATA step causes SAS to execute the compiled DATA step code.

In conclusion, when you write 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 syntax is compiled and executed before SAS language is compiled and executed. The discussion on how SAS processes macro language continues in Chapter 5.

# **Chapter 3 Macro Variables**

<b>Introduction .....</b>	<b>37</b>
<b>Basic Concepts of Macro Variables.....</b>	<b>38</b>
<b>Referencing Macro Variables .....</b>	<b>39</b>
<b>Understanding Macro Variable Resolution and the Use of Single and Double Quotation Marks.....</b>	<b>41</b>
<b>Displaying Macro Variable Values .....</b>	<b>42</b>
Using the %PUT Statement.....	43
Displaying Macro Variable Values As They Resolve by Enabling the SYMBOLGEN System Option .....	46
<b>Understanding Automatic Macro Variables .....</b>	<b>48</b>
<b>Understanding User-Defined Macro Variables.....</b>	<b>51</b>
Creating Macro Variables with the %LET Statement .....	52
<b>Combining Macro Variables with Text .....</b>	<b>55</b>
Placing Text before a Macro Variable Reference.....	55
Placing Text after a Macro Variable Reference.....	57
Concatenating Permanent SAS Data Set Names and Catalog Names with Macro Variables .....	58
<b>Referencing Macro Variables Indirectly .....</b>	<b>60</b>

---

## **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. SAS defines automatic macro variables each time a SAS session starts, and these variables 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. SAS updates others during your SAS session.

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, this chapter's 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 also stores the automatic macro variables it creates 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 or your code changes it. SAS deletes the local symbol table, and thus the local macro variable, when the macro program that created the local macro variable 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 usually have to 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: a value in lowercase remains lowercase, uppercase remains uppercase, and mixed case remains mixed case.

SAS generally removes leading and trailing blanks from the macro variable value before the value is placed in the macro symbol table unless the value is masked by special macro functions. (Chapters 6 and 8 describe these functions.)

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.**

The name you give a macro variable can be up to 32 characters in length in SAS®9. This 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 *SAS Macro Language: Reference*, which can be found in Documentation in the Knowledge Base, at <http://support.sas.com>.

- **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 they 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 you want SAS to resolve a macro variable reference in a SAS language statement, enclose the reference in double quotation marks; if you enclose the reference in single quotation marks, SAS will not resolve the reference. The next section describes the use of quotation marks.

### Example 3.1: Defining and Referencing Macro Variables

The two %LET statements in Example 3.1 create two macro variables, REPTITLE and REPVAR, in open code and assign values to the variables. (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. Example 3.1 also references automatic macro variable SYSDAY that SAS defined automatically at the start of the SAS session.

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

```
%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 Example 3.1 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. The following program adds a PROC MEANS step to the program shown above.

```
%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;
```

Example 3.1 defines the two user-defined macro variables REPTITLE and REPVAR and assigns them values only once, but the program references them more than once. After the macro processor resolves the macro variable references, and assuming the revised program was submitted on a Friday, the program that SAS 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 in double quotation marks. SAS does not resolve a macro variable reference enclosed within single quotation marks. 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 Example 3.2 encloses text in double quotation marks. SAS resolves the macro variable references on that statement. Single quotation marks enclose the text in the second TITLE statement. SAS does not resolve the macro variable references on that statement.

```

%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 Example 3.2 was submitted Friday, May 2, 2014, 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 SAS displays no warnings or error messages. Output 3.1 presents the output from the PROC MEANS step in Example 3.2.

#### Output 3.1 Output for Example 3.2

### Means by &reptitle as of &sysday

#### The MEANS Procedure

Analysis Variable : saleprice Sale Price		
Section	N Obs	Sum
Certification and Training	163	5152.11
Networking	154	3721.72
Operating Systems	189	5235.93
Programming	104	3150.00
Software	289	9987.98
Web Development	197	4171.67

## 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 text or 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. 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 next five options are keywords that list different classifications of macro variables.

The last three options are keywords that can simulate color-coded SAS messages. The ‘...’ represents the text that you want the %PUT statement to display following the keyword.

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 and their values.
_LOCAL_	Lists user-defined local macro variables and their values. 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 and their values.
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 ERROR, NOTE, and WARNING messages that are generated by SAS. Note that the keywords must be capitalized and must terminate with a colon (:). You can also terminate the keyword with a dash (-) instead of a colon. A dash tells SAS that this is a continuation line, and SAS does not display the keyword. 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 Example 3.3 lists the values of all the automatic macro variables.

```
%put _automatic_;
```

An excerpt of the SAS log after submitting Example 3.3 from the Editor of a SAS®9 session under Windows 7 follows.

```
AUTOMATIC AFDSID 0
AUTOMATIC AFDSNAME
AUTOMATIC AFLIB
AUTOMATIC AFSTR1
AUTOMATIC AFSTR2
AUTOMATIC FSPBDV
AUTOMATIC SYSADDBITS 32
AUTOMATIC SYSBUFFR
AUTOMATIC SYSCC 0
AUTOMATIC SYSCHARWIDTH 1
AUTOMATIC SYSCMD
AUTOMATIC SYSDATE 20MAR14
AUTOMATIC SYSDATE9 20MAR2014
AUTOMATIC SYSDAY Thursday
AUTOMATIC SYSDEVIC
AUTOMATIC SYSDMG 0
AUTOMATIC SYSSDN      _NULL_
AUTOMATIC SYSENCODING wlatin1
AUTOMATIC SYSENDIAN LITTLE
AUTOMATIC SYSENV FORE
AUTOMATIC SYSERR 0
AUTOMATIC SYSERRORTEXT
AUTOMATIC SYSFILRC 0
```

**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. A %PUT \_LOCAL\_ statement must be inside the macro program where you want to examine the local macro variables.

The \_GLOBAL\_ option on the %PUT statement in Example 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”.

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

The SAS log from these statements follows.

```
88  %let reptitle=Book Section;
89  %let repvar=section;
90  %put _global_;
GLOBAL REPTITLE Book Section
GLOBAL REPVAR section
```

The macro facility also has two SAS functions, %SYMLOCAL and %SYMGLOBL, that can check whether a specific macro variable is local or global. These functions can be useful when you write macro programs, which is the topic of Chapter 4.

**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. Example 3.1 is modified below in Example 3.5 to include four %PUT statements. The first three %PUT statements mix text and macro variable values. The fourth %PUT statement shows how you can write a %PUT statement so that the macro variable name is included in the display. This action of this last %PUT statement is similar to the PUT variable-name= syntax of the PUT SAS language statement. This feature became available in SAS 9.3

```
%let reptitle=Book Section;
%let repvar=section;
%put My macro variable REPTITLE has the value &reptitle;
%put My macro variable REPVAR has the value &repvar;
%put Automatic macro variable SYSDAY has the value &sysday;
%put &=reptitle &=repvar &=sysday;
title "Frequencies by &reptitle as of &sysday";
proc freq data=books.ytdsales;
  tables &repvar;
run;
```

The SAS log from Example 3.5 follows:

```

57  %let reptitle=Book Section;
58  %let repvar=section;
59
60  %put My macro variable REPTITLE has the value &reptitle;
My macro variable REPTITLE has the value Book Section
61  %put My macro variable REPVAR has the value &repvar;
My macro variable REPVAR has the value section
62  %put Automatic macro variable SYSDAY has the value &sysday;
Automatic macro variable SYSDAY has the value Wednesday
63  %put &=reptitle &=repvar &=sysday;
REPTITLE=Book Section REPVAR=section SYSDAY=Wednesday
64
65  title "Frequencies by &reptitle as of &sysday";
66  proc freq data=books.ytdsales;
67    tables &repvar;
68  run;
```

NOTE: There were 3346 observations read from the data set  
BOOKS.YTDSALES.

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 you include in SAS language statements.

---

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

The most useful SAS option for you to use as you start writing programs that contain macro variables is SYMBOLGEN. 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. The SYMBOLGEN option displays macro variable values only when they are referenced; macro variable values are not displayed at the time they are created with %LET. SAS decides whether SYMBOLGEN displays the values before or after the macro variable reference.

**Example 3.6: Displaying Macro Variable Values with the SYMBOLGEN Option**

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

```
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 sum maxdec=2;
  class &repvar;
  var saleprice;
run;
```

The SAS log for Example 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 3346 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 3346 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 global macro symbol table. Other than in data lines, you can use these macro variables anywhere in your SAS programs. Typically, SAS uses automatic macro variables 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. For a more complete list, see *SAS Macro Language: Reference*. There are three types of automatic macro variables. Table 3.2 groups the macro variables 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 SAS can change these values.
- The values of the third type are initialized at the start of the SAS session and either you or SAS can modify these values.

**Table 3.2 A sample of automatic macro variables**

Type of Automatic Macro Variable	Automatic Macro Variable	Description
<b>Values that remain fixed</b>	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
<b>Values that SAS can change</b>	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
	SYSMACRONAME	returns the name of the currently executing macro program
	SYSNOBS	contains the number of observations read from the last data set that was closed by the previous procedure or DATA step
	SYSRC	returns a value corresponding to an error condition
	SYSWARNINGTEXT	text of the last warning message generated within the SAS log
<b>Values that SAS or you can change</b>	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.

SAS uses the SYS prefix for automatic macro variables. It also uses the SQL prefix for automatic macro variables it creates when PROC SQL executes. Avoid using these prefixes when you create 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.

### **Example 3.7: Using Automatic Variables**

Example 3.7 references five automatic macro variables. The automatic macro variable names are in bold. Assume that the program was run on June 22, 2014. The program extracts a subset of observations from BOOK.YTDSALES for the Web Development section for books sold from June 16–June 22, 2014, and PROC PRINT lists the selected observations.

```
data lastweek;
  set books.ytdsales;
  where section='Web Development' and datesold > "&sysdate"d-6;
run;

proc print data=lastweek;
  title "Web Development 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 Example 3.7.

**Output 3.2 Partial output from Example 3.7**

**Web Development Titles Sold in the Past Week**  
**Report Date: Sunday 22JUN14 09:03**

Obs	booktitle	datesold	saleprice
1	Web Development Title 2	06/17/2014	\$17.59
2	Web Development Title 3	06/18/2014	\$23.68
3	Web Development Title 5	06/19/2014	\$17.59
4	Web Development Title 5	06/21/2014	\$23.68
5	Web Development Title 5	06/22/2014	\$23.68
6	Web Development Title 7	06/19/2014	\$17.59
7	Web Development Title 8	06/21/2014	\$17.59
8	Web Development Title 9	06/19/2014	\$23.68
9	Web Development Title 9	06/17/2014	\$23.68
10	Web Development Title 13	06/21/2014	\$17.59
11	Web Development Title 14	06/22/2014	\$23.68

Data Set Used: WORK.LASTWEEK SAS Version: 9.4

## Understanding User-Defined Macro Variables

The applications of user-defined macro variables are limitless. Other than in data lines, you can create and reference user-defined macro variables 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

The program in Example 3.8 shows several ways of using the %LET statement. 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 the first few statements demonstrate.

Each illustration starts with one or more %LET statements followed by the SAS log that shows resolution of the macro variables after executing %PUT statements that list macro variable values. The illustrations are separated by a dashed line. Numbers identify some of the code, and description of these annotations follows the program.

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

100  %let nocalc=53*21 + 100.1;      ①
101  %put &nocalc;
NOCALC=53*21 + 100.1
-----
%let value1=982;
%let value2=813;
%let result=&value1 + &value2;
```

```

103 %let value1=982;
104 %put &value1;
VALUE1=982
105 %let value2=813;
106 %put &value2;
VALUE2=813
107 %let result=&value1 + &value2;      ①
108 %put &result;
RESULT=982 + 813
-----
%let reptext=This report is for *** Department XYZ ***;

110 %let reptext=This report is for *** Department XYZ ***;
111 %put &=reptext;
REPTEXT=This report is for *** Department XYZ ***
-----
%let region=Region 3;
%let text=Sales Report;
%let moretext="Sales Report";
%let reptitle=&text &region;
%let reptitle2=&moretext &region;

113 %let region=Region 3;
114 %put &=region;
REGION=Region 3
115 %let text=Sales Report;    ②
116 %put &=text;
TEXT=Sales Report
117 %let moretext="Sales Report";
118 %put &=moretext;
MORETEXT="Sales Report"
119 %let reptitle=&text &region;
120 %put &=reptitle;
REPTITLE=Sales Report Region 3
121 %let reptitle2=&moretext &region;
122 %put &=reptitle2;
REPTITLE2="Sales Report" Region 3
-----
%let sentence=      This one started with leading blanks.;

124 %let sentence=      This one started with leading blanks.; ③
125 %put Now no leading blanks:&sentence;
Now no leading blanks:This one started with leading blanks.
-----
```

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

127 %let chars=Symbols: !@#$%^&*;      ④
128 %put &chars;
Symbols: !@#$%^&*
-----
%let novalue=;

130 %let novalue=;      ⑤
131 %put &=novalue;
NOVALUE=
-----
%let holdvars=varnames;
%let &holdvars=title author datesold;

133 %let holdvars=varnames;      ⑥
134 %put &holdvars;
varnames
135 %let &holdvars=title author datesold;  ⑥
136 %put &holdvars;
varnames
137 %put &varnames;
title author datesold
```

Text was added to the %PUT statement for the macro variable SENTENCE to more clearly show that the leading blanks were removed. General observations to make from the %LET assignments in Example 3.8 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 100 and SAS log line 107)
- ❷ When assigning values to macro variables, quotation marks are not used to enclose the value (SAS log lines 113 and 115). When quotation marks are used, the quotation marks become part of the text associated with the macro variables. (SAS log line 117) (The use of quotation marks in macro programming is a special topic in the macro facility, and Chapter 8 discusses this topic.)
- ❸ Leading blanks are removed from the value of a macro variable. (SAS log line 124)

- ④ 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 127)
  - ⑤ A macro variable can have a null value. (SAS log line 130)
  - ⑥ 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 133 and 135)
- 

## 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

Example 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.

```
%let mosold=4;
%let level=12;

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 book412;
  set books.ytdsales(where=(month(datesold)=4));

  attrib over12 length=$3 label="Cost > $12";
  if cost > 12 then over12='YES';
  else over12='NO';
run;
proc freq data=book412;
  title "Frequency Count of Books Sold During Month 4";
  title2 "Grouped by Cost Over $12";
  tables over12;
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=15;
```

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

```
data book1215;
  set books.ytdsales(where=(month(datesold)=12));

  attrib over15 length=$3 label="Cost > $15";
  if cost > 15 then over15='YES';
  else over15='NO';
run;
proc freq data=book1215;
  title "Frequency Count of Books Sold During Month 12";
  title2 "Grouped by Cost Over $15";
  tables over15;
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. While not required unless you follow a macro variable reference with text, all macro variable references can be terminated with periods.

The code above 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 Example 3.10. 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. Example 3.10 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. This first program in Example 3.10 does not execute.

```
*----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 program does not define the five macro variables PREFIX1, PREFIX2, PREFIX3, PREFIX4, and PREFIX5, 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.

The program is revised below. This newer version contains the necessary delimiters that tell the macro processor when the macro variable references end. Now the macro variable references resolve as desired, and the text that follows the references is concatenated to the results of the resolution.

```
*----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**

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

```
*----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 SAS completely tokenizes the PROC FREQ statement. In the program above, 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 the period terminates 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 can 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 the program below. 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.

```
*----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 reread the reference until no more ampersands can be resolved.

### Example 3.12: Resolving Two Ampersands That Precede a Macro Variable Reference

The first example of referencing macro variables indirectly follows in Example 3.12. Six macro variables define six sections in the computer department of the retailer. A report program analyzes sales information for a section. The macro variable `N` represents the section number. Example 3.12 produces the sales information for Section 3, Operating Systems.

The indirect macro variable reference in Example 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.

```
%let section1=Certification and Training;
%let section2=Networking;
%let section3=Operating Systems;
%let section4=Programming;
%let section5=Software;
%let section6=Web Development;
```

```

-----Look for section number defined by macro var n;
%let n=3;
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;

```

### **Example 3.13: Illustrating the Resolution of Macro Variable References When Combining Them**

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

The macro processor resolves &SECTION&N in two parts: &SECTION and &N. &N can be resolved, and in this example, &N equals 3. The macro variable &SECTION is not defined and cannot be resolved, thus causing SAS to write a warning message to the SAS log. The following statements demonstrate how &SECTION&N does not resolve as desired.

```

options symbolgen;
%let section3=Operating Systems;
%let n=3;

%put &section&n;

```

The SAS log for Program 3.13a follows.

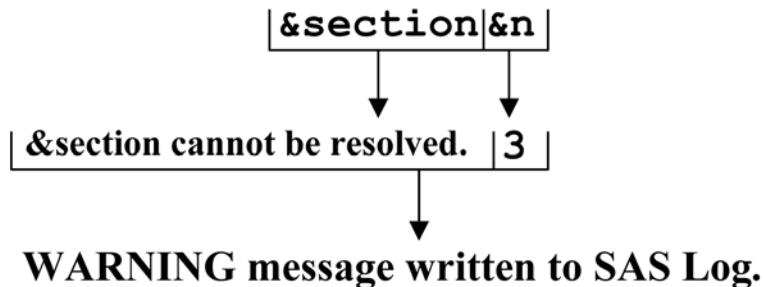
```

1   options symbolgen;
2   %let section3=Operating Systems;
3   %let n=3;
4
5   %put &section&n;
WARNING: Apparent symbolic reference SECTION not resolved.
SYMBOLGEN: Macro variable N resolves to 3
&section3

```

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

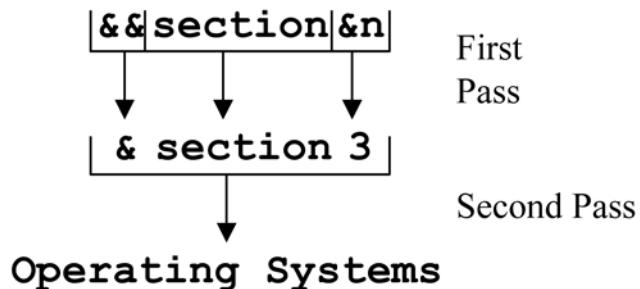
**Figure 3.1 How the macro processor resolves the two concatenated macro variable references in Example 3.13**



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 3, yielding `&SECTION3`. On the second pass, the macro variable reference `&SECTION3` 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**



The statements are modified below to include another ampersand before SECTION.

```
options symbolgen;
%let section3=Operating Systems;
%let n=3;

%put &&section&n;
```

The SAS log for these statements follows

```
6   options symbolgen;
7   %let section3=Operating Systems;
8   %let n=3;
9
10  %put &&section&n;
SYMBOLGEN:  && resolves to &.
SYMBOLGEN:  Macro variable N resolves to 3
SYMBOLGEN:  Macro variable SECTION3 resolves to Operating Systems
Operating Systems
```

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

```
131 options symbolgen;
132 %let section1=Certification and Training;
133 %let section2=Networking;
134 %let section3=Operating Systems;
135 %let section4=Programming;
136 %let section5=Software;
137 %let section6=Web Development;
138 *----Look for section number defined by macro var n;
139 %let n=3;
140 proc means data=books.ytdsales;
SYMBOLGEN:  && resolves to &.
SYMBOLGEN:  Macro variable N resolves to 3
SYMBOLGEN:  Macro variable SECTION3 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 3
SYMBOLGEN:  Macro variable SECTION3 resolves to Operating Systems
143   var saleprice;
144 run;

NOTE: There were 578 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
```

**Example 3.14: Resolving Multiple Ampersands That Precede a Macro Variable**

Example 3.14 illustrates how the macro processor resolves multiple ampersands preceding a macro variable reference. Three ampersands precede the macro variable reference in this example.

Example 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 3, Operating Systems.

```

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 n=3;
%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 Example 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=Operating Systems;
33   %let section4=Programming;
34   %let section5=Software;
35   %let section6=Web Development;
36
37   %let n=3;
38   %let wherevar=Section;
39
40   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 3
SYMBOLGEN: Macro variable SECTION3 resolves to Operating Systems
41   title "Sales for &wherevar: &&&wherevar&n";
SYMBOLGEN: Macro variable WHEREVAR resolves to Section
42   where &wherevar="&&&wherevar&n";

```

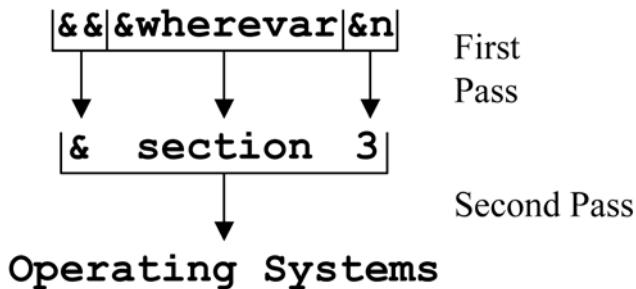
```

SYMBOLGEN:  && resolves to &.
SYMBOLGEN:  Macro variable WHEREVAR resolves to Section
SYMBOLGEN:  Macro variable N resolves to 3
SYMBOLGEN:  Macro variable SECTION3 resolves to Operating Systems
46      var saleprice;
47      run;

NOTE: There were 578 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**

<b>Introduction .....</b>	<b>67</b>
<b>Creating Macro Programs .....</b>	<b>68</b>
<b>Executing a Macro Program .....</b>	<b>74</b>
<b>Displaying Notes about Macro Program Compilation in the SAS Log .....</b>	<b>75</b>
<b>Displaying Messages about Macro Program Processing in the SAS Log .....</b>	<b>77</b>
Using MPRINT to Display the SAS Statements Submitted by a Macro Program.....	77
Using the MLOGIC Option to Trace Execution of a Macro Program.....	78
<b>Passing Values to a Macro Program through Macro Parameters.....</b>	<b>79</b>
Specifying Positional Parameters in Macro Programs.....	80
Specifying Keyword Parameters in Macro Programs .....	82
Specifying Mixed Parameter Lists in Macro Programs.....	86
Defining a Macro Program That Can Accept a Varying Number of Parameter Values.....	89

---

## **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 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.
<i>program</i>	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.
< <i>parameter-list</i> >	The macro program name must not be a reserved word in the macro facility. (See <i>SAS Macro Language: Reference</i> , which can be found in Documentation in the Knowledge Base, at <a href="http://support.sas.com">http://support.sas.com</a> .) Note that the macro program name is not preceded with a percent sign on the %MACRO statement.
	Names one or more macro variables local to the macro program 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.

---

Macro Program Element	Description
</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 <i>command-style macros</i> . For additional information on this option, refer to <i>SAS Macro Language: Reference</i> .
DES='text'	Specifies a description (up to 40 characters) for the macro program entry in the macro catalog.
MINDELIMITER= 'single character'	Specifies a value that will override the value of the MINDELIMITER= global SAS option only during execution of the macro program. The delimiter separates elements of an expression when using the IN operator. The value must be a single character enclosed in single quotation marks. The default is a blank. See MINOPERATOR option below.
MINOPERATOR/ NOMINOPERATOR	Specifies whether the macro processor recognizes and evaluates the mnemonic IN and the special character # as logical operators when evaluating arithmetic or logical expressions during the execution of the macro program. The setting of this argument overrides the setting of the NOMINOPERATOR global system option.
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 <i>statement-style macros</i> . For additional information on this option, refer to <i>SAS Macro Language: Reference</i> .

---

---

<b>Macro Program Element</b>	<b>Description</b>
</option(s)>	SECURE/NOSECURE  Encrypts the contents of a macro program when SAS stores it in a stored compiled macro library. By default, SAS does not encrypt macro programs. The NOSECURE option is a convenience feature that can be used when you develop macro program code. After code has been tested, a global edit of the macro program library can change the text from NOSECURE to SECURE.
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 on the %MACRO statement and the MSTORED SAS system option be set. See Chapter 10 and <i>SAS Macro Language: Reference</i> for more information.
STORE	Stores the compiled macro program as an entry in a SAS catalog in a permanent SAS data library. When using the STORE option, the SASMSTORE= SAS system option, must also be specified. See Chapter 10 and <i>SAS Macro Language: Reference</i> for more information
<text>	is any combination of <ul style="list-style-type: none"> <li>• text strings</li> <li>• macro variables, macro functions, or macro language statements</li> <li>• SAS programming statements</li> </ul> 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, and recommended so that your code is easier to read.

---

### Example 4.1: Demonstrating a Macro Program

Example 4.1 shows an example of a macro program definition. Macro program PROFITCHART produces a horizontal bar chart that analyzes profit for the current week by bookstore section.

```
%macro profitchart;
  ods graphics / maxlengendarea=0;

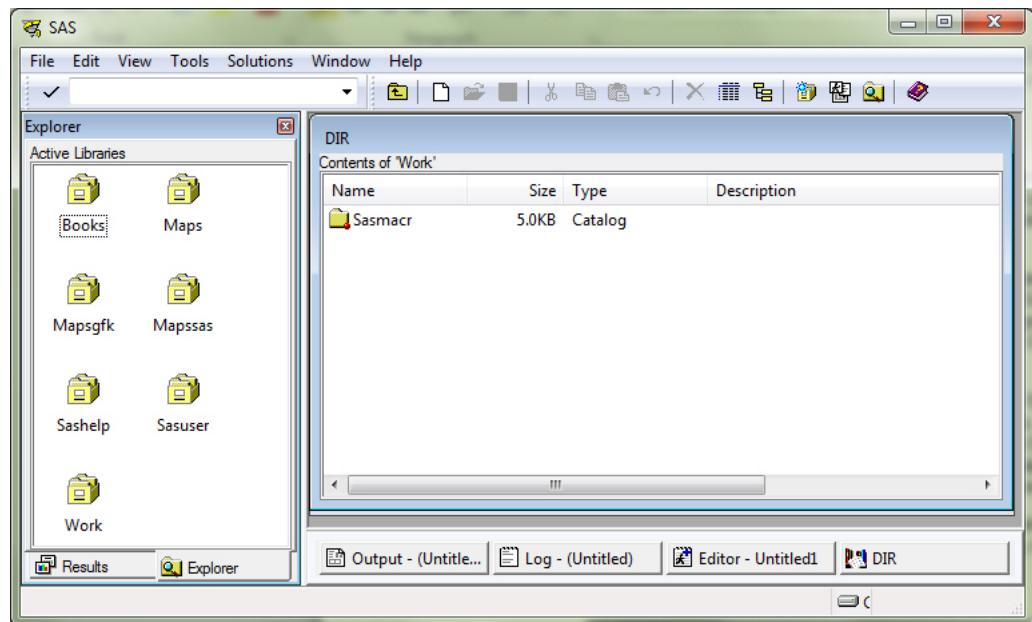
  title "Profit Report for Week Ending &sysdate9";
  proc sgplot data=temp;
    where today()-6 <=datesold <= today();
    hbar section / response=profit group=section stat=sum
      datalabel;
    yaxis label=' ';
  run;
%mend profitchart;
```

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.

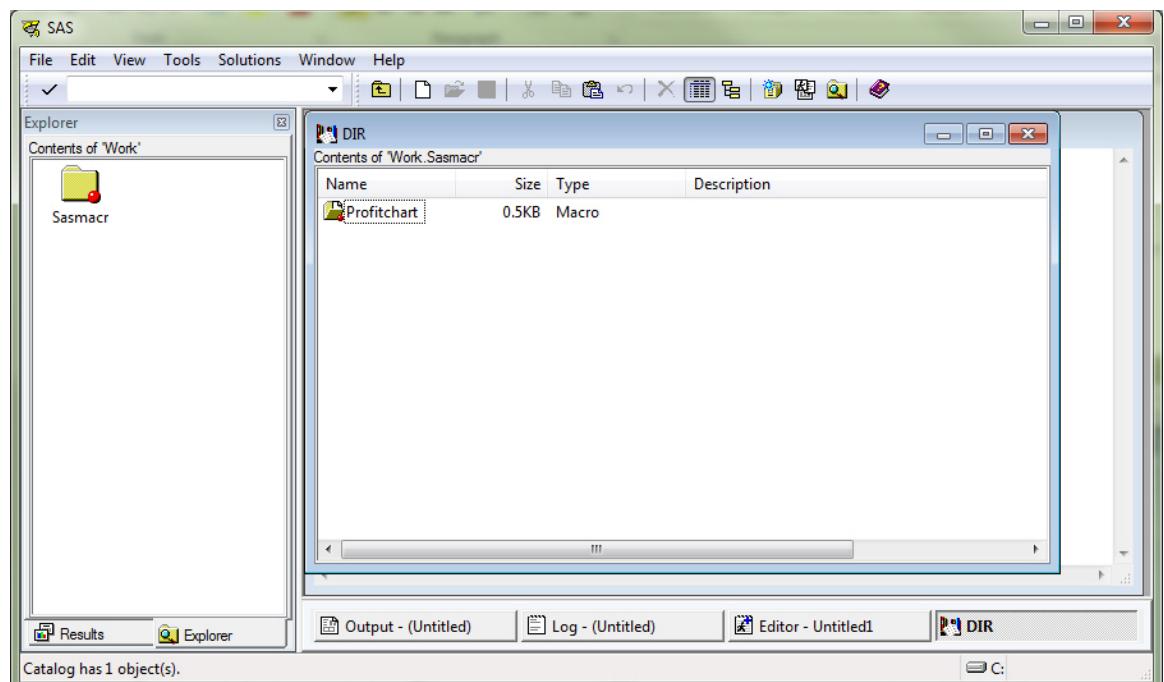
When the macro processor compiles the macro language statements in the macro program, it saves the results in a SAS catalog. By default, SAS stores macro programs in a catalog in the WORK library called SASMACR. Macro programs can also be saved in permanent catalogs and structures called autocall libraries. 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 WORK.SASMACR catalog throughout the SAS session. When the SAS session ends, SAS deletes the WORK.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 Figure 4.1. One catalog, the WORK.SASMACR catalog, exists.

**Figure 4.1 Viewing the WORK library**

Opening the WORK.SASMACR catalog presents the window shown in Figure 4.2. Note that the PROFITCHART macro program is in this catalog. The entry in the catalog for PROFITCHART is the compiled version of PROFITCHART. 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.)

**Figure 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 Example 4.2.

**Example 4.2**

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

If you want to delete a macro program from the WORK.SASMACR catalog, you can issue the %SYSMACDELETE statement.

```
%sysmacdelete profitchart;
```

---

## Executing a Macro Program

A macro program executes when you submit 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 first 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 in WORK.SASMACR, and depending on SAS system option settings, SAS looks in other locations for compiled macro code before it determines that the macro program reference cannot be resolved. When the search is unsuccessful, SAS writes an error message to the SAS log. Chapter 10 describes ways to tell SAS to look in other locations for compiled macro program code.

A semicolon does not terminate the call to the macro program. The call to a macro program is not a SAS statement. A semicolon is not required to terminate the call to a macro program. Importantly, adding a terminating semicolon to a macro program call might cause errors in its execution.

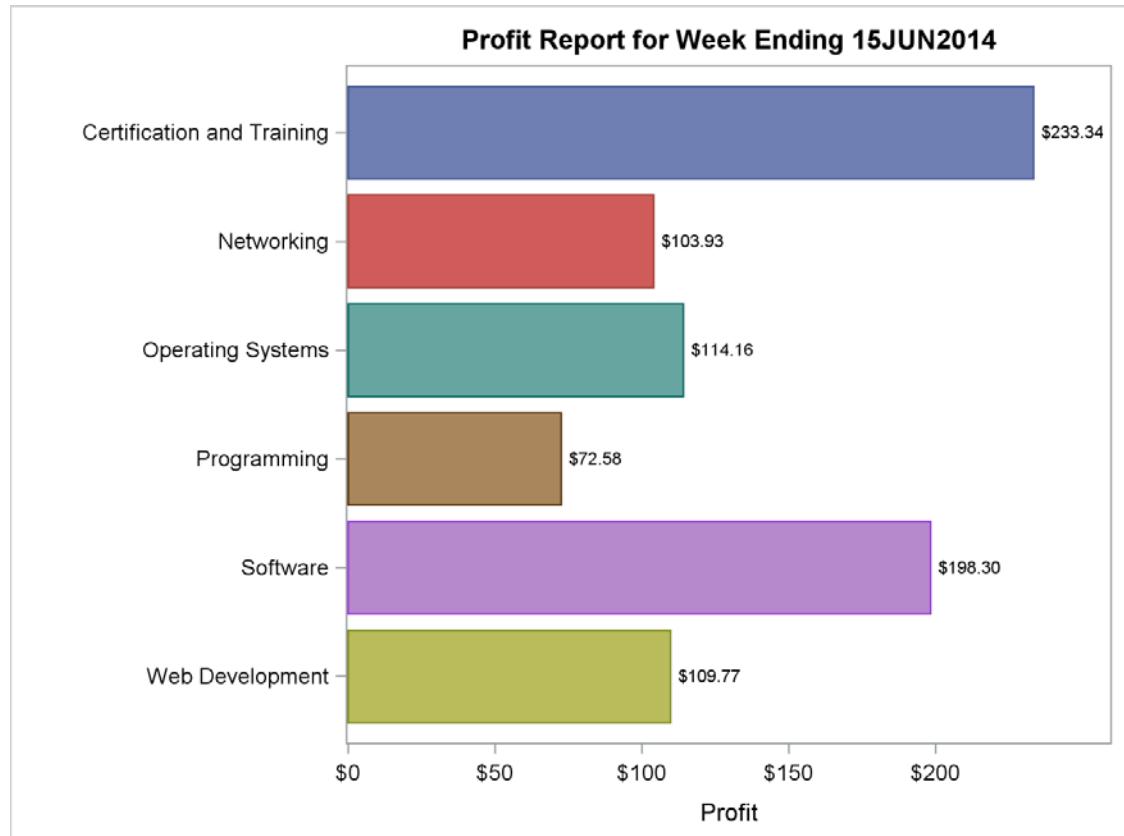
Example 4.3 calls the macro program defined in Example 4.1. Assume that the macro program in Example 4.1 was already submitted and its compiled code is in the SASMACR catalog. Assume the program was submitted on Friday, June 15, 2014. When Example 4.3 is submitted, the DATA step and the PROC SGLOT step in macro program PROFITCHART execute.

### Example 4.3

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

%profitchart
```

Output 4.1 presents the output from Example 4.3.

**Output 4.1 Output from Example 4.3**


---

## 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, and NONE is the default setting.
- The NOAUTOCALL value suppresses compilation notes for autocall macro programs and displays all other types of 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 ALL value causes display of all macro program compilation notes.

The default setting for MCOMPILENOTE= is NONE. Assume that was the setting when Example 4.1 executed. The following program adds MCOMPILENOTE=ALL to Example 4.1 and has no other changes. With this new setting for MCOMPILENOTE=, SAS writes compilation notes to the SAS log.

```
options mcompilenote=all;
%macro profitchart;
  ods graphics / maxlegendarea=0;

  title "Sales Report for Week Ending &sysdate9";
  proc sgplot data=temp;
    where today()-6 <=datesold <= today();
    hbar section / response=profit group=section stat=sum
      datalabel;
    yaxis label=' ';
  run;
%mend saleschart;
```

SAS generates the following note when this revised version of Example 4.1 is compiled.

```
NOTE: The macro PROFITCHART completed compilation without errors.
      7 instructions 340 bytes.
```

The following program contains a macro compilation error. A %IF statement has been added to Example 4.1, and the %IF-%THEN-%DO statement requires a matching %END statement. Instead of a %END statement, the program incorrectly includes an END statement.

```
options mcompilenote=all;
%macro profitchart;
  ods graphics / maxlegendarea=0;

  %if &sysday=FRIDAY %then %do;
    title "Sales Report for Week Ending &sysdate9";
    proc sgplot data=temp;
      where today()-6 <=datesold <= today();
      hbar section / response=profit group=section stat=sum
        datalabel;
      yaxis label=' ';
    run;
  end;
%mend profitchart;
```

The SAS log produced by this revised version of Example 4.1 follows. The ERROR message indicates that PROFITCHART did not compile. The last part of the NOTE indicates that no instructions were saved.

```
ERROR: There were 1 unclosed %DO statements.  The macro PROFITCHART
      will not be compiled.
```

```

ERROR: A dummy macro will be compiled.
NOTE: The macro PROFITCHART completed compilation with errors.
      0 instructions 0 bytes.

```

## Displaying Messages about Macro Program Processing in the SAS Log

SAS options MPRINT and MLOGIC write macro processing information to the SAS log. 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 SAS creates either in open code or inside macro programs.

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

```

411  options nosymbolgen nomprint nomlogic;
412  data temp;
413  set books.ytdsales;
414  attrib profit label='Profit' format=dollar8.2;
415  profit=saleprice-cost;
416  run;

NOTE: There were 3346 observations read from the data set
BOOKS.YTDSALES.
NOTE: The data set WORK.TEMP has 3346 observations and 11 variables.
NOTE: DATA statement used (Total process time):
      real time            0.03 seconds
      cpu time             0.01 seconds

417  %profitchart
NOTE: PROCEDURE SGLOT used (Total process time):
      real time            0.78 seconds
      cpu time             0.21 seconds

NOTE: Listing image output written to SGPlot.png.
NOTE: There were 60 observations read from the data set WORK.TEMP.
      WHERE (datesold>=(TODAY()-6)) and (datesold<=TODAY());
NOTE: PROCEDURE SGLOT used (Total process time):
      real time            0.03 seconds
      cpu time             0.00 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

does not write to the SAS log SAS language statements submitted from within a macro program. 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 Example 4.3 with the MPRINT option enabled. Now you can verify the SAS language statements that the macro program constructed.

```

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 3346 observations read from the data set
      BOOKS.YTDSALES.
NOTE: The data set WORK.TEMP has 3346 observations and 11
      variables.
NOTE: DATA statement used (Total process time):
      real time            0.01 seconds
      cpu time             0.00 seconds

531 %profitchart
MPRINT(PROPITCHART): ods graphics / maxlegendarea=0;
MPRINT(PROPITCHART): title "Sales Report for Week Ending 15JUN2014";
MPRINT(PROPITCHART): proc sgplot data=temp;
MPRINT(PROPITCHART): where today()-6 <=datesold <= today();
MPRINT(PROPITCHART): hbar section / response=profit group=section
stat=sum datalabel;
MPRINT(PROPITCHART): yaxis label=' ';
MPRINT(PROPITCHART): run;

NOTE: PROCEDURE SGPLOT used (Total process time):
      real time            1.26 seconds
      cpu time             0.23 seconds

NOTE: Listing image output written to SGPlot.png.
NOTE: There were 60 observations read from the data set WORK.TEMP.
      WHERE (datesold>=(TODAY()-6)) and (datesold<=TODAY());

```

---

## 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 Example 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 3346 observations read from the data set
      BOOKS.YTDSALES.
NOTE: The data set WORK.TEMP has 3346 observations and 11
      variables.
NOTE: DATA statement used (Total process time):
      real time            0.01 seconds
      cpu time             0.00 seconds

579
580 %profitchart
MLOGIC(PROPFITCHART): Beginning execution.

NOTE: PROCEDURE SGPLOT used (Total process time):
      real time            1.24 seconds
      cpu time             0.25 seconds

NOTE: Listing image output written to SGPlot.png.
NOTE: There were 60 observations read from the data set WORK.TEMP.
      WHERE (datesold>=(TODAY()-6)) and (datesold<=TODAY()));

MLOGIC(PROPFITCHART): 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 that are used 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.

You can specify the initial values of the parameters when you call the macro program. When the macro program starts, the macro processor initializes the corresponding macro variables with the values that were assigned to the parameters. Additionally, when you define a macro program, you can write the %MACRO statement so that it has initial values. Later, when you call such a macro program, you do not have to repeat the assignment of these default starting values.

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 general format of a macro program definition containing positional parameters follows.

```
%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 4.4: Defining a Macro Program with Positional Parameters

Example 4.4 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 can be used to select specific and other options that are valid on the PROC MEANS statement. The parameter values specified for START and STOP define the reporting time period.

```
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,01JUN2014,15JUN2014)
*****Second call to LISTPARM, first parameter is null,;
*****second and third parameters specified;
%listparm(,01SEP2014,15SEP2014)

```

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. For example, the PROC MEANS option MAXDEC= would qualify as a parameter value with a special character because of the presence of the equal sign (=).

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.

```

652 *****First call to LISTPARM, all 3 parameters specified;
653 %listparm(n sum,01JUN2014,15JUN2014)
MLOGIC(LISTPARM): Beginning execution.
MLOGIC(LISTPARM): Parameter OPTS has value n sum
MLOGIC(LISTPARM): Parameter START has value 01JUN2014
MLOGIC(LISTPARM): Parameter STOP has value 15JUN2014
MPRINT(LISTPARM):   title "Books Sold by Section Between 01JUN2014 and
15JUN2014";
MPRINT(LISTPARM):   proc means data=books.ytdsales n sum;
MPRINT(LISTPARM):   where "01JUN2014" d le datesold le "15JUN2014" d;
MPRINT(LISTPARM):   class section;
MPRINT(LISTPARM):   var saleprice;
MPRINT(LISTPARM):   run;

NOTE: There were 136 observations read from the data set
BOOKS.YTDSALES.
WHERE (datesold>='01JUN2014'D and datesold<='15JUN2014'D);

```

```
NOTE: PROCEDURE MEANS used (Total process time):
      real time            0.03 seconds
      cpu time             0.03 seconds
```

MLOGIC(LISTPARM): Ending execution.

The SAS log for the second call to LISTPARM follows.

```
654  -----Second call to LISTPARM, first parameter is null,;
655  -----second and third parameters specified;
656 %listparm(,01SEP2014,15SEP2014)
MLOGIC(LISTPARM): Beginning execution.
MLOGIC(LISTPARM): Parameter OPTS has value
MLOGIC(LISTPARM): Parameter START has value 01SEP2014
MLOGIC(LISTPARM): Parameter STOP has value 15SEP2014
MPRINT(LISTPARM): title "Books Sold by Section Between 01SEP2014 and
15SEP2014";
MPRINT(LISTPARM): proc means data=books.ytdsales ;
MPRINT(LISTPARM): where "01SEP2014"d le datesold le "15SEP2014"d;
MPRINT(LISTPARM): class section;
MPRINT(LISTPARM): var saleprice;
MPRINT(LISTPARM): run;

NOTE: There were 135 observations read from the data set
BOOKS.YTDSALES.
      WHERE (datesold>='01SEP2014'D and datesold<='15SEP2014'D);
NOTE: PROCEDURE MEANS used (Total process time):
      real time            0.03 seconds
      cpu time             0.00 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 that defines the macro program, an equal sign (=) follows each parameter name.

A macro program with many parameters is easier to define and use when it has 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 the default value, you can omit the keyword parameter completely from the macro program call.

You can specify keyword parameters 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 that is

defined with keyword parameters is not required as it is with macro programs defined with positional parameters. The general format of a macro program definition that defines 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>;
```

Enclose keyword parameter lists with parentheses, and separate them 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 use that default value when you call the macro program, you do not have to specify the corresponding keyword parameter. SAS does not have a limit on the number of keyword parameters that you can define.

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 4.5: Defining a Macro Program with Keyword Parameters**

Example 4.5 defines the macro program KEYPARM and calls it three times. Example 4.5 does the same task as Example 4.4. The difference is that the macro program in Example 4.5 is defined with keyword parameters while macro program LISTPARM in Example 4.4 is defined with positional parameters.

Macro program KEYPARM defines three keyword parameters: OPTS, START, and STOP. All three keyword parameters are assigned default values on the %MACRO KEYPARM statement.

The three calls to KEYPARM in Example 4.5 have these features:

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

```

options mprint mlogic;

%macro keyparm(opts=N SUM MIN MAX,
               start=01JAN2014,stop=31DEC2014);
  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=01JUN2014,stop=15JUN2014)

-----Second call to KEYPARM: specify start and stop,;
-----opts is null: should see default stats for PROC MEANS;
%keyparm(opts=,start=01SEP2014,stop=15SEP2014)

-----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.

```

686 -----First call to KEYPARM: specify all keyword parameters;
687 %keyparm(opts=n sum,start=01JUN2014,stop=15JUN2014)
MLOGIC (KEYPARM): Beginning execution.
MLOGIC (KEYPARM): Parameter OPTS has value n sum
MLOGIC (KEYPARM): Parameter START has value 01JUN2014
MLOGIC (KEYPARM): Parameter STOP has value 15JUN2014
MPRINT (KEYPARM):   title "Books Sold by Section Between 01JUN2014 and
15JUN2014";
MPRINT (KEYPARM):   proc means data=books.ytdsales n sum;
MPRINT (KEYPARM):   where "01JUN2014"d le datesold le "15JUN2014"d;
MPRINT (KEYPARM):   class section;
MPRINT (KEYPARM):   var saleprice;
MPRINT (KEYPARM):   run;

NOTE: There were 136 observations read from the data set
BOOKS.YTDSALES.
      WHERE (datesold>='01JUN2014'D and datesold<='15JUN2014'D);
NOTE: PROCEDURE MEANS used (Total process time):
      real time          0.02 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.

```

688
689 *----Second call to KEYPARM: specify start and stop,;
690 *----opts is null: should see default stats for PROC MEANS;
691 %keyparm(opts=,start=01SEP2014,stop=15SEP2014)
MLOGIC(KEYPARM): Beginning execution.
MLOGIC(KEYPARM): Parameter OPTS has value
MLOGIC(KEYPARM): Parameter START has value 01SEP2014
MLOGIC(KEYPARM): Parameter STOP has value 15SEP2014
MPRINT(KEYPARM): title "Books Sold by Section Between 01SEP2014 and
15SEP2014";
MPRINT(KEYPARM): proc means data=books.ytdsales ;
MPRINT(KEYPARM): where "01SEP2014"d le datesold le "15SEP2014"d;
MPRINT(KEYPARM): class section;
MPRINT(KEYPARM): var saleprice;
MPRINT(KEYPARM): run;

NOTE: There were 135 observations read from the data set
BOOKS.YTDSALES.
      WHERE (datesold>='01SEP2014'D and datesold<='15SEP2014'D);
NOTE: PROCEDURE MEANS used (Total process time):
      real time            0.03 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, 2014, and December 31, 2014. The SAS log for the third call to KEYPARM follows.

```

693 *----Third call to KEYPARM: use defaults for start and stop,; *--
--specify opts;
694 %keyparm(opts=n sum)
MLOGIC(KEYPARM): Beginning execution.
MLOGIC(KEYPARM): Parameter OPTS has value n sum
MLOGIC(KEYPARM): Parameter START has value 01JAN2014
MLOGIC(KEYPARM): Parameter STOP has value 31DEC2014
MPRINT(KEYPARM): title "Books Sold by Section Between 01JAN2014 and
31DEC2014";
MPRINT(KEYPARM): proc means data=books.ytdsales n sum;
MPRINT(KEYPARM): where "01JAN2014"d le datesold le "31DEC2014"d;
MPRINT(KEYPARM): class section;

```

```

MPRINT (KEYPARAM) :   var saleprice;
MPRINT (KEYPARAM) :   run;

NOTE: There were 3346 observations read from the data set
BOOKS.YTDSALES.
      WHERE (datesold>='01JAN2014'D and datesold<='31DEC2014'D);
NOTE: PROCEDURE MEANS used (Total process time):
      real time            0.03 seconds
      cpu time             0.01 seconds

MLOGIC (KEYPARAM) :  Ending execution.

```

---

## Specifying Mixed Parameter Lists in Macro Programs

You can define both positional parameters and keyword parameters 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 4.6: Defining a Macro Program with Positional Parameters and Keyword Parameters

Example 4.6 defines the macro program, MIXDPARM, with two positional parameters and two keyword parameters, and it calls that macro program twice. Example 4.6 completes the same task as Examples 4.4 and 4.5.

Macro program MIXDPARM includes 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. Both START and STOP are assigned default values.

The first call to MIXDPARM has these characteristics:

- The null value assigned to the STATS positional parameter causes PROC MEANS to calculate default statistics.
- The value MISSING assigned to parameter OTHROPTS requests the MISSING option on the PROC MEANS statement. The MISSING option specifies 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, 2014. It does not assign a value to the STOP keyword parameter. Without a value specified for STOP, the default value of December 31, 2014, assigned in the macro program definition, is used as the stop date. The first call to MIXDPARM analyzes information for December 2014.

The second call to MIXDPARM has these characteristics:

- No values for either the positional or the keyword parameters are specified. With parameters STAT and OTHROPTS both null, PROC MEANS calculates SAS default statistics and settings. Since no values for START and STOP are specified, the PROC MEANS step calculates statistics for the range of dates defined by the default settings assigned in the macro program definition. This means that PROC MEANS analyzes data for all of 2014.
- Note that a comma does not separate the null values for positional parameters STATS and OTHROPTS.

```
options mprint mlogic;

%macro mixdparm(stats,othropts,
                 start=01JAN2014,stop=31DEC2014);
  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 2014 and allow
   missing values to be valid in creating combinations of
   the CLASS variables;
%mixdparm(,missing,start=01DEC2014)

/*----Compute default stats for all of 2014;
%mixdparm()
```

The SAS log for the first call to MIXDPARM follows.

```
14 %*----Compute default stats for December 2014 and allow
15      missing values to be valid in creating combinations of
16      the CLASS variables;
17  %mixdparm(,missing,start=01DEC2014)
MLOGIC (MIXDPARM) : Beginning execution.
MLOGIC (MIXDPARM) : Parameter STATS has value
MLOGIC (MIXDPARM) : Parameter OTHROPTS has value missing
MLOGIC (MIXDPARM) : Parameter START has value 01DEC2014
MLOGIC (MIXDPARM) : Parameter STOP has value 31DEC2014
MPRINT (MIXDPARM) : title "Books Sold by Section Between 01DEC2014 and
31DEC2014"; 
MPRINT (MIXDPARM) : proc means data=books.ytdsales missing;
MPRINT (MIXDPARM) : where "01DEC2014"d le datesold le "31DEC2014"d;
MPRINT (MIXDPARM) : class section;
MPRINT (MIXDPARM) : var saleprice;
MPRINT (MIXDPARM) : run;
NOTE: There were 287 observations read from the data set
BOOKS.YTDSALES.
      WHERE (datesold>='01DEC2014'D and datesold<='31DEC2014'D);
NOTE: PROCEDURE MEANS used (Total process time):
      real time          0.30 seconds
      cpu time           0.04 seconds
MLOGIC (MIXDPARM) : Ending execution.
.
```

The SAS log for the second call to MIXDPARM follows.

```

19  %*----Compute default stats for all of 2014;
20  %mixdparm()
MLOGIC (MIXDPARM) : Beginning execution.
MLOGIC (MIXDPARM) : Parameter STATS has value
MLOGIC (MIXDPARM) : Parameter OTHROPTS has value
MLOGIC (MIXDPARM) : Parameter START has value 01JAN2014
MLOGIC (MIXDPARM) : Parameter STOP has value 31DEC2014
MPRINT (MIXDPARM) : title "Books Sold by Section Between 01JAN2014 and
31DEC2014";
MPRINT (MIXDPARM) : proc means data=books.ytdsales ;
MPRINT (MIXDPARM) : where "01JAN2014"d le datesold le "31DEC2014"d;
MPRINT (MIXDPARM) : class section;
MPRINT (MIXDPARM) : var saleprice;
MPRINT (MIXDPARM) : run;

NOTE: There were 3346 observations read from the data set
BOOKS.YTDSALES.
      WHERE (datesold>='01JAN2014'D and datesold<='31DEC2014'D);
NOTE: PROCEDURE MEANS used (Total process time):
      real time          0.03 seconds
      cpu time          0.00 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 and all you have to do is simply change a parameter value each time, you might find it useful to add the PARMBUFF option to the definition of your macro program. With PARMBUFF, you can submit your macro program only once and specify the complete list of values that you want processed on that single call to your macro program. Additionally, when you use PARMBUFF, each time you execute your macro program you can specify a different number of parameter values.

The PARMBUFF option assigns the entire list of parameter values in the call to your macro program to the automatic macro variable SYSPBUFF. Your macro program can then parse the list of values and direct 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(<parameter-list>) / 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 add positional and keyword parameters to the macro program as well when you use the PARMBUFF option. These defined parameters receive values, and SAS also assigns to SYSPBUFF the values from these defined parameters. Because you could be dealing with a varying number of parameter values, you would typically place your defined parameters at the beginning of the macro program call and place the list of varying values at the end. When the code in your macro program parses the values in SYSPBUFF, it will 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(parmvalue-1, parmvalue-2, ...,
         parmvalue-n)
```

### **Example 4.7: Defining a Macro Program with the PARMBUFF Option**

Example 4.7 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 SGPlot step that displays monthly sales by section. When no parameter values are specified, PBUFFPARMS submits a summary PROC SGPlot 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 Example 4.7.

Example 4.7 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 SGPlot 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 SAS includes the parentheses in the value it assigns to SYSPBUFF.

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

```
%macro pbuffparms / parmbuff;
  ods graphics on / reset=all;

  %*----Process this section when parameter values specified;
  %if &sysbuff ne %then %do;
    %let i=1;
    %let month=%scan(&sysbuff,&i);
    %do %while(&month ne);
      proc sgplot data=books.ytdsales
        (where=(month(datesold)=&month));
        title "Section Sales Report for Month &month";
        hbar section / response=saleprice stat=sum
          datalabel;
        yaxis label=' ';
      run;
      %let i=%eval(&i+1);
      %let month=%scan(&sysbuff,&i);
    %end;
  %end;

  %*----Process this section when no parameter values
  specified;
  %else %do;
    proc sgplot data=books.ytdsales;
      title "Annual Section Sales by Quarter";
      hbar section / response=saleprice stat=sum
        group=datesold;
      yaxis label=' ';
      label datesold='Quarter';
      format datesold qtr.;
    run;
  %end;

%mend pbuffparms;

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

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

The SAS log for Example 4.7 follows. The MPRINT option is enabled. This shows that three PROC SGPLOT steps executed, two from the first call to PBUFFPARMS and one from the second call to PBUFFPARMS.

```

196  -----Analyze sales for August and November;
197  %pbuffparms(8,11)
MPRINT(PBUFFPARMS): ods graphics on / reset=all;
MPRINT(PBUFFPARMS): proc sgplot data=books.ytdsales
(where=(month(datesold)=8));
MPRINT(PBUFFPARMS): title "Section Sales Report for Month 8";
MPRINT(PBUFFPARMS): hbar section / response=saleprice stat=sum
datalabel;
MPRINT(PBUFFPARMS): yaxis label=' ';
MPRINT(PBUFFPARMS): run;

NOTE: PROCEDURE SGPLOT used (Total process time):
      real time            0.40 seconds
      cpu time             0.09 seconds

NOTE: Listing image output written to SGPlot.png.
NOTE: There were 299 observations read from the data set
BOOKS.YTDSALES.
      WHERE MONTH(datesold)=8;

MPRINT(PBUFFPARMS): proc sgplot data=books.ytdsales
(where=(month(datesold)=11));
MPRINT(PBUFFPARMS): title "Section Sales Report for Month 11";
MPRINT(PBUFFPARMS): hbar section / response=saleprice stat=sum
datalabel;
MPRINT(PBUFFPARMS): yaxis label=' ';
MPRINT(PBUFFPARMS): run;

NOTE: PROCEDURE SGPLOT used (Total process time):
      real time            0.39 seconds
      cpu time             0.09 seconds

NOTE: Listing image output written to SGPlot1.png.
NOTE: There were 255 observations read from the data set
BOOKS.YTDSALES.
      WHERE MONTH(datesold)=11;

198
199  -----Analyze sales for entire year;
200  %pbuffparms()
MPRINT(PBUFFPARMS): ods graphics on / reset=all;
MPRINT(PBUFFPARMS): proc sgplot data=books.ytdsales;
MPRINT(PBUFFPARMS): title "Annual Section Sales by Quarter";
MPRINT(PBUFFPARMS): hbar section / response=saleprice stat=sum
group=datesold;

```

```
MPRINT(PBUFFPARMS):  yaxis label=' ';
MPRINT(PBUFFPARMS):  label datesold= 'Quarter';
MPRINT(PBUFFPARMS):  format datesold qtr.;
MPRINT(PBUFFPARMS):  run;

NOTE: PROCEDURE SGPLOT used (Total process time):
      real time            0.38 seconds
      cpu time             0.03 seconds

NOTE: Listing image output written to SGPlot.png.
NOTE: There were 3346 observations read from the data set
BOOKS.YTDSALES.
```



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

<b>Introduction .....</b>	<b>95</b>
<b>Understanding Macro Symbol Tables .....</b>	<b>95</b>
Understanding the Global Macro Symbol Table .....	97
Understanding Local Macro Symbol Tables.....	101
Working with Global Macro Variables and Local Macro Variables .....	108
Defining the Domain of a Macro Variable by Using the %GLOBAL or %LOCAL Macro Language Statements .....	110
<b>Processing of Macro Programs .....</b>	<b>114</b>
How a Macro Program Is Compiled.....	114
How a Macro Program Executes .....	120

---

## **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.

This chapter continues the discussion of the technical aspects of macro processing that began in Chapter 2. This chapter describes symbol tables, both global and those created locally 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. SAS can resolve references to local macro variables from within the macro program that defined the macro variables. A macro program can also call other macro programs. Local macro variables defined by a macro program that calls a second macro program are available to this second macro program. They are also available to macro programs that this second macro program calls and so on.

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.

The macro processor creates a local macro symbol table when SAS executes a macro program that contains macro variables. SAS stores a macro variable in this local macro symbol table only if the macro variable does not exist already either in the global macro symbol table, or in the case of nested macro program calls, in previously defined local macro symbol tables higher in scope than your macro program. A local macro symbol table and all of its macro variables are deleted when the macro program that it is associated with ends.

As your SAS programming assistant, the macro processor keeps track of the domain of each macro variable that your SAS program defines. 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.

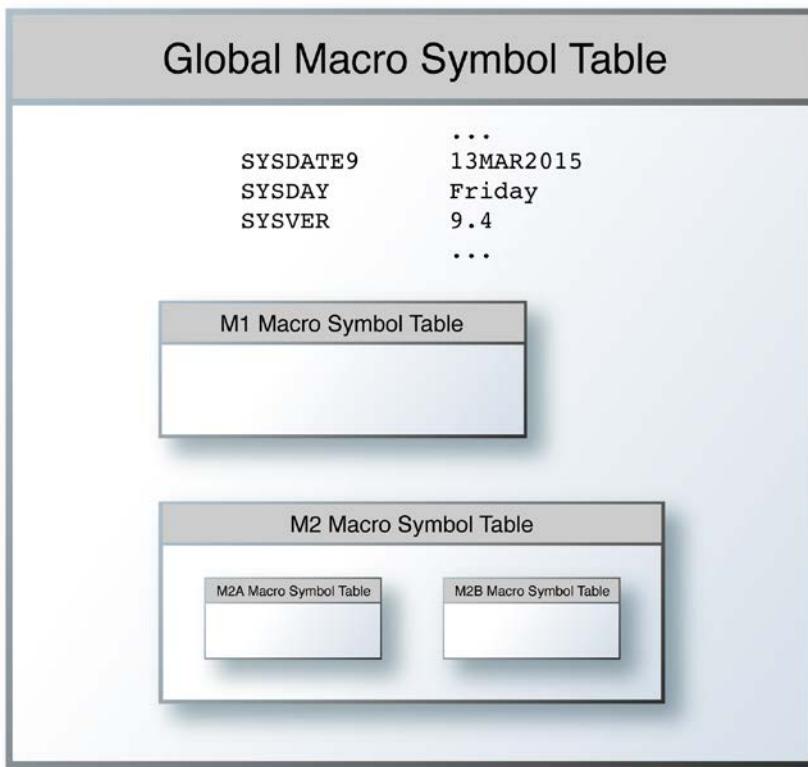
Figure 5.1 presents an example of how the domains of the macro symbol tables are defined by default. Two macro programs, M1 and M2, are invoked. In addition, two more macro programs, M2A and M2B, are invoked from within macro program M2 when macro program M2 is executing.

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 also 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, SAS stores the automatic macro variables SYSDATE9, SYSDAY, and SYSVER, as shown in Figure 5.1, 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. Four ways that you can add macro variables to the global macro symbol table are:

- 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 CALL SYMPUTX when the global symbol table is specified. (CALL SYMPUTX provides an interface between the SAS DATA step and the macro processor, and Chapter 9 describes how to use it.)
- Define a global macro variable with the INTO clause on the PROC SQL SELECT statement. Chapter 9 describes the INTO clause.

You can reference global macro variables throughout the SAS session in which they are created. They can be referenced from open code and from inside macro programs. By default, 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. Starting with SAS 9.4, you can also make user-defined global macro variables read-only.

### **Example 5.1: Referencing a Global Macro Variable in Open Code and from within a Macro Program**

Example 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 Example 5.1 was submitted on March 13, 2015.

```
options symbolgen mprint;

%let subset=Software;

%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;
```

Example 5.1 generates the following SAS log.

```

236 options symbolgen mprint;
237
238 %let subset=Software;
239
240 %macro makeds;
241   data temp;
242     set books.ytdsales(where=(section="&subset"));
243     attrib qtrsold label='Quarter of Sale';
244     qtrsold=qtr(datesold);
245   run;
246 %mend makeds;
247
248 %makeds
MPRINT(MAKEDS):  data temp;
SYMBOLGEN:  Macro variable SUBSET resolves to Software
MPRINT(MAKEDS):  set books.ytdsales(where=(section="Software"));
MPRINT(MAKEDS):  attrib qtrsold label='Quarter of Sale';
MPRINT(MAKEDS):  qtrsold=qtr(datesold);
MPRINT(MAKEDS):  run;

NOTE: There were 857 observations read from the data set
BOOKS.YTDSALES.
      WHERE section='Software';
NOTE: The data set WORK.TEMP has 857 observations and 11 variables.
NOTE: DATA statement used (Total process time):
      real time            0.04 seconds
      cpu time             0.00 seconds

249
250 proc tabulate data=temp;
SYMBOLGEN:  Macro variable SYSDATE9 resolves to 13MAR2015
251   title "Book Sales Report Produced &sysdate9";
252   class qtrsold;
253   var saleprice listprice;
254   tables qtrsold all,
255         (saleprice listprice)*(n*f=6. sum*f=dollar12.2) /
256         box="Section: &subset";
SYMBOLGEN:  Macro variable SUBSET resolves to Software
257   keylabel all='** Total **';
258   run;

NOTE: There were 857 observations read from the data set WORK.TEMP.
NOTE: PROCEDURE TABULATE used (Total process time):
      real time            0.30 seconds
      cpu time             0.03 seconds

```

Output 5.1 presents the output for Example 5.1.

#### Output 5.1 Output for Example 5.1

### Book Sales Report Produced 13MAR2015

Section: Software	Sale Price		List Price	
	N	Sum	N	Sum
Quarter of Sale				
1	220	\$7,590.38	220	\$8,929.64
2	195	\$6,713.06	195	\$7,897.53
3	214	\$7,289.90	214	\$8,576.18
4	228	\$7,781.64	228	\$9,154.68
** Total **	857	\$29,374.98	857	\$34,558.03

Figure 5.2 shows a representation of the global macro symbol table that results when macro program MAKEDS in Example 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 Example 5.1 executes

Global Macro Symbol Table	
SYSDATE9	13MAR2015
SYSDAY	Friday
SYSVER	9.4
	...
SUBSET	Software

### Understanding Local Macro Symbol Tables

If your macro program creates macro variables and does not specify them as global macro variables, the macro processor usually creates a local macro symbol table when 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 macro symbol tables. By default, the domain of macro variables that a macro program creates 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 you should probably 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.

### **Example 5.2: Referencing a Macro Variable Defined as a Parameter to a Macro Program – Version 1**

Example 5.1 in the previous section is revised below in Example 5.2 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.)

```
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(Software)

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.

```
55  %makeds(Software)
MPRINT(MAKEDS):   data temp;
SYMBOLGEN:  Macro variable SUBSET resolves to Software
```

```

MPRINT(MAKEDS): set books.ytdsales(where=(section="Software"));
MPRINT(MAKEDS): attrib qtrsold label='Quarter of Sale';
MPRINT(MAKEDS): qtrsold=qtr(datesold);
MPRINT(MAKEDS): run;

NOTE: There were 857 observations read from the data set
      BOOKS.YTDSALES.
      WHERE section='Software';
NOTE: The data set WORK.TEMP has 857 observations and 11 variables.
NOTE: Compressing data set WORK.TEMP decreased size by 41.18 percent.
      Compressed is 10 pages; un-compressed would require 17 pages.
NOTE: DATA statement used (Total process time):
      real time           0.06 seconds
      cpu time            0.01 seconds

56
57 proc tabulate data=temp;
SYMBOLGEN: Macro variable SYSDATE9 resolves to 13MAR2015
58   title "Book Sales Report Produced &sysdate9";
59   class qtrsold;
60   var saleprice listprice;
61   tables qtrsold all,
62     (saleprice listprice)*(n*f=6. sum*f=dollar12.2) /
63     box="Section: &subset";
WARNING: Apparent symbolic reference SUBSET not resolved.
64   keylabel all='** Total **';
65   run;

NOTE: There were 857 observations read from the data set WORK.TEMP.
NOTE: PROCEDURE TABULATE used (Total process time):
      real time           0.14 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 Example 5.2. This output shows that the macro processor did not resolve the macro variable reference in the box text area of the report.

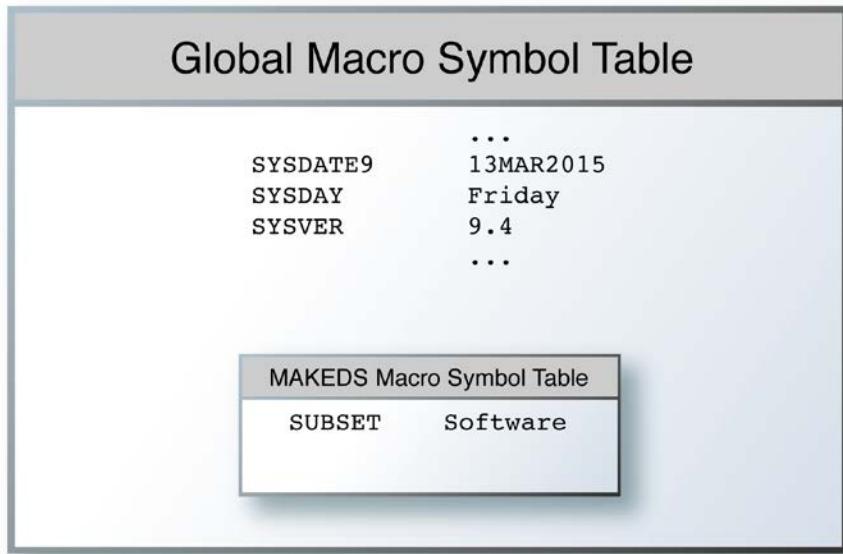
#### Output 5.2 Output for Example 5.2

## Book Sales Report Produced 13MAR2015

Section: &subset	Sale Price		List Price	
	N	Sum	N	Sum
<b>Quarter of Sale</b>				
<b>1</b>	220	\$7,590.38	220	\$8,929.64
<b>2</b>	195	\$6,713.06	195	\$7,897.53
<b>3</b>	214	\$7,289.90	214	\$8,576.18
<b>4</b>	228	\$7,781.64	228	\$9,154.68
<b>** Total **</b>	857	\$29,374.98	857	\$34,558.03

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 Example 5.2 when SUBSET is a local macro variable**



Examples 5.3 and 5.4 show two different ways to modify Example 5.2 so that the reference to macro variable SUBSET in the PROC TABULATE step can be resolved. Example 5.3 is the easier of the two. Example 5.4 further illustrates the concepts of macro symbol tables.

### **Example 5.3: Referencing a Macro Variable Defined as a Parameter to a Macro Program – Version 2**

Example 5.3 places the PROC TABULATE step within the MAKEDS macro program. When Example 5.3 executes, the reference to macro variable SUBSET in the box text of the report is resolved because SUBSET is local to macro program MAKEDS.

```

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(Software)

```

#### **Example 5.4: Referencing a Macro Variable Defined as a Parameter to a Macro Program – Version 3**

In Example 5.4, the %GLOBAL statement instructs the macro processor to create global macro variable, GLBSUBSET. 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 local macro variable SUBSET can be transferred to global macro variable GLBSUBSET. The macro variable reference on the BOX= option in PROC TABULATE is changed to the name of the global macro variable, GLBSUBSET.

```

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(Software)

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 third version follows.

```

89  %makeds(Software)
SYMBOLGEN: Macro variable SUBSET resolves to Software
MPRINT(MAKEDS): data temp;
SYMBOLGEN: Macro variable SUBSET resolves to Software
MPRINT(MAKEDS): set books.ytdsales(where=(section="Software"));
MPRINT(MAKEDS): attrib qtrsold label='Quarter of Sale';
MPRINT(MAKEDS): qtrsold=qtr(datesold);
MPRINT(MAKEDS): run;

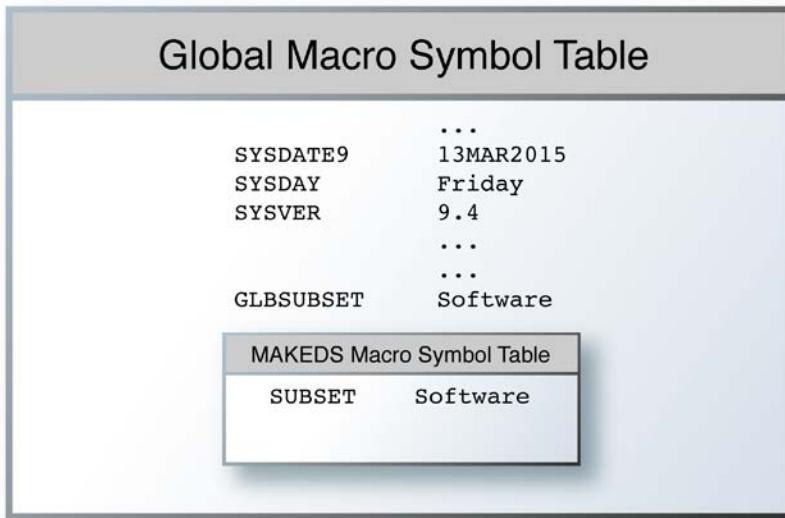
NOTE: There were 857 observations read from the data set
      BOOKS.YTDSALES.
      WHERE section='Software';
NOTE: The data set WORK.TEMP has 857 observations and 11 variables.
NOTE: DATA statement used (Total process time):
      real time            0.03 seconds
      cpu time             0.00 seconds
90
91  proc tabulate data=temp;
SYMBOLGEN: Macro variable SYSDATE9 resolves to 13MAR2015
92      title "Book Sales Report Produced &sysdate9";
93      class qtrsold;
94      var saleprice listprice;
95      tables qtrsold all,
96          (saleprice listprice)*(n*f=6. sum*f=dollar12.2) /
97          box="Section: &glbsubset";
SYMBOLGEN: Macro variable GLBSUBSET resolves to Software
98      keylabel all='** Total **';
99      run;

NOTE: There were 857 observations read from the data set WORK.TEMP.
NOTE: PROCEDURE TABULATE used (Total process time):
      real time            0.12 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 Example 5.4 has executed**



## Working with Global Macro Variables and Local Macro Variables

The macro processor follows a specific set of rules when it resolves 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.

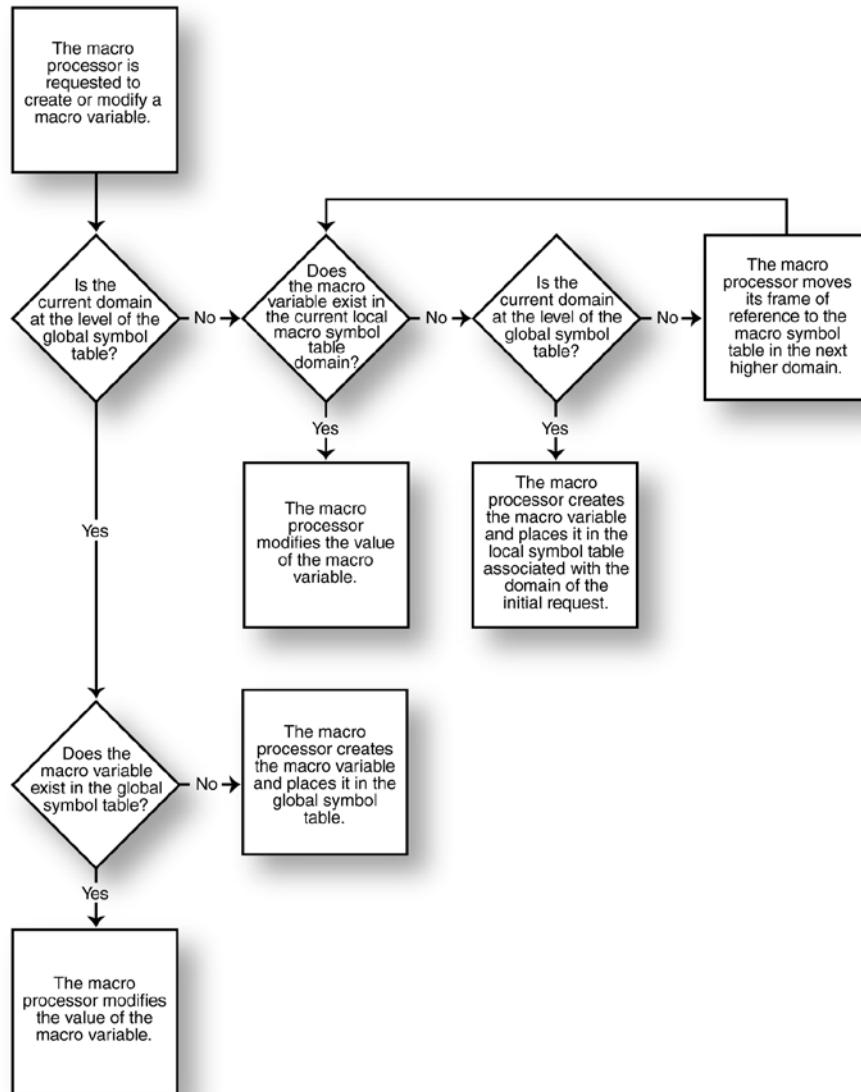
When your macro program references a global macro variable, the macro processor does not by default create a local macro variable with that name. Instead, the macro processor uses the global macro variable.

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, your code will be much easier to read and maintain if you do not nest macro program definitions.

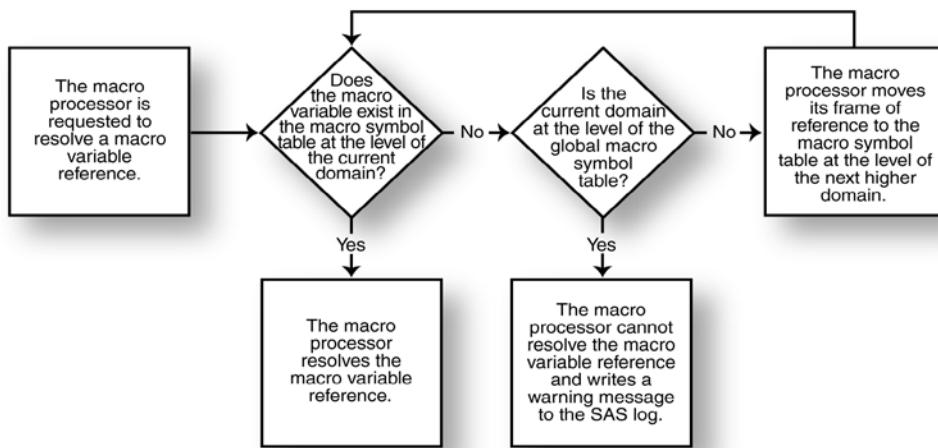
The macro processor follows the path in Figure 5.5 when it creates or modifies macro variables.

**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 %GLOBAL and %LOCAL statements each have two forms. The first form defines one or more macro variables that you can modify and delete within the SAS session, or within the macro program when working with a local macro variable.

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

The second form, which is new in SAS 9.4, defines a single macro variable as read-only and assigns it a value. The value cannot be changed in the current SAS session, and the macro variable cannot be deleted in the current SAS session. Defining your macro variables as read-only can ensure that your programs do not change the values of specific macro variables or delete them during the SAS session or execution of a macro program.

```
%GLOBAL READONLY / macro-variable=value ;
%LOCAL READONLY / macro-variable=value ;
```

The remainder of this section describes how to use macro variables that are defined the first way shown above. The values of these macro variables can be changed during the SAS session, and these macro variables can be deleted.

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. Example 5.4 includes a %GLOBAL statement.

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.

### **Example 5.5: Using the Same Name for a Macro Variable Defined Globally and Defined Locally**

Example 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 Software.

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 Development.

```
options symbolgen mprint;

%let subset=Software;

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

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 Example 5.5 follows.

```

118 %loclmvar
MPRINT(LOCLMVAR): proc means data=books.ytdsales n sum maxdec=2;
SYMBOLGEN: Macro variable SYSDATE9 resolves to 13MAR2015
MPRINT(LOCLMVAR): title "Book Sales Report Produced 13MAR2015";
SYMBOLGEN: Macro variable SUBSET resolves to Web Development
MPRINT(LOCLMVAR): title2 "Uses LOCAL SUBSET macro variable: Web
Development";
SYMBOLGEN: Macro variable SUBSET resolves to Web Development
MPRINT(LOCLMVAR): where section="Web Development";
MPRINT(LOCLMVAR): var saleprice;
MPRINT(LOCLMVAR): run;

NOTE: There were 571 observations read from the data set
      BOOKS.YTDSALES.
      WHERE section='Web Development';
NOTE: PROCEDURE MEANS used (Total process time):
      real time           0.07 seconds
      cpu time            0.01 seconds
119
120 proc means data=books.ytdsales n sum maxdec=2;
SYMBOLGEN: Macro variable SYSDATE9 resolves to 13MAR2015
121 title "Book Sales Report Produced &sysdate9";
SYMBOLGEN: Macro variable SUBSET resolves to Software
122 title2 "Uses GLOBAL SUBSET macro variable: &subset";
123 where section="&subset";
SYMBOLGEN: Macro variable SUBSET resolves to Software
124 var saleprice;
125 run;

NOTE: There were 857 observations read from the data set
      BOOKS.YTDSALES.
      WHERE section='Software';
NOTE: PROCEDURE MEANS used (Total process time):
      real time           0.03 seconds
      cpu time            0.01 seconds

```

Output 5.3 presents the output from Example 5.5.

**Output 5.3 Output from Example 5.5**

**Book Sales Report Produced 13MAR2015**  
**Uses LOCAL SUBSET macro variable: Web Development**

**The MEANS Procedure**

Analysis Variable : saleprice Sale Price	
N	Sum
571	12065.77

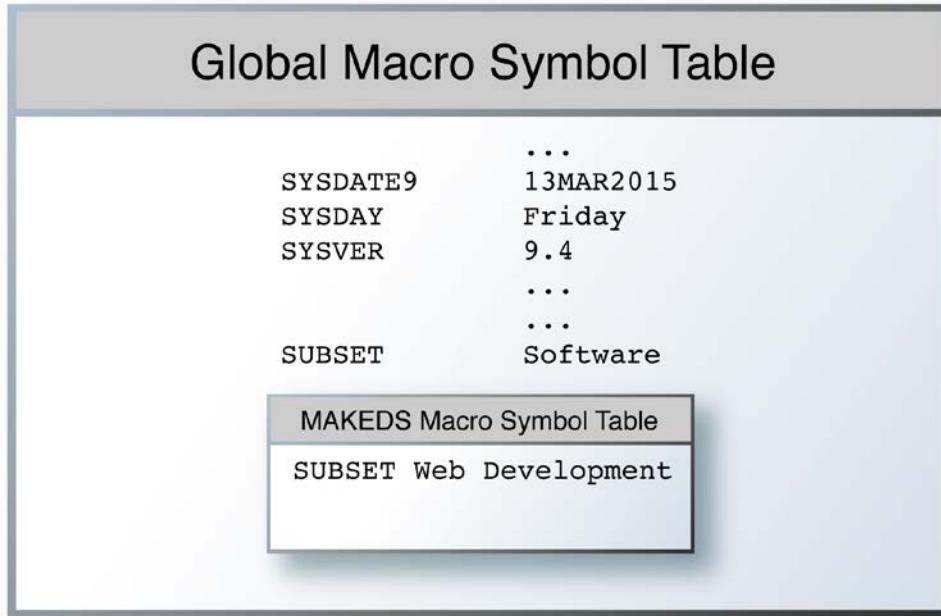
---

**Book Sales Report Produced 13MAR2015**  
**Uses GLOBAL SUBSET macro variable: Software**

**The MEANS Procedure**

Analysis Variable : saleprice Sale Price	
N	Sum
857	29374.98

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

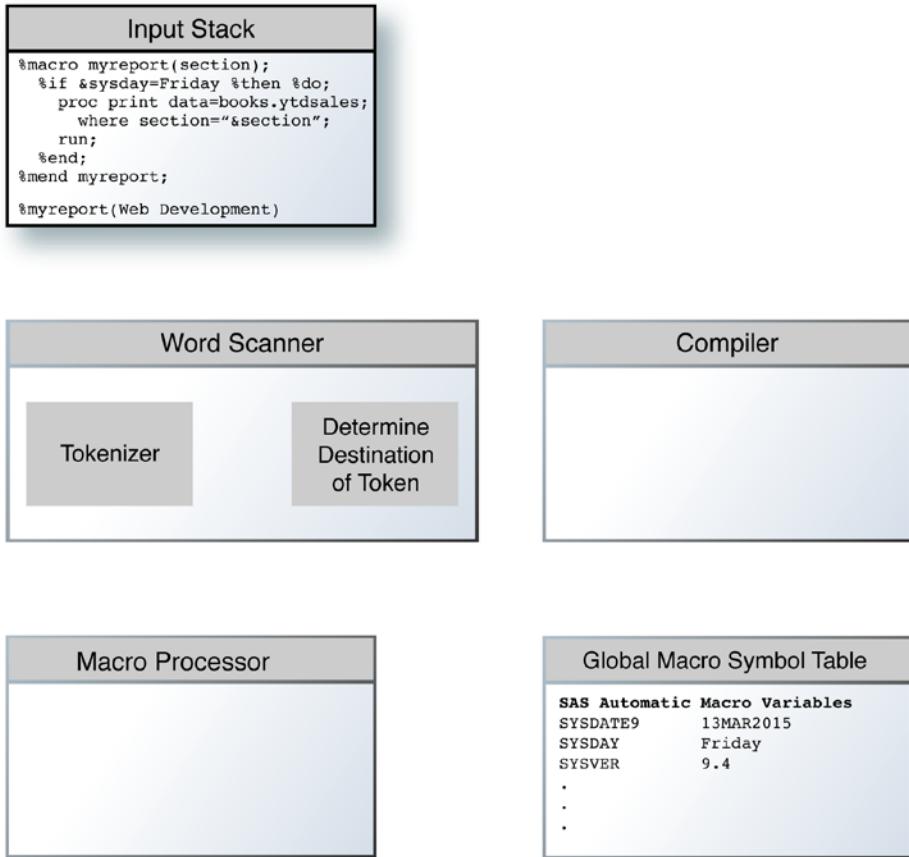
Understanding how the macro processor compiles and executes macro programs will help you more quickly write and debug SAS programs that contain macro language. This section describes how SAS and the macro processor process macro programs.

---

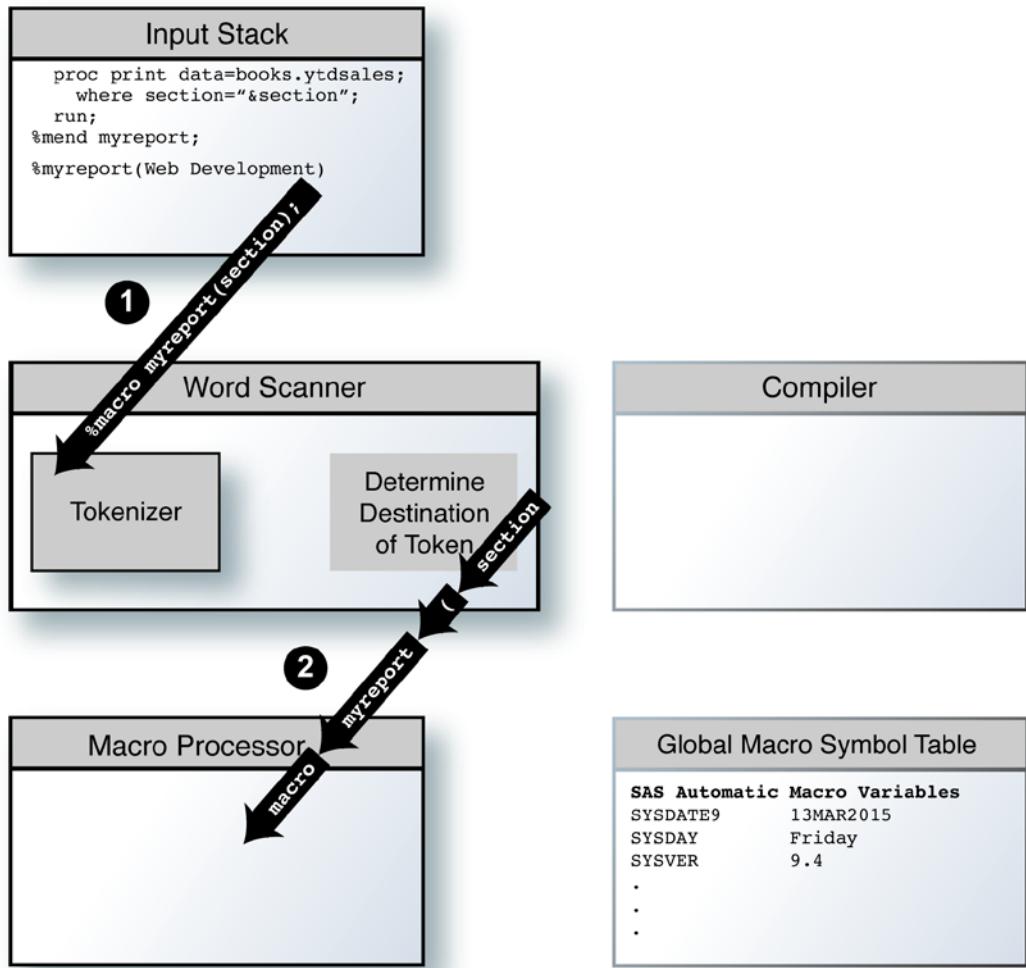
### How a Macro Program Is Compiled

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.

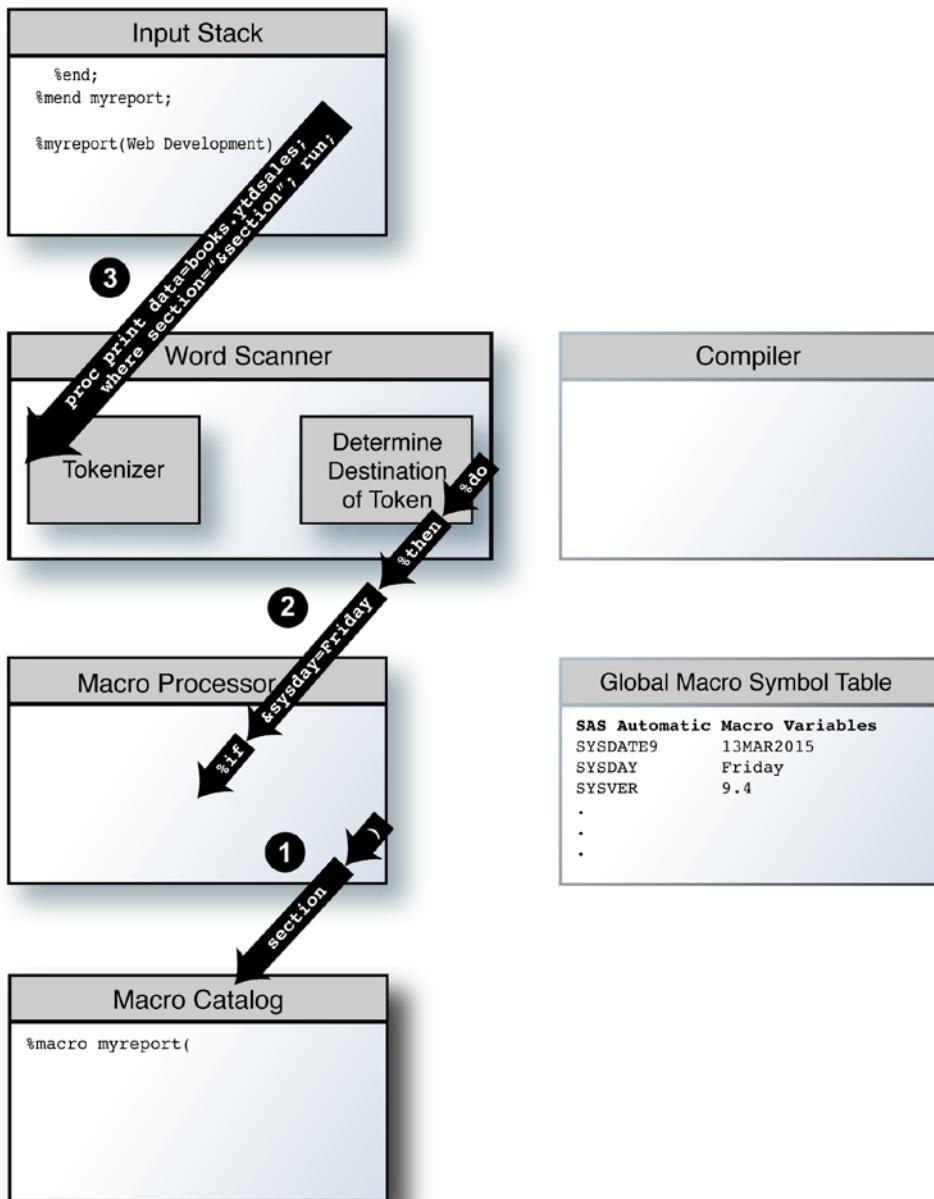
**Figure 5.9 Statements are transferred from the input stack to the word scanner**

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



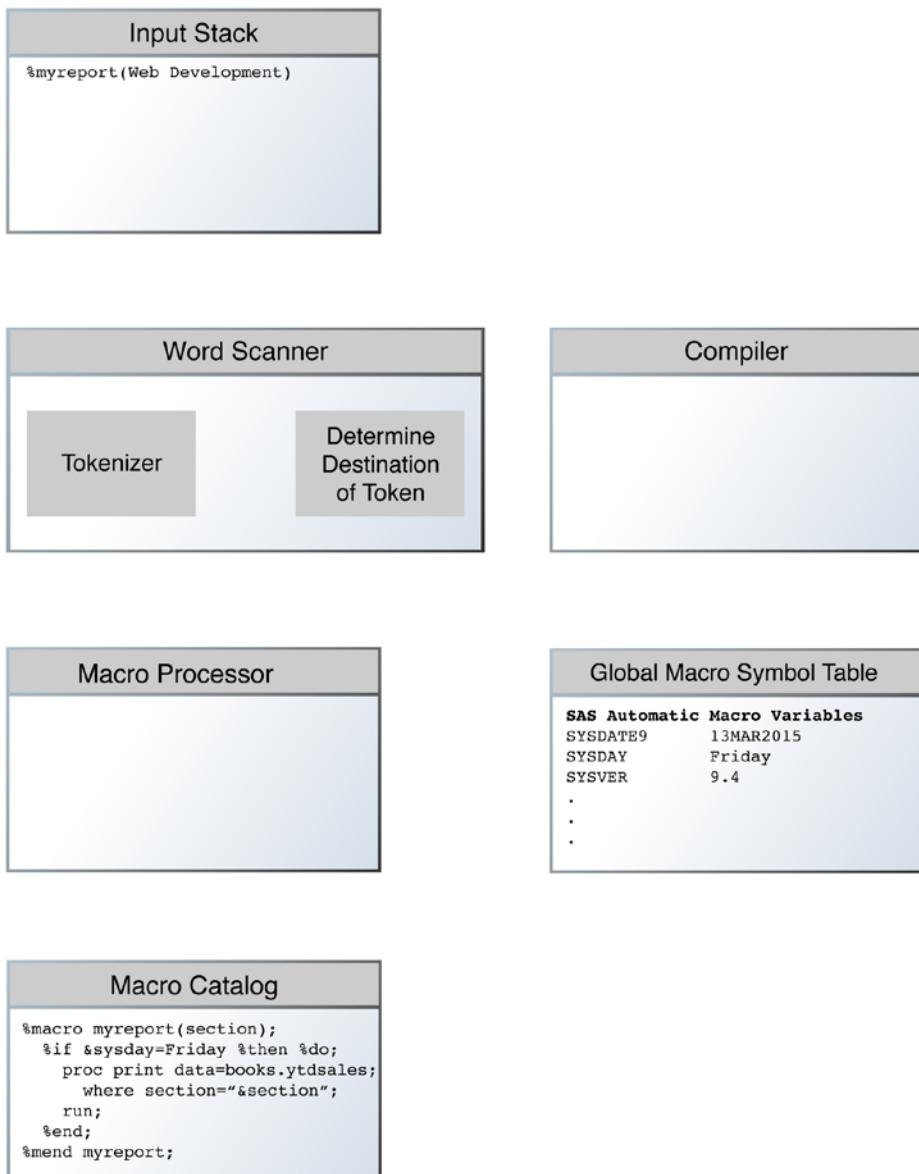
The three steps in Figure 5.10 are:

- 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 executes. 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

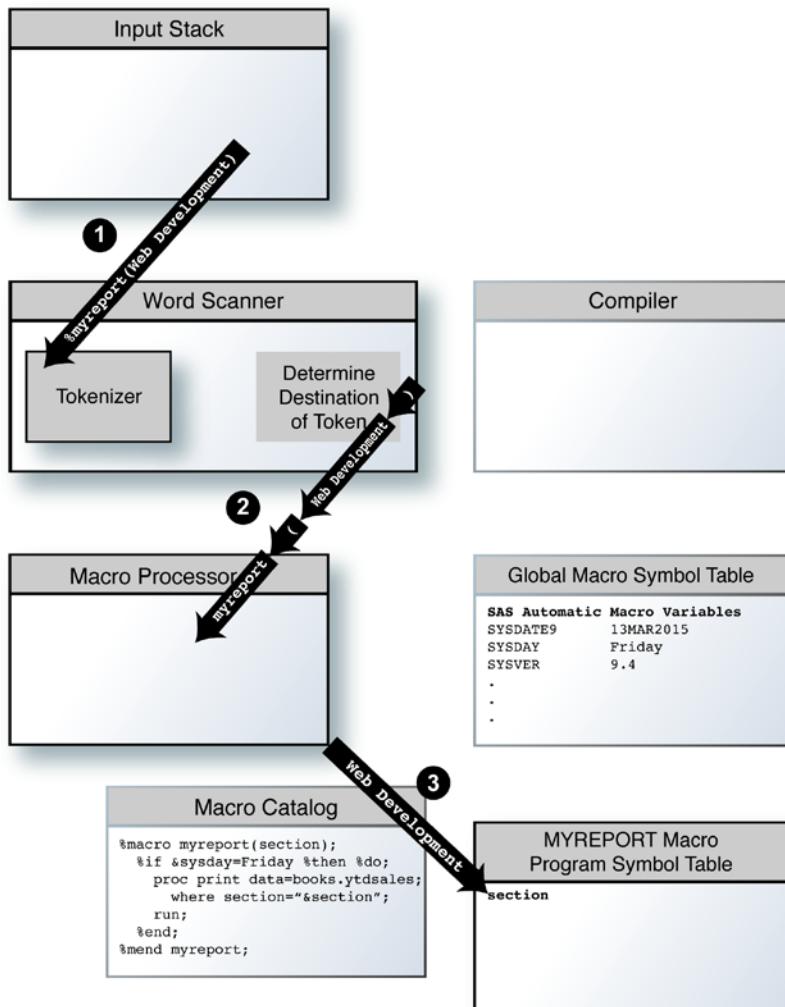


## How a Macro Program Executes

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 Development.

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

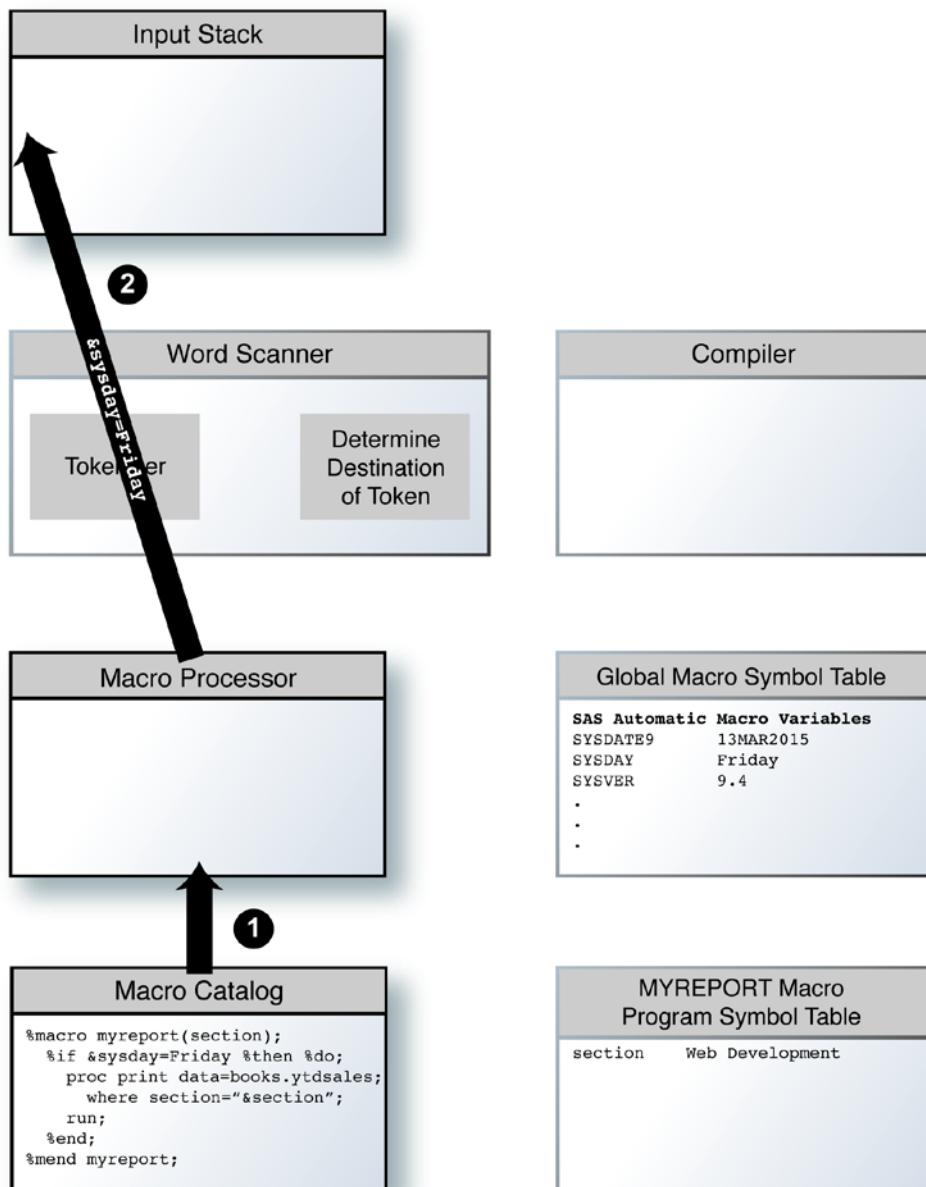


The three steps in Figure 5.12 are:

- ① Macro program MYREPORT has been called.
- ② The word scanner detects the percent sign macro trigger followed by text and transfers tokens to the macro processor.
- ③ 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 continues executing MYREPORT.

Figure 5.13 The macro program MYREPORT continues executing

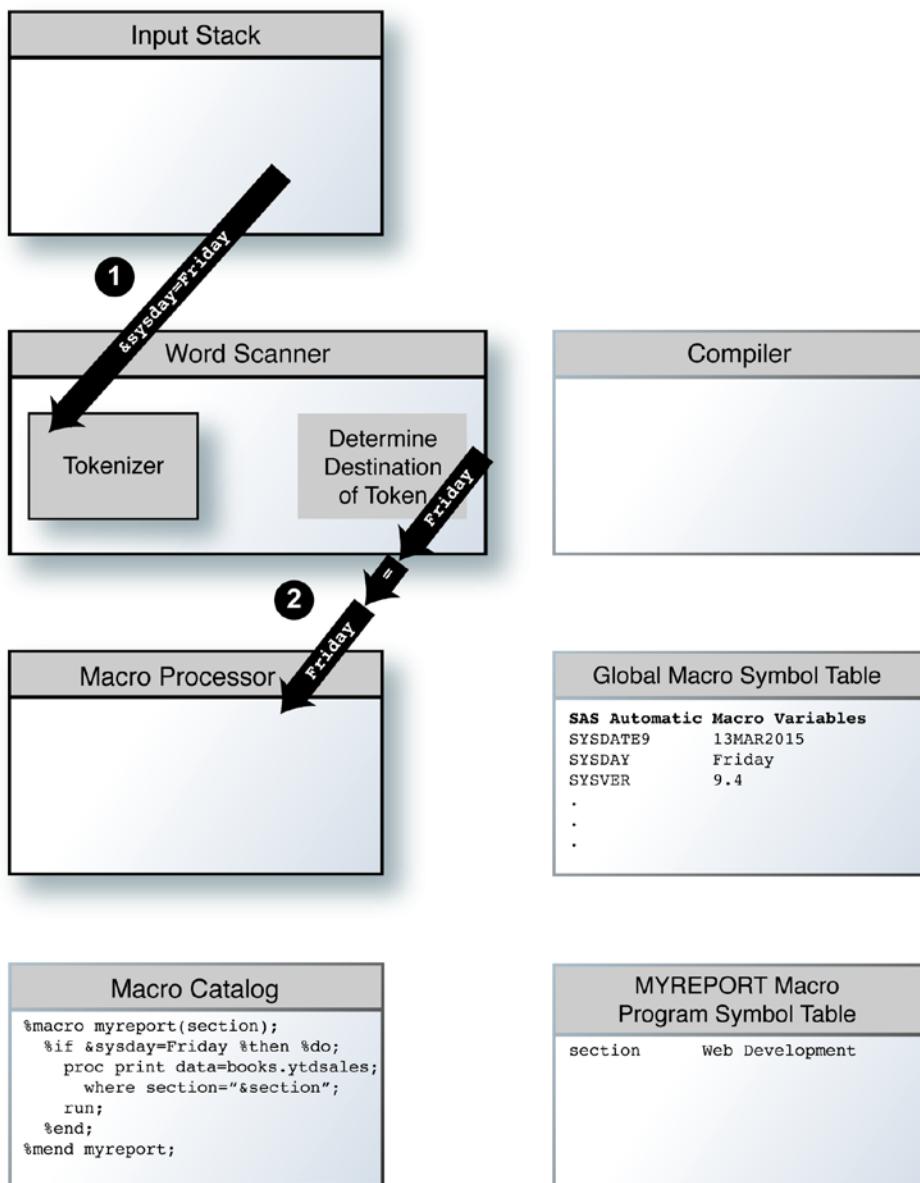


The two steps in Figure 5.13 are:

- 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

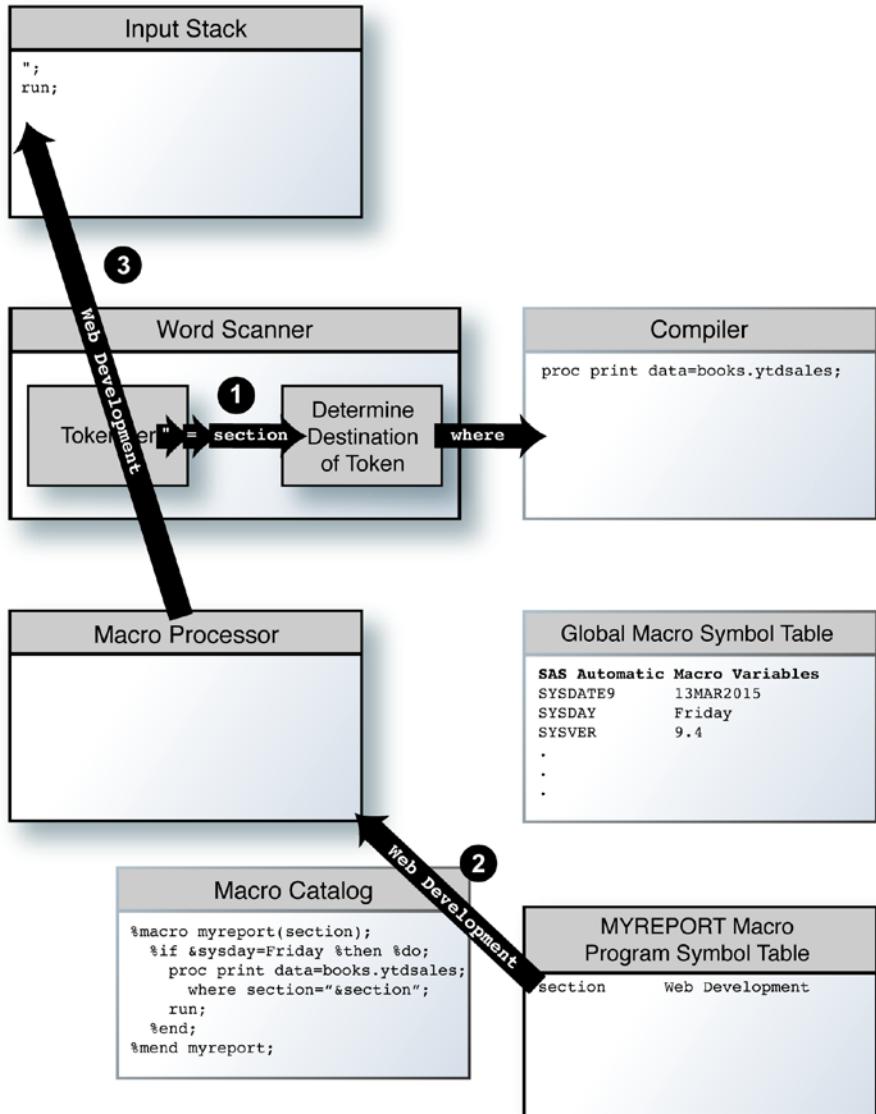


The two steps in Figure 5.14 are:

- 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.

Since MYREPORT was run on a Friday, 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**

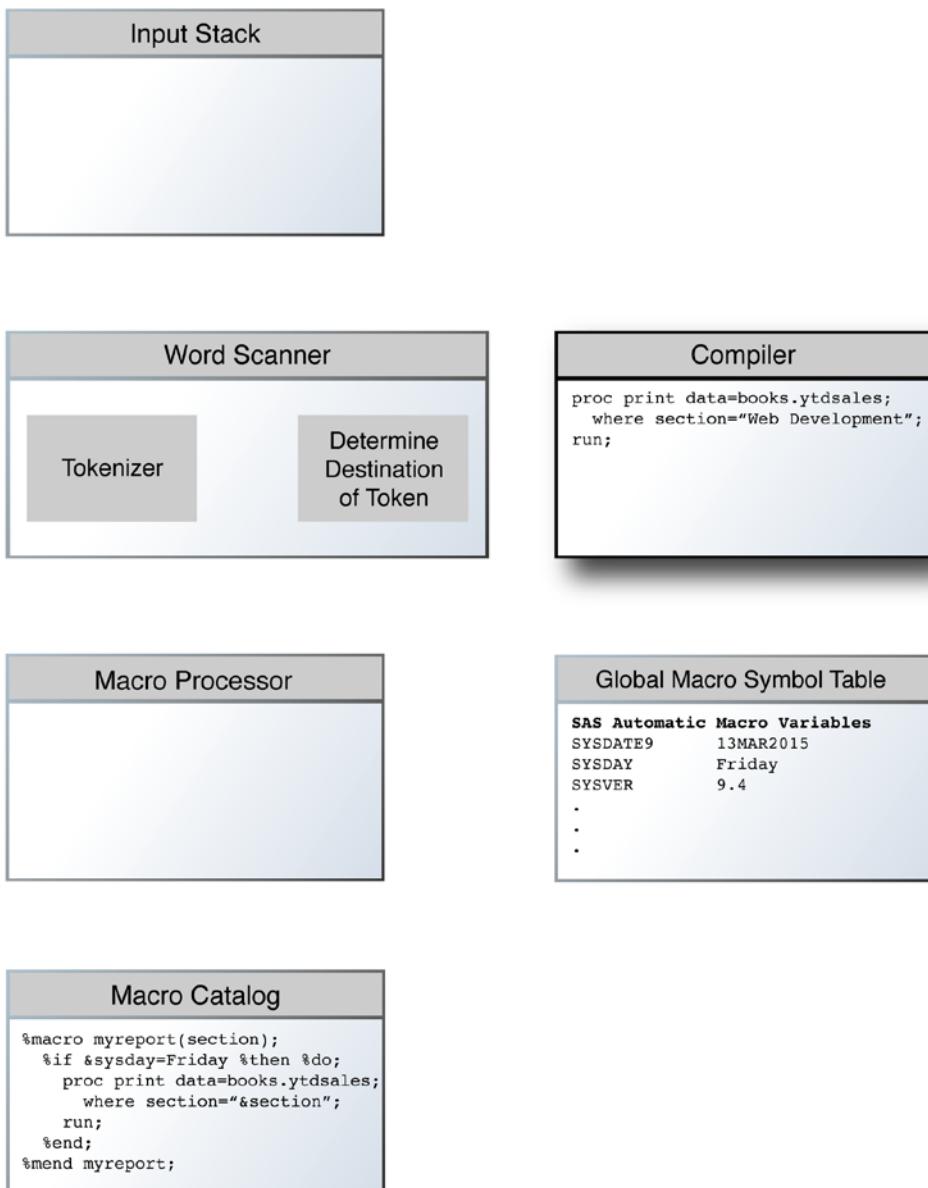


The three steps in Figure 5.15 are:

- ① The word scanner tokenizes the SAS language statements and passes them to the compiler.
- ② 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.
- ③ 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**

<b>Introduction .....</b>	<b>129</b>
<b>Macro Character Functions.....</b>	<b>130</b>
<b>Macro Evaluation Functions.....</b>	<b>133</b>
<b>Macro Quoting Functions .....</b>	<b>135</b>
<b>Macro Variable Attribute Functions.....</b>	<b>137</b>
<b>Other Macro Functions.....</b>	<b>141</b>
Using the %SYSFUNC and %QSYSFUNC Macro Functions .....	141
<b>SAS Supplied Autocall Macro Programs Used Like Functions .....</b>	<b>147</b>

---

## **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 PROC FCMP 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. They can modify their arguments, return information about an argument, or return text values. 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 instead if the result of the macro function might 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( <i>source</i> , <i>string</i> )	returns the position in <i>source</i> of the first character of <i>string</i> .
%LENGTH( <i>string text expression</i> )	returns the length of <i>string</i> or the length of the results of the resolution of <i>text expression</i> .
%SCAN( <i>argument</i> , <i>n &lt;,charlist&lt;,modifiers&gt;&gt;</i> )	searches for the <i>n</i> th word in <i>argument</i> , where <i>argument</i> can be text or an expression and the words in <i>argument</i> are separated by <i>charlist</i> . Specify <i>modifiers</i> to modify how %SCAN determines word boundaries in

<pre>%SUBSTR(argument,position&lt;,length&gt;)</pre>	<p><i>argument</i>. Use %QSCAN when you need to mask special characters or mnemonic operators in the result.</p>
<pre>%UPCASE(string text expression)</pre>	<p>extracts a substring of <i>length</i> characters from <i>argument</i> starting at <i>position</i>. Use %QSUBSTR when you need to mask special characters or mnemonic operators in the result.</p>
	<p>converts <i>character string</i> or <i>text expression</i> to uppercase. Use %QUPCASE when you need to mask special characters or mnemonic operators in the result.</p>

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

Example 6.1 shows how the %SUBSTR function extracts text from strings of characters. The WHERE statement in the PROC MEANS step selects observations from the first day of the current month through the day the program was run. The current month is determined by extracting that information from the value of automatic macro variable SYSDATE.

```
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, 2014.

```
proc means data=books.ytdsales;
  title "Sales for 01SEP through 15SEP2014";
  where "01SEP13"d le datesold le "15SEP14"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 Example 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, Example 6.2 does not specify the optional third argument to %SCAN.

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

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

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

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

While not used in this simple example, you can add modifiers to %SCAN to alter how %SCAN determines boundaries between items in your argument string. For example, if you add the B modifier, the macro processor scans your argument string from right to left instead of the default direction of left to right.

```
%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 Example 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 BOOKTITLE are converted to uppercase. This increases the likelihood of matches when the two are compared.

```
%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 the macro processor converts back to text. Otherwise, you can request %SYSEVALF to convert the result to a different format, as shown in Table 6.2. When you want to use one of these four conversion types, specify it 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 of the usage of %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);	3.333333333333
%put %sysevalf(&b/&a,ceil);	4
%put %sysevalf(&b/&a,floor);	3
%put %sysevalf(&b/&a,boolean);	1

---

%PUT Statement	Results in SAS log
<pre>%let missvalue=.; %put %sysevalf(&amp;b-&amp;missvalue,                boolean);</pre>	<p>NOTE: Missing values were generated as a result of performing an operation on missing values during %SYSEVALF expression evaluation.</p> <p>0</p>

---

## 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 it compiles the instruction. 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 two types based upon when they act: compilation and 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 ( <i>character-string</i>   <i>text expression</i> )	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 ( <i>character-string</i>   <i>text expression</i> )	Does the same as %BQUOTE and additionally masks ampersands (&) and percent signs (%). However, SAS recommends that you use %SUPERQ instead of %NRBQUOTE.
%QUOTE ( <i>character-string</i>   <i>text expression</i> )	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 ( <i>character-string</i>   <i>text expression</i> )	Does the same as %QUOTE and additionally masks ampersands (&) and percent signs (%). However, SAS recommends that you use %SUPERQ instead of %NRQUOTE.
%STR ( <i>character-string</i> )	Mask special characters and mnemonic operators in constant text at macro compilation.
%NRSTR ( <i>character-string</i> )	Does the same as %STR and additionally masks ampersands (&) and percent signs (%).

---

Function	Action
%SUPERQ ( <i>macro-variable-name</i> )	Masks all special characters including ampersands (&), percent signs (%), and mnemonic operators at macro execution and prevents further resolution of the value. Returns the value of a macro variable without attempting to resolve any macro program or macro variable references in the value.
%UNQUOTE ( <i>character-string</i>   <i>text expression</i> )	Unmasks all special characters and mnemonic operators in a value at macro execution.

---

## Macro Variable Attribute Functions

Table 6.5 lists the three macro variable attribute functions that 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 Macro variable attribute functions**

---

Function	Action
%SYMEXIST ( <i>macro-variable-name</i> )	Determines 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 returns a value of 1; otherwise, it returns a 0.
%SYMGLOBL ( <i>macro-variable-name</i> )	Determines 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 ( <i>macro-variable-name</i> )	Determines 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.

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

```
%* 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 (Software)
```

```

/* 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  %syndel 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(Software)
***** 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 857 observations read from the data set
      BOOKS.YTDSALES.
      WHERE section='Software';
NOTE: The data set WORK.TEMP has 857 observations and 11
      variables.
NOTE: DATA statement used (Total process time):
      real time            0.03 seconds
      cpu time             0.00 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 857 observations read from the data set
      WORK.TEMP.
NOTE: PROCEDURE TABULATE used (Total process time):
      real time            0.12 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 presents the topic of masking special characters and mnemonic operators, and it 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 ( <i>function(argument(s))</i> <i>&lt;,format&gt;</i> )	executes SAS <i>function</i> or user-written <i>function</i> and returns the results to the macro facility (see also macro statement %SYSCALL)
%QSYSFUNC ( <i>function(argument(s))</i> <i>&lt;,format&gt;</i> )	does the same as %SYSFUNC and it also masks special characters and mnemonic operators in the result
%SYSGET ( <i>host-environment-variable</i> )	returns the value of <i>host-environment-variable</i> to the macro facility
%SYSPROD ( <i>SAS-product</i> )	returns a code to indicate whether <i>SAS-product</i> is licensed at the site where SAS 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. Providing 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 Example 6.5 shows how the elements of a date can be formatted using %SYSFUNC and the DATE SAS language function.

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

On January 30, 2014, the title statement would resolve to

```
Sales for January 2014
```

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

Example 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.

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

%getopt(number)
%getopt(orientation)
%getopt(date)
%getopt(symbolgen)
%getopt(compress)
```

The SAS log for Example 6.6 follows.

```
58  %getopt(number)
Option number = NUMBER
59  %getopt(orientation)
Option orientation = PORTRAIT
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 Example 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 it is to be determined to be 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.

Example 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 SAS names. 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.

```

%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 their mean. 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.

```
%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. In Example 6.8, 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 include decimal places. Also, %EVAL would return an integer result, which would be inaccurate. Code that includes %SYSEVALF follows.

```
%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 the first group of statements in this example:

```
***** 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**

Example 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 this value 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.

```
%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;
  title "Publisher List Report &sysdate9";
  title2 "Last Update of &dsname:
sysfunc(putn(&lastupdate,datetime.))";

  column publisher saleprice;
  define publisher / group;
  define saleprice / format=dollar11.2;
  rbreak after / summarize;
run;
```

Output 6.1 presents the output from Example 6.9.

**Output 6.1 Output from Example 6.9**

**Publisher List Report 08JAN2015**  
**Last Update of books.ytdsales: 04JAN15:16:21:08**

Publisher	Sale Price
AMZ Publishers	\$9,900.29
Bookstore Brand Titles	\$11,141.01
Doe&Lee Ltd.	\$4,248.17
Eversons Books	\$4,051.97
IT Training Texts	\$13,377.10
Mainst Media	\$6,342.22
Nifty New Books	\$6,422.89
Northern Associates Titles	\$13,411.67
Popular Names Publishers	\$3,747.11
Professional House Titles	\$7,017.76
Technology Smith	\$5,503.41
Wide-World Titles	\$9,981.80
	<b>\$95,145.40</b>

**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 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 might 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 ( <i>text</i>   <i>text expression</i> )	Remove multiple, leading, and trailing blanks from the argument. Use %QCMPRES if the result might contain a special character or mnemonic operator.
%DATATYP ( <i>text</i>   <i>text expression</i> )	Returns the data type (CHAR or NUMERIC) of a value.
%LEFT ( <i>text</i>   <i>text expression</i> )	Aligns an argument to the left by removing leading blanks. Use %QLEFT if the result might contain a special character or mnemonic operator.
%LOWCASE ( <i>text</i>   <i>text expression</i> )	Changes a value from uppercase characters to lowercase. Use %QLOWCASE if the result might contain a special character or mnemonic operator.
%TRIM ( <i>text</i>   <i>text expression</i> )	Remove trailing blanks from the argument. Use %QTRIM if the result might contain a special character or mnemonic operator.
%VERIFY ( <i>source</i>   <i>excerpt</i> )	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 the first section of code 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 call to %VERIFY in this example can return only 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 you do not enclose the text in quotation marks as you would when processing text in SAS language statements.

```
%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. The following program uses %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 this revised version of Example 6.1.

```
%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 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**

<b>Introduction .....</b>	<b>151</b>
<b>Macro Language Statements.....</b>	<b>151</b>
<b>Constructing Macro Expressions.....</b>	<b>154</b>
Understanding Arithmetic Expressions.....	155
Understanding Logical Expressions .....	156
Understanding the IN Operator As Used in Macro Language Statements .....	157
<b>Conditional Processing with the Macro Language.....</b>	<b>158</b>
<b>Iterative Processing with the Macro Language .....</b>	<b>167</b>
Writing Iterative %DO Loops in the Macro Language.....	167
Conditional Iteration with %DO %UNTIL.....	170
Conditional Iteration with %DO %WHILE .....	172
<b>Branching in Macro Processing.....</b>	<b>174</b>

---

## **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 stored compiled macro library catalog.
%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.
%MEND	End 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 (see also the %SYSFUNC macro function).

Statement	Action
%SYSEXEC	Execute operating system commands immediately and return the success or failure status of the command 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. The %TO specification is required, and the %BY specification is optional.
%DO %UNTIL	Repetitively execute a section of macro code <i>until</i> 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 <i>while</i> 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

---

Statement	Action
	which the %LOCAL statement was issued.
%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, MAX, and concatenation (||) operators are not available in the macro language. The IN operator requires that you set the MINOPERATOR option. 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**

<b>Operator</b>	<b>Mnemonic</b>	<b>Action</b>	<b>Precedence Rating</b>
<code>**</code>		exponentiation	1
<code>+</code>		positive prefix	2
<code>-</code>		negative prefix	2
<code>¬</code>	NOT	logical not*	3
<code>^</code>			
<code>~</code>			
<code>*</code>		multiplication	4
<code>/</code>		division	4
<code>+</code>		addition	5
<code>-</code>		subtraction	5
<code>&lt;</code>	LT	less than	6
<code>&lt;=</code>	LE	less than or equal to	6
<code>=</code>	EQ	equal	6
<code>#</code>	IN	equal to one of a list	6
<code>= ~ = ^ =</code>			
<code>=</code>	NE	not equal*	6
<code>&gt;</code>	GT	greater than	6
<code>&gt;=</code>	GE	greater than or equal to	6
<code>&amp;</code>	AND	logical and	7
<code> </code>	OR	logical or	8

\*The symbol to use depends on your keyboard.

## 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 and you want to use them in calculations.

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. Looking for one of the values 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;
```

You can simplify the %IF statement if you use the IN operator as shown in the following statement. The list of acceptable values follow the IN operator.

```
%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=',';
```

You will need to set the SAS system option MINOPERATOR. The SAS system option MINOPERATOR | NOMINOPERATOR controls whether the SAS Macro Facility recognizes the word "IN" (case insensitive) and special symbol # as infix operators when it evaluates logical or integer expressions. The default setting is NOMINOPERATOR.

You can set MINOPERATOR | NOMINOPERATOR with either an OPTIONS statement or by adding the option to the %MACRO statement that defines the macro program where you need to evaluate an IN expression. The setting that you specify on the %MACRO statement overrides the system option setting for the duration of execution of the macro program.

---

## 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 the macro program 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 has two parameters. Four different types of logical expression evaluations that compare the two parameters are made with each call to COMP2VARS. Example 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.

```
%macro comp2vars(value1,value2);
%put COMPARISION 1:;
%if &value1 ne &value2 %then
  %put &value1 is not equal to &value2..;
%else %put &value1 equals &value2..;

%put COMPARISION 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**

Example 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 report to produce.

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.

```
%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 saleprice;
  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 Example 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.

Example 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.

```
%macro publisherreport(reptype);
  %let reptype=%upcase(&reptype);

  title "Publisher Report";
  footnote
    "Macro Program: &sysmacroname  Report Type: &reptype";

  proc report data=books.ytdsales nowd;
    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;
    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';
    %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;
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;
  column publisher saleprice cost profit;

  define publisher / group;
  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;
  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;
  column publisher saleprice cost profit n;
  define publisher / group;
  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';

  compute profit;
    profit=saleprice.sum-cost.sum;
  endcomp;

  rbreak after / summarize;
  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;
  column publisher saleprice cost profit
    datesold,(saleprice=saleprice2);

  define publisher / group;
  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;
  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 Example 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. The multiple publishers mapping to one vendor are listed after the IN operator, and names of the publishers are separated by exclamation points.

The default delimiter is a space. Since some values for PUBLISHER contain spaces, a delimiter other than a space must be used to separate publisher names.

To find out the delimiter setting of your SAS session, check the value of the MINDELIMITER SAS option. To use a different delimiter, such as the exclamation point, specify the new delimiter either as a SAS option or specify it on the MINDELIMITER= option of the %MACRO statement. The MINDELIMITER= option on the %MACRO statement overrides the current setting of the MINDELIMITER SAS option during execution of the macro program.

Note that the MINOPERATOR SAS option must be in effect as well when Example 7.4 is submitted. This option controls whether the SAS Macro Facility recognizes the word "IN" (case insensitive) and the special symbol # as infix operators when it evaluates logical or integer expressions. Example 7.4 starts with an OPTIONS statement that sets MINOPERATOR.

```
options minoperator;
%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 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. However, any reference to it later within the loop requires an ampersand in front of the reference.

The start and stop values and the %BY 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 1, 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

Example 7.5 uses the iterative %DO to generate several PROC TABULATE and PROC SGPLOT steps. Macro program MULTREP generates statistics and a bar chart for each year between the bounds on the %DO statement. In this example, PROC TABULATE and PROC SGPLOT are each executed three times: once for 2011, once for 2012, and once for 2013.

```
%macro multrep(startyear,stopyear);
%do yrvalue=&startyear %to &stopyear;
```

```

title "Sales Report for &yrvalue";
proc tabulate data=sales.year&yrvalue;
    class section;
    var cost listprice saleprice;
    tables section all='Total',
        (cost listprice saleprice)*
            (n*f=6. (min max sum)*f=dollar8.2);
run;

proc sgplot data= sales.year&yrvalue;
    hbar section / response=saleprice stat=sum datalabel;
    yaxis display=(nolabel noticks);
run;
%end;
%mend multrep;

-----Produce 3 sets of reports: one for 2011, one for 2012,
-----and one for 2013;
%multrep(2011,2013)

```

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 2011";
proc tabulate data=sales.year2011;
    class section;
    var cost listprice saleprice;
    tables section all='Total',(cost listprice saleprice)*
        (n*f=6. (min max sum)*f=dollar8.2);
run;
proc sgplot data=sales.year2011;
    hbar section / response=saleprice stat=sum datalabel;
    yaxis display=(nolabel noticks);
run;

title "Sales Report for 2012";
proc tabulate data=sales.year2012;
    class section;
    var cost listprice saleprice;
    tables section all='Total',(cost listprice saleprice)*
        (n*f=6. (min max sum)*f=dollar8.2);
run;
proc sgplot data=sales.year2012;
    hbar section / response=saleprice stat=sum datalabel;
    yaxis display=(nolabel noticks);
run;

title "Sales Report for 2013";
proc tabulate data=sales.year2013;
    class section;

```

```

var cost listprice saleprice;
tables section all='Total', (cost listprice saleprice) *
(n*f=6. (min max sum)*f=dollar8.2);
run;
proc sgplot data=sales.year2013;
hbar section / response=saleprice stat=sum datalabel;
yaxis display=(nolabel noticks);
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 Example 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. A semicolon placed after the data set reference 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, and this 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.

```

%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 sgplot data=manyyears;
    title "Charts Analyze Data for: &yearstring";
    hbar section / response=saleprice stat=sum datalabel;
    yaxis display=(nolabel noticks);
  run;
%mend sumyears;

-----Concatenate three data sets: one from 2011, one from;
-----2012, and one from 2013;
%sumyears(2011,2013)

```

The macro processor resolves the call to YEARLYCHARTS as follows.

```
data manyyears;
  set sales.year2011 sales.year2012 sales.year2013;
run;

proc gchart data=manyyears;
  title "Charts Analyze Data for: 2011 2012 2013";
  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

Example 7.7 demonstrates the use of %DO %UNTIL. Macro program MOSALES 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.

Example 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 1 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 it 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.

Example 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.

```
%macro mosales / parmbuff;
  %let listindex=1;
  %do %until (%scan(&syspbuff,&listindex) eq );
    %let repmonth=%scan(&syspbuff,&listindex);
    proc means data=books.ytdsales n sum maxdec=0;
      %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 maxdec=0;
  title "Sales during month 3";
  where month(datesold)=3;
  class section;
  var saleprice;
```

```

run;
proc means data=books.ytdsales n sum maxdec=0;
  title "Sales during month 5";
  where month(datesold)=5;
  class section;
  var saleprice;
run;
proc means data=books.ytdsales n sum maxdec=0;
  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 maxdec=0;
  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

Example 7.8 shows an application of %DO %WHILE. Macro program MOYRSALES computes sales statistics for a month in specific years. It has two parameters: YEARLIST and REPMONTH.

The parameter YEARLIST is defined to be a list of years for which to compute sales statistics. The second parameter, REPMONTH, is the month within each year 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 MOYRSALES requests statistics for May in 2011 and in 2013. The %DO %WHILE loop executes twice, once for each year. The %SCAN function selects each year from YEARLIST. The macro variable YEARIDX determines which item the %SCAN function should select, and the program increments YEARIDX by one at the bottom of the %DO %WHILE loop.

The %DO %WHILE loop does not execute a third time. On the third iteration, the %SCAN function does not find a third year. Therefore, the macro expression on the %DO %WHILE statement resolves to false, and this stops the loop.

```
%macro moyrsales(yearlist,repmonth);
  %let yearidx=1;
  %do %while (%scan(&yearlist,&yearidx) ne );
    %let saleyear=%scan(&yearlist,&yearidx);
    proc means data=sales.year&saleyear n sum maxdec=0;
      title "Sales during &repmonth/&saleyear";
      where month(datesold)=&repmonth;
      class section;
      var saleprice;
    run;
    %let yearidx=%eval(&yearidx+1);
  %end;
%mend moyrsales;

%moyrsales(2011 2013,5)
```

After resolution by the macro processor, the SAS code submitted for compilation and execution is as follows. A PROC MEANS step is created for each year.

```
proc means data=sales.year2011 n sum maxdec=0;
  title "Sales during 5/2011";
  where month(datesold)=5;
  class section;
  var saleprice;
run;

proc means data=sales.year2013 n sum maxdec=0;
  title "Sales during 5/2013";
  where 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 years are specified on the call to %MOYRSALES as follows:

```
%moyrsales(,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 to invoke a macro program that is 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

Example 7.9 shows how labels and %GOTO statements can be used. Macro program DETAIL 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 the value specified for DSNAME.

The third parameter, NUMOBS=, is a keyword parameter that is initialized to MAX. The value assigned to NUMOBS specifies the number of observations to list with PROC PRINT. The value MAX indicates that all observations in data set DSNAME will be printed.

This example calls DETAIL three times. The code that executes is described below.

```
%macro detail(dsname,varlist,numobs=max);
  %* Does DSNAME exist?;
  %let foundit=%sysfunc(exist(&dsname));
  %if &foundit le 0 %then %goto nodataset;

  title "PROC PRINT of &dsname";
  proc print data=&dsname(obs=&numobs);
    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,obs=5)

-----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 data set BOOKS.YTDSALES exists. The PROC PRINT step lists the variables specified in VARLIST, and it lists the first five observations in BOOKS.YTDSALES. After PROC PRINT completes processing, 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(obs=5)";
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(obs=max);
  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**

<b>Introduction .....</b>	<b>177</b>
<b>Why Are Quoting Functions Called Quoting Functions?.....</b>	<b>178</b>
<b>Illustrating the Need for Macro Quoting Functions .....</b>	<b>179</b>
<b>Describing the Commonly Used Macro Quoting Functions.....</b>	<b>180</b>
<b>Understanding How Macro Quoting Functions Work .....</b>	<b>181</b>
<b>Applying Macro Quoting Functions.....</b>	<b>182</b>
<b>Specifying Macro Program Parameters That Contain Special Characters or Mnemonic Operators.....</b>	<b>189</b>
<b>Unmasking Text and the %UNQUOTE Function .....</b>	<b>198</b>
<b>Using Quoting Versions of Macro Character Functions and Autocall Macro Programs.....</b>	<b>198</b>

---

## **Introduction**

The SAS macro language is a text-handling language. The macro facility relies on specific syntax structures in the language to perform its tasks when it constructs SAS code for you. The macro facility relies on triggers such as ampersands and percent signs to understand when you want it to resolve a macro variable and to 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 four commonly used quoting functions:

- %STR and %NRSTR
- %BQUOTE
- %SUPERQ

This chapter also describes a fifth macro quoting function, %UNQUOTE, which removes the mask from a value so that the special characters and mnemonic operators in the value are not treated as text.

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
- %UPCASE

The autocall macro programs include:

- %QCMPRES
- %LEFT
- %LOWCASE
- %TRIM

---

## 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 that 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 any of the semicolons within the parentheses 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 four most commonly used macro quoting functions: %STR, %NRSTR, %BQUOTE, and %SUPERQ. Three lesser-used functions, %NRBQUOTE, %QUOTE, and %NRQUOTE, are mentioned at the end of the section.

Use the %STR and %NRSTR functions to mask items during compilation. Use the %BQUOTE function to mask text or resolved values of text expressions during execution.

The “NR” function, %NRSTR, does the same as its non-“NR” counterpart, %STR, and this function also masks the ampersand (&) and percent sign (%) macro triggers.

The %SUPERQ function is also an execution function. Use it when you want to mask the value of a macro variable so that this 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 can 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**

The %BQUOTE function masks special characters and mnemonic operators contained in the results from resolving macro expressions. The macro processor resolves macro expressions during *execution*. Use this function 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 %BQUOTE function masks resolved values, and resolution occurs at execution. %BQUOTE masks all special characters and mnemonic operators except for ampersands and percent signs. Use %SUPERQ when you want to mask the same items as %BQUOTE and additionally mask 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. 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.

### **Other Macro Quoting Functions: %NRBQUOTE, %QUOTE, and %NRQUOTE**

The %NRBQUOTE function masks the same items as %BQUOTE and it additionally masks ampersands and percent signs. However, SAS recommends that you use %SUPERQ instead of %NRBQUOTE since %SUPERQ is more complete in its masking.

The %QUOTE and %NRQUOTE functions 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 and unmatched quotation marks. 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.

```
%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**

Example 8.2 demonstrates how to mask 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 Example 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.

```
%let month=%substr(Jan, Feb, Mar,5,3);
%put WITHOUT Quoting Functions &=month;
%let month=%substr(%str(Jan, Feb, Mar),5,3);
%put WITH Quoting Functions &=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;
WITHOUT Quoting Functions MONTH=
1180 %let month=%substr(%str(Jan, Feb, Mar),5,3);
1181 %put WITH Quoting Functions &=month;
WITH Quoting Functions MONTH=Feb
```

### Example 8.3: Using %STR to Preserve Leading and Trailing Blanks

Example 8.3 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.

```
%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**

Example 8.4 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, and the %STR function does not mask macro triggers. The %NRSTR function masks all that %STR masks, and it also masks macro triggers.

When the ampersand is not masked, 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.

When the percent sign is not masked, 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.

```
%let reporttitle=Jan&Feb %Sales Report;
%put WITHOUT Quoting Functions &=reporttitle;
%let reporttitle=%nrstr(Jan&Feb %Sales Report);
%put WITH Quoting Functions &=reporttitle;
```

The SAS log for the four statements in Example 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;
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;
WITH Quoting Functions REPORTTITLE=Jan&Feb %Sales Report

```

### **Example 8.5: Using %STR to Mask Unbalanced Quotation Marks and Preceding Percent Signs**

Example 8.5 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 set of eight statements without applying %STR and you do not include the preceding percent signs in the first %LET statement, the second through fourth 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 masks 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.

```

%let names=O'DONOVAN,O'HARA,O'MALLEY;
%let name3=%qscan(&names,3);
%put WITHOUT STR and Percent Signs &=names;
%put WITHOUT STR Quoting Function &=name3;
%let names=%str(O%'DONOVAN,O%'HARA,O%'MALLEY);
%let name3=%qscan(&names,3);
%put WITH STR and Percent Signs &=names;
%put WITH STR Quoting Function &=name3;

```

Because the first group of statements do not execute, 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;

```

```

WITH STR and Percent Signs NAMES=O'DONOVAN,O'HARA,O'MALLEY
31   %put WITH STR Quoting Function &=name3;
WITH STR Quoting Function NAME3=O'MALLEY

```

### **Example 8.6: Masking Macro Triggers and Unbalanced Quotation Marks with %NRSTR and Preceding Percent Signs**

Example 8.6 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 of %STR so that the two ampersands are masked. 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.

```

%let names=%nrstr(O%DONOVAN&O%HARA&O%'MALLEY);
%let name3=%qscan(&names,3);
%put WITH NRSTR Quoting Function &=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;
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 Example 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.

Example 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 to treat a mnemonic operator as text. In Example 8.7, you would use a quoting function to mask OR so that the macro processor treats it 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.

```
%let state=OR;
%let value=%sysevalf(&state eq OR, boolean);
%put WITHOUT Quoting Functions &=value;
%let value=%sysevalf( %bquote(&state) eq %str(OR) , boolean);
%put WITH Quoting Functions &=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;
WITHOUT Quoting Functions VALUE=
1203 %let value=%sysevalf( %bquote(&state) eq %str(OR) ,
1203! boolean);
1204 %put WITH Quoting Functions &=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 Example 8.8 masks from interpretation text that looks like a macro variable reference. Example 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 SALES\_DL. 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.

```
proc means data=books.ytdsales
            (where=(publisher='Doe&Lee Ltd.')) noprint;
  var saleprice;
  output out=salesdl sum=;
run;
```

```

data _null_;
  set salesdl;
  call symputx('totalsales_dl',
               cat('The total sales for Doe&Lee Ltd is ',
                   put(saleprice,dollar10.2),'.'));
run;
footnote "%superq(totalsales dl)";

```

Example 8.8 executes without any warnings or errors. The footnote becomes:

```
The total sales for Doe&Lee Ltd is $ $4,248.97.
```

Without the %SUPERQ function, the macro processor writes a WARNING to the SAS log. The FOOTNOTE statement without the %SUPERQ function is:

```
footnote "&totalsales_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 might 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 might have. 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 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 could 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 Example 8.9 generates a PROC PRINT step that lists books sold during specific months from a specific section. It has two parameters. The first parameter is the list of months with months specified numerically. The second parameter 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 Software section. The underlined part of each call to MOSECTDETAIL shows the value that the macro processor interprets as the first parameter, MONTHLIST.

```
%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,Software)

%mosectdetail(%str(3,6),Software)
```

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="Software";
  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: Preventing Misinterpretation of Special Characters in Parameter Values**

Example 8.10 defines a macro program called PUBLISHERSALES that constructs a PROC REPORT step and has three positional parameters. The first parameter, STYLE, specifies the ODS style of the HTML 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 parameter values exactly as they are specified 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 FONT= attribute requires that the values supplied to it be enclosed in parentheses. The %STR quoting function also masks the parentheses that are found in the FONT= specification for STYLEHEADER in Example 8.10.

The call to PUBLISHERSALES in Example 8.10:

- formats the report in the MEADOW style
- italicizes the headers and writes them in 14 pt Garamond
- writes the body of the report in 12 pt.

```
%macro publishersales(style,styleheader,stylereport);
  ods html style=&style;

  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 html close;
%mend publishersales;

%publishersales(meadow,
  %str(font style=italic font size=14pt font=(Garamond)),
  %str(font size=12pt))
```

After resolution by the macro processor, SAS submits the following code:

```
ods listing close;
ods html style=meadow;

title "Sales by Publisher";
proc report data=books.ytdsales
  style(header)={font_style=italic font_size=14pt
                 font=(Garamond)}
  style(report)={font_size=12pt}
  nowd;
  column publisher saleprice n;
  define publisher / group;
  define saleprice / format=dollar10.2;
run;

ods html close;
```

Consider what happens if the %STR functions does not surround the second and third parameters in the call to PUBLISHERSALES.

```
%publishersales(meadow,
  font_style=italic font_size=14pt font=(Garamond),
  font_size=12pt)
```

The macro processor does not execute this call to PUBLISHERSALES. It stops after reading the first two attributes specified in the second positional parameter. It interprets FONT\_STYLE=ITALIC and FONT\_SIZE=14PT as keyword parameters and not as elements in STYLEHEADER, the second positional parameter. Since PUBLISHERSALES does not define any keyword parameters,

the call to PUBLISHERSALES ends in error and SAS writes the following messages to the SAS log.

```
ERROR: The keyword parameter FONT_STYLE was not defined with
      the macro.
ERROR: The keyword parameter FONT_SIZE was not defined with the
      macro.
```

### **Example 8.11: Masking Special Characters and Mnemonic Operators in Parameter Values**

Example 8.11 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 ways of 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.

Example 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.

```
%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;
  footnotel;
%end;
%else %do;
  footnote justify=&jfootnote color=&cfootnote
    "&footnotetext";
%end;
%mend mypages;

options mprint;

-----First call of MYPAGES;
%mypages(titletext=Sales Report,ctitle=blue,
         footnotetext=Last Review Date: Feb 1%str(,,) 2014)

-----Second call of MYPAGES;
%mypages(titletext=2013+ 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(AuditedApproved),
         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 parameter 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, 2014";

```

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 2014. 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, 2014)
```

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 "2013+ Sales";
footnote justify=right color=black "Prepared with SAS 9.4";
```

While it is not necessary to mask the plus sign in the macro program call, it is necessary that the value be masked 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 FOOTNOTETEXT 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.

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.
```

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. SAS resolves the references to SYSVER when the call to MYPAGES is tokenized. This resolution happens before the %IF statement that applies %SUPERQ to FOOTNOTETEXT executes.

**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 that 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 condition 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, SAS writes 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)
```

Recall that a percent sign can serve as a mask for an unmatched parenthesis. 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 last call to MYPAGES executes as desired, and it produces 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 Example 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.

```
%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 the SAS 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 SAS 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 Example 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.

```
%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**

Example 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 like 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.

```

%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 Example 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**

<b>Introduction .....</b>	<b>201</b>
<b>Understanding DATA Step Interfaces to the Macro Facility.....</b>	<b>201</b>
Understanding the SYMGET Function.....	202
Understanding the SYMPUT and SYMPUTX Call Routines .....	209
Understanding the CALL EXECUTE Routine .....	216
Understanding the RESOLVE Function .....	227
<b>Using Macro Facility Features in PROC SQL.....</b>	<b>231</b>
Creating and Updating Macro Variables with PROC SQL.....	232
Using the Macro Variables Created by PROC SQL.....	238
Displaying Macro Option Settings with PROC SQL and Dictionary Tables .....	241

---

## **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 PROC SQL.

The following interfaces are not described in this chapter and are beyond the scope of this book. For further information about these topics, see Documentation in the Knowledge Base, at <http://support.sas.com>.

- SAS Component Language (SCL) functions and routines
- %SYSLPUT and %SYSRPUT, two macro functions that provide an interface with SAS/CONNECT

---

## **Understanding DATA Step Interfaces to the Macro Facility**

Five functions and four call routines in the SAS language can interact with the macro processor *during* execution of a DATA step. Table 9.1 lists these nine tools.

**Table 9.1 DATA step interface tools**

Tool	Description
CALL SYMDEL( <i>macro-variable</i> , <,'NOWARN')	Deletes <i>macro-variable</i> from the global symbol table. The NOWARN option suppresses the SAS warning in the SAS log if <i>macro-variable</i> does not exist.
SYMEXIST( <i>argument</i> )	Determines if the macro variable specified as <i>argument</i> exists. SYMEXIST returns a 0 if the macro variable does not exist, and a 1 if it does.
SYMGET( <i>argument</i> )	Obtains the value of a macro variable specified as <i>argument</i> and returns this value as a <i>character</i> value.
SYMGLOBL( <i>argument</i> )	Determines if the macro variable specified as <i>argument</i> is global to the DATA step during execution of the DATA step. SYMGLOBL returns a 0 if the macro variable is not in the global symbol table, and a 1 if it is.
SYMLOCAL( <i>argument</i> )	Determines if the macro variable specified as <i>argument</i> is local to the DATA step during execution of the DATA step. SYMLOCAL returns a 0 if the macro variable is not in an existing local symbol table during DATA step execution, and a 1 if it is.
CALL SYMPUT( <i>macro-variable</i> , <i>value</i> )	Assigns <i>value</i> to <i>macro-variable</i> . This routine does not trim leading and trailing blanks.
CALL SYMPUTX( <i>macro-variable</i> , <i>value</i> <, <i>symbol-table</i> >)	Assigns <i>value</i> to <i>macro-variable</i> . 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 (G=global, L=local, and F=most local if it exists).
CALL EXECUTE( <i>argument</i> )	Executes the resolved value of <i>argument</i> . Arguments that resolve to a macro facility reference execute immediately. Arguments that resolve to SAS language statements execute after the DATA step that includes CALL EXECUTE ends.
RESOLVE( <i>argument</i> )	Macro facility resolves <i>argument</i> during DATA step execution where <i>argument</i> can be an expression that includes macro variables and macro program calls.

## Understanding the SYMGET Function

The SYMGET SAS language function retrieves macro variable values from the macro symbol tables during execution of a DATA step. The SYMGET function returns a character value. With this function, you can create and update data set variables with information that the macro processor retrieves from macro variables.

A macro variable that you reference with SYMGET must exist before you apply it 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 if it follows the CALL SYMPUT or CALL SYMPUTX call.

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.

The SYMGET function accepts 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 SAS session.

```
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=CRT283817;
%let networki=NET3UD697;
%let operatin=OPSI18375;
%let programm=PRG8361WQ;
%let software=SFT3521P8;
%let webdevel=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.
  e.g. section=Programming, then COMPSECT="Programm";

  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 *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} newpricel-newprice3;

  format newpricel-newprice3 dollar10.2;

  drop i;

  do i=1 to 3;
    factor{i}=input(symget(cats('factor',put(i,1.))),best8.);
    newprice{i}=factor{i}*saleprice;
  end;
run;

```

The next three examples illustrate the three types of arguments that SYMGET can receive.

### **Example 9.1: Using a Data Set Variable Name as the Argument to the SYMGET Function**

Example 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, %LET statements create six global macro variables, 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 it stores the extracted value in DATA step 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 after blanks in those first eight characters have been removed.

```

%let certific=CRT283817;
%let networki=NET3UD697;
%let operatin=OPSI18375;
%let programm=PRG8361WQ;
%let software=SFT3521P8;
%let webdevel=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.
      e.g. section=Programming, then COMPSECT="Programm";
      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;
```

Output 9.1 presents a partial listing of the PROC PRINT report produced by Example 9.1. The output shows the values assigned to SECTIONID by SYMGET.

**Output 9.1 Partial output from Example 9.1****Defining the Section Identification Code**

Obs	section	compsect	sectionid
1	Software	Software	SFT3521P8
2	Software	Software	SFT3521P8
3	Networking	Networki	NET3UD697
4	Networking	Networki	NET3UD697
5	Programming	Programm	PRG8361WQ
6	Certification and Training	Certific	CRT283817
7	Certification and Training	Certific	CRT283817
8	Web Development	WebDevel	WBD188377
9	Web Development	WebDevel	WBD188377
10	Software	Software	SFT3521P8
11	Software	Software	SFT3521P8
12	Software	Software	SFT3521P8
13	Networking	Networki	NET3UD697
14	Networking	Networki	NET3UD697
15	Operating Systems	Operatin	OPSI18375

**Example 9.2: Retrieving Macro Variable Values and Creating Numeric Data Set Variables with SYMGET**

Example 9.2 directly references two macro variables with the SYMGET 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, SAS determines 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.

```
%let webfctr=1.20;
%let sftfctr=1.35;
data temp;
  set books.ytdsales(where=
    section in ('Web Development', 'Software'));
  if section='Web Development' then
    costfctr=input(symget('webfctr'),best8.);
  else if section='Software' then
    costfctr=input(symget('sftfctr'),best8.);
  newprice=costfctr*cost;
run;
proc print data=temp;
  title "Prices based on COSTFCTR";
  var booktitle section cost costfctr newprice;
  format newprice dollar8.2;
run;
```

Output 9.2 presents a partial listing of the PROC PRINT report produced by Example 9.2. The output shows that a value was assigned to COSTFCTR depending on the value of SECTION.

#### **Output 9.2 Partial output from Example 9.2**

#### **Prices based on COSTFCTR**

Obs	booktitle	section	cost	costfctr	newprice
1	Software Title 13	Software	\$13.20	1.35	\$17.81
2	Software Title 16	Software	\$13.20	1.35	\$17.81
3	Web Development Title 9	Web Development	\$12.38	1.20	\$14.86
4	Web Development Title 10	Web Development	\$12.38	1.20	\$14.86
5	Software Title 2	Software	\$18.38	1.35	\$24.81

#### **Example 9.3: Using the Resolution of a Character Expression As an Argument to SYMGET**

The DATA step in Example 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, which overrides the default length of 200 bytes.

```
%let managerquarter1=LPL;
%let managerquarter2=EMB;
%let managerquarter3=EMB;
%let managerquarter4=ADL;

data managers;
  set books.ytdsales;

  length managerinit $ 3;

  managerinit=
    symget(cats('managerquarter',put(qtr(datesold),1.)));
run;

proc print data=managers;
  title "Sale Dates and Managers";
  var datesold managerinit;
run;
```

Output 9.3 presents a partial listing of the PROC PRINT report produced by Example 9.3. The output shows the values assigned to MANAGERINITS by SYMGET.

**Output 9.3 Partial output from Example 9.3****Sale Dates and Managers**

<b>Obs</b>	<b>datesold</b>	<b>managerinits</b>
<b>1</b>	10/20/2014	ADL
<b>2</b>	12/02/2014	ADL
<b>3</b>	02/25/2014	LPL
<b>4</b>	01/16/2014	LPL
<b>5</b>	05/18/2014	EMB
<b>6</b>	07/09/2014	EMB
<b>7</b>	10/17/2014	ADL
<b>8</b>	12/16/2014	ADL

**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 SYMPUTX.** The syntax of the CALL SYMPUTX routine is:

```
CALL SYMPUTX(macro-variable,text<,symbol-table>)
```

**CALL SYMPUT.** The syntax of the SYMPUT routine is:

```
CALL SYMPUT(macro-variable,text)
```

The two functions differ in that CALL SYMPUTX trims leading and trailing blanks from the value assigned to the macro variable while CALL SYMPUT does not. Additionally, the CALL SYMPUTX allows you to specify the symbol table in which you want to store the macro variable.

In most situations, SAS recommends CALL SYMPUTX, and the examples in this section use only CALL SYMPUTX. CALL SYMPUTX is a newer function than CALL SYMPUT.

The first two arguments to CALL SYMPUTX and 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 Software. Since CALL SYMPUTX is a SAS language routine, you must enclose literal text arguments in quotation marks.

```
call symputx('booksect', 'Software');
```

- 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 N30. The name of the macro variable, N30, is saved in DATA step character variable RESULTVAR.

```
resultvar='n30';
call symputx(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 AUTHORMNAME. 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 defines a macro variable named AUTHORMNAME5.

```
call symputx(cats('authormname',put(_n_,4.)),
cat('Author Name: ',author));
```

The third argument to CALL SYMPUTX, which tells the macro processor the symbol table where to store the macro variable, is optional. This third 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, which is where the macro processor will store 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 have only 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 SYMPUTX Once at the End of a DATA Step**

In Example 9.4, CALL SYMPUTX creates macro variable N30 whose value is the total number of books that sold for at least \$30.00. The program then places this tally in the title of a PROC MEANS report.

The program directs that CALL SYMPUTX execute once when the DATA step reaches the end of the data set and that it store the formatted value of data set variable NHIGH in macro variable N30. The PUT function formats the value assigned to N30 with the COMMA format.

If the DATA step was written so that CALL SYMPUTX 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 that sold for more than \$30.00, it is necessary to execute CALL SYMPUTX only *once* after the tally is complete. The second IF statement directs that the CALL SYMPUTX routine execute only when the DATA step reaches the end of data set BOOKS.YTDSALES.

```
data _null_;
  set books.ytdsales end=eof;
  if saleprice ge 30 then nhigh+1;
  if eof then call symputx('n30',put(nhigh,comma.));
run;
proc means data=books.ytdsales n mean min max sum maxdec=2;
  title "All Books Sold";
  title2 "Number of Books Sold for More Than $30: &n30";
  var saleprice;
run;
```

Output 9.4 presents the PROC MEANS report. The title includes the total number of books that sold for at least \$30.00.

**Output 9.4 Output from Example 9.4**

**All Books Sold**  
**Number of Books Sold for More Than \$30: 1,326**

**The MEANS Procedure**

Analysis Variable : saleprice Sale Price				
N	Mean	Minimum	Maximum	Sum
3346	28.44	17.59	39.06	95145.40

If you used CALL SYMPUT instead of CALL SYMPUTX, you might need to do additional processing to remove leading and trailing blanks. 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. Depending on the output destination of your report, you may have leading blanks before the value.

Leading blanks are removed when the output is sent to the HTML destination. If the output goes to the PDF or LISTING destinations, leading blanks are not removed.

Consider what happens when the CALL SYMPUT routine is used instead of CALL SYMPUTX, the PUT function is removed, and the destination is PDF.

```
if eof then call symput('n30',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 \$30: \_\_\_\_\_ 1326

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).

**Example 9.5: Executing CALL SYMPUTX Multiple Times in a DATA Step**

In Example 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.

```
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 SYMPUTX**

CALL SYMPUTX creates multiple macro variables in this example. The DATA\_NULL\_ step creates two macro variables for each section in the output data set PROC FREQ created. 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.

```
proc freq data=books.ytdsales noprint;
  tables section / out=sectname;
run;
data _null_;
  set sectname;
  call symputx(cats('name',put(_n_,1.)),section);
  call symputx(cats('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.

```
GLOBAL NAME1 Certification and Training
GLOBAL N1 524
GLOBAL NAME2 Networking
GLOBAL N2 479
```

```

GLOBAL NAME3 Operating Systems
GLOBAL N3 578
GLOBAL NAME4 Programming
GLOBAL N4 337
GLOBAL NAME5 Software
GLOBAL N5 857
GLOBAL NAME6 Web Development
GLOBAL N6 571

```

### **Example 9.7: Creating a Macro Variable with CALL SYMPUTX and Specifying Its Symbol Table**

Example 9.7 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 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 processor store the macro variables in the global symbol table, the macro processor stores them in the local symbol table defined by macro program STATSECTION. Once STATSECTION completed processing, its local symbol table is deleted, and the values of the three macro variables are lost.

Macro program STATSECTION includes a %PUT \_LOCAL\_ statement 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.

```

%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(Software)

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 Software
```

After executing %STATSECTION and the subsequent TITLE statements, the titles become:

```
Section Results for Average Sale Price: $34.28
Minimum Sale Price: $28.04
Maximum Sale Price: $39.06
```

If you dropped the third argument in the three CALL SYMPUTX calls, the macro processor stores the three macro variables in the STATSECTION local symbol table. Then 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 $28.04
STATSECTION MAX $39.06
STATSECTION SECTION Software
STATSECTION AVERAGE $34.28
```

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 that if you want to try out the code that does not specify that the macro processor save the macro variables in the global symbol table and you 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 in a DATA step takes as its argument a character expression or constant text that SAS resolves to a macro program invocation or SAS statements. With CALL EXECUTE, you can conditionally call a macro program using SAS language logic statements such as IF-THEN-ELSE. Additionally, you can assign a DATA step variable's value as a parameter to the macro program that CALL EXECUTE references.

The argument that you supply to CALL EXECUTE does not only have to resolve to a macro program reference. As mentioned above, the argument to CALL EXECUTE can be SAS language statements. For detailed information on how to use CALL EXECUTE in this context, 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. However, if that macro program generates a DATA step or PROC step, those steps 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**

Example 9.8 is a simple example that demonstrates how a macro program invoked by CALL EXECUTE executes immediately within the DATA step. The 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.

```
%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 results of the %PUT \_AUTOMATIC\_ statement appear before the notes that indicate the DATA step has ended. SAS informs you when your CALL EXECUTE does not generate any SAS language statements, as shown by the last note.

```
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');
56 run;
**** Start list of automatic macro variables
AUTOMATIC AFDSID 0
AUTOMATIC AFDSNAME
AUTOMATIC AFLIB
```

```

AUTOMATIC AFSTR1
AUTOMATIC AFSTR2
AUTOMATIC FSPBDV
AUTOMATIC SYSBUFFR
.
.
.
AUTOMATIC SYSTIME 08:20
AUTOMATIC SYSUSERID My Userid
AUTOMATIC SYSVER 9.4
.
.
.
***** End list of automatic macro variables
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**

Example 9.9 demonstrates that macro statements execute during execution of a DATA step and that SAS statements generated by the macro program execute *after* the DATA step finishes. It defines and invokes macro program LISTLIBRARY that submits two %PUT statements and a PROC DATASETS step. Two SAS language PUT statements illustrate the timing of SAS language statements within the DATA step that submits CALL EXECUTE.

The macro program LISTLIBRARY issues a PROC DATASETS for the BOOKS library. Within LISTLIBRARY, a %PUT precedes the PROC step code, and a second %PUT statement follows the PROC step code. Within the DATA step, one PUT statement precedes and a second PUT statement follows CALL EXECUTE.

```

%macro listlibrary;
  %put
  **** This macro statement in LISTLIBRARY precedes the PROC step code. ;
  proc datasets library=books;
    run;
    quit;
  %put
  **** This macro statement in LISTLIBRARY follows the PROC step code. ;
%mend listlibrary;
data _null_;
  put "This SAS language statement precedes the macro program call." ;
  call execute('%listlibrary') ;
  put "This SAS language statement follows the macro program call." ;
run;

```

The SAS log for this program shows that the %PUT statements execute during execution of the DATA step as do the SAS language PUT statements. 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.

```

140  %macro listlibrary;
141      %put **** This macro statement in LISTLIBRARY precedes the PROC
step;
142      proc datasets library=books;
143      run;
144      quit;
145      %put **** This macro statement in LISTLIBRARY follows the PROC
step;
146  %mend listlibrary;
147
148  data _null_;
149      put "This SAS language statement precedes the macro program
call.";
150      call execute('%listlibrary');
151      put "This SAS language statement follows the macro program call.";
152  run;

This SAS language statement precedes the macro program call.
**** This macro statement in LISTLIBRARY precedes the PROC step
**** This macro statement in LISTLIBRARY follows the PROC step
This SAS language statement follows the macro program call.
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

      #   Name       Member      File
          Type        Size  Last Modified

```

```

1   YTDSALES  DATA      771072  02Feb15:16:21:13
1   +
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**

Example 9.10 uses CALL EXECUTE to conditionally execute the macro program REP16K. 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 \$16,000, the CALL EXECUTE statement in the DATA step calls macro program REP16K.

The argument to CALL EXECUTE is the macro program name, REP16K. This macro program has one parameter, SECTION. The argument to CALL EXECUTE is a text expression that resolves to the call to REP16K 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 REP16K 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, total sales exceed \$16,000 in two sections, “Certification and Training” and “Software”. Thus, two PROC REPORT steps execute after the DATA step.

```

%macro rep16k(section);
  proc report data=books.ytdsales center nowd;
    where section="&section";
    title "Sales > $16,000 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;
  run;
%mend rep16k;

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 > 160000 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.

```

222 %macro rep16k(section);
223   proc report data=books.ytdsales nowd;
224     where section="&section";
225     title "Sales > $16,000 Summary for &section";
226     column publisher n saleprice;
227     define publisher / group;
228     define n / "Number of Books Sold" ;
229     define saleprice / sum format=dollar10.2 "Sale Price" ;
230     rbreak after / summarize;
231     compute after;
232       publisher="**Total";
233     endcomp;
234   run;
235 %mend rep16k;
236
237 options mprint;
238
239 proc means data=books.ytdsales nway noprint;
240   class section;
241   var saleprice;
242   output out=sectsale sum=totlsale;
243 run;

```

NOTE: There were 3346 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	0.04 seconds
cpu time	0.00 seconds

```

244
245 data _null_;
246   set sectsale;
247
248 if totlsale > 16000 then

```

```

249      call execute(cats('%rep16k(',section,')'));
250 run;

MPRINT(REP16K): proc report data=books.ytdsales nowd;
MPRINT(REP16K): where section="Certification and Training";
MPRINT(REP16K): title "Sales > $16,000 Summary for Certification and
Training";
MPRINT(REP16K): column publisher n saleprice;
MPRINT(REP16K): define publisher / group;
MPRINT(REP16K): define n / "Number of Books Sold" ;
MPRINT(REP16K): define saleprice / sum format=dollar10.2 "Sale
Price" ;
MPRINT(REP16K): rbreak after / summarize;
MPRINT(REP16K): compute after;
MPRINT(REP16K): publisher="**Total";
MPRINT(REP16K): endcomp;
MPRINT(REP16K): run;
MPRINT(REP16K): proc report data=books.ytdsales nowd;
MPRINT(REP16K): where section="Software";
MPRINT(REP16K): title "Sales > $16,000 Summary for Software";
MPRINT(REP16K): column publisher n saleprice;
MPRINT(REP16K): define publisher / group;
MPRINT(REP16K): define n / "Number of Books Sold" ;
MPRINT(REP16K): define saleprice / sum format=dollar10.2 "Sale
Price" ;
MPRINT(REP16K): rbreak after / summarize;
MPRINT(REP16K): compute after;
MPRINT(REP16K): publisher="**Total";
MPRINT(REP16K): endcomp;
MPRINT(REP16K): run;
NOTE: There were 6 observations read from the data set WORK.SECTSALE.
NOTE: DATA statement used (Total process time):
      real time          0.02 seconds
      cpu time           0.01 seconds

NOTE: CALL EXECUTE generated line.
1 + proc report data=books.ytdsales nowd;      where
section="Certification and Training";
      title "Sales > $16,000 Summary for Certification and Training";
      column publisher n
      saleprice;      define publisher / group;      define n / "Number of
Books
2 + Sold" ;      define saleprice / sum format=dollar10.2 "Sale
Price" ;      rbreak after /
      summarize;      compute after;      publisher="**Total";      endcomp;
run;

NOTE: There were 524 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           0.01 seconds

3  + proc report data=books.ytdsales nowd;      where
section="Software";      title "Sales >
$16,000 Summary for Software";      column publisher n saleprice;
define publisher /
group;      define n / "Number of Books Sold" ;      define saleprice
/ sum
4  + format=dollar10.2 "Sale Price" ;      rbreak after / summarize;
compute after;
publisher="**Total";      endcomp;      run;

NOTE: There were 857 observations read from the data set
      BOOKS.YTDSALES.
      WHERE section='Software';
NOTE: PROCEDURE REPORT used (Total process time):
      real time           0.04 seconds
      cpu time           0.01 seconds

```

### **Example 9.11: Using CALL EXECUTE to Call a Specific Macro Program**

Example 9.11 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.

Example 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 \$20,000, then CALL EXECUTE specifies a call to macro program HIGHREPORT. If sales are less than \$12,000, then CALL EXECUTE specifies a call to macro program LOWREPORT. Neither macro program is called if sales are between \$12,000 and \$20,000.

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, section “Software” exceeds total sales of \$20,000. The DATA step calls macro program HIGHREPORT once.

Two sections, “Networking” and “Programming,” have total sales less than \$12,000. The DATA step calls macro program LOWREPORT twice, once for each of these sections.

```
%macro highreport(section);
  proc report data=books.ytdsales nowd;
    where section=&section";
    title "Sales > $20,000 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;
    compute after;
      publisher="**Total";
    endcomp;
  run;
%mend highreport;

%macro lowreport(section);
  proc report data=books.ytdsales nowd;
    where section=&section";
    title "Sales < $12,000 Report for Section &section";
    column datesold n saleprice;
    define datesold / group format=monname15. "Month Sold"
      style(column)=[just=left];
    define n / "Number of Books Sold";
    define saleprice / sum format=dollar10.2 "Sales Total";
    rbreak after / summarize;
  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 < 12000 then
    call execute(cats('%lowreport(',section,')'));
  else if totlsect > 20000 then
    call execute(cats('%highreport(',section,')'));
run;
```

The SAS log for this program shows the three calls to the two macro programs. Compared to Example 9.10, MPRINT is not in effect when Example 9.11 executes. The results of the CALL EXECUTE call, however, are displayed.

```
951 %macro highreport(section);
952   proc report data=books.ytdsales nowd;
953     where section=&section";
954     title "Sales > $20,000 Report for Section &section";
```

```

955      column publisher n saleprice;
956      define publisher / group;
957      define n / "Number of Books Sold" ;
958      define saleprice / sum format=dollar10.2 "Sale Price" ;
959      rbreak after / summarize;
960      compute after;
961      publisher="**Total";
962      endcomp;
963      run;
964 %mend highreport;
965
966 %macro lowreport(section);
967   proc report data=books.ytdsales nowd;
968     where section=&section";
969     title "Sales < $12,000 Report for Section &section";
970     column datesold n saleprice;
971     define datesold / group format=monname15. "Month Sold"
972           style(column)=[just=left];
973     define n / "Number of Books Sold";
974     define saleprice / sum format=dollar10.2 "Sales Total";
975     rbreak after / summarize;
976     run;
977 %mend lowreport;
978
979 proc means data=books.ytdsales nway noprint;
980   class section;
981   var saleprice;
982   output out=sectsale sum=totlsect;
983 run;

NOTE: There were 3346 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            0.04 seconds
      cpu time             0.01 seconds

984
985 data _null_;
986   set sectsale;
987   if totlsect < 12000 then
988     call execute(cats('%lowreport(',section,')'));
989
990   else if totlsect > 20000 then
991     call execute(cats('%highreport(',section,')'));
992 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	0.01 seconds

```

NOTE: CALL EXECUTE generated line.
1 + proc report data=books.ytdsales nowd;      where
  section="Networking";      title "Sales
< $12,000 Report for Section Networking";      column datesold n
  saleprice;      define
  datesold / group format=monname15. "Month Sold"
  style
2 +(column)=[just=left];      define n / "Number of Books Sold";
  define saleprice / sum
  format=dollar10.2 "Sales Total";      rbreak after / summarize;
run;
NOTE: There were 479 observations read from the data set
      BOOKS.YTDSALES.
      WHERE section='Networking';
NOTE: PROCEDURE REPORT used (Total process time):
      real time            0.03 seconds
      cpu time             0.01 seconds

3 + proc report data=books.ytdsales nowd;      where
  section="Programming";      title
  "Sales < $12,000 Report for Section Programming";      column datesold
  n saleprice;
  define datesold / group format=monname15. "Month Sold"
4 + style(column)=[just=left];      define n / "Number of Books
  Sold";      define saleprice
  / sum format=dollar10.2 "Sales Total";      rbreak after / summarize;
run;
NOTE: There were 337 observations read from the data set
      BOOKS.YTDSALES.
      WHERE section='Programming';
NOTE: PROCEDURE REPORT used (Total process time):
      real time            0.03 seconds
      cpu time             0.00 seconds

5 + proc report data=books.ytdsales nowd;      where
  section="Software";      title "Sales >
$20,000 Report for Section Software";      column publisher n
  saleprice;      define
  publisher / group;      define n / "Number of Books Sold" ;
  define
6 + saleprice / sum format=dollar10.2 "Sale Price" ;      rbreak
  after / summarize;
  compute after;      publisher="**Total";      endcomp;      run;

NOTE: There were 857 observations read from the data set
      BOOKS.YTDSALES.
      WHERE section='Software';
NOTE: PROCEDURE REPORT used (Total process time):
      real time            0.03 seconds
      cpu time             0.00 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 attempts 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 have a different length, 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 Example 9.12 was presented above in the discussion on the types of arguments that RESOLVE can accept. The goal of Example 9.12 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.

Example 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.

Output 9.5 shows the results of the PROC FREQ crosstabulation of QUARTER and QUARTERSALENAME.

```
%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 Example 9.12.

#### Output 9.5 Output from Example 9.12

## Quarter by Quarter Sale Name

### The FREQ Procedure

<b>quarter</b>	<b>quartersalename</b>	<b>Frequency</b>
1	Holiday Clearance	796
2	2 for the Price of 1	860
3	Back to School	822
4	New Releases	868

### **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 and it returns text to the DATA step.

As in Example 9.12, Example 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 Example 9.12, the argument to the RESOLVE function in Example 9.13 is constructed during execution of the DATA step, and the argument is enclosed in single quotation marks, which prevent the macro processor from attempting to resolve the call to the macro program during compilation of the DATA step.

```
%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 Example 9.13.

#### Output 9.6 Output from Example 9.13

### Quarter Sales with Quarter Sale Name

Obs	Quarter	Total Sales	Sale Name
1	1	\$22,854.27	Holiday Clearance
2	2	\$24,293.13	2 for the Price of 1
3	3	\$23,401.99	Back to School
4	4	\$24,596.01	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 macro variables that hold information about the processing of the PROC SQL step. 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 can create and update 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.

In general, SAS adds the macro variables that you create with PROC SQL 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 :macro-variable-specification
    <, . . . macro-variable-specification>
  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. The macro variable specification can be written as follows.

- If you want to store the first value returned into a single macro variable, use this syntax. Leading and trailing blanks are preserved unless the TRIMMED option is specified.  
`:macro-variable <TRIMMED>`
- If you want to store a series of values that are returned in a *single* macro variable, use this syntax. Unless the NOTRIM option is included, SAS removes leading and trailing blanks from the values saved in the macro variable. The SEPARATED BY option is required, and it specifies the character(s) that indicate row separation in the value of *macro-variable*.  
`:macro-variable <SEPARATED BY 'character(s)'><NOTRIM>`
- If you want to store the values that the SELECT statement returns in a series of macro variables, use this syntax. The value N does not have to be exactly equal to the number of values returned. If N is greater than the number of values returned, SAS creates the number of macro variables equal to the number returned. If N is less than the number of values returned,

SAS creates only N macro variables. Unless the NOTRIM option is included, SAS removes leading and trailing blanks from the values saved in the macro variables.

```
:macro-variable-1 - macro-variable-n <NOTRIM>
```

- Another way of specifying a series of macro variables follows. The upper limit of the series is not specified. In this situation, SAS creates the number of macro variables equal to the number of values returned. Unless the NOTRIM option is included, SAS removes leading and trailing blanks from the values saved in the macro variables.

```
:macro-variable-1 - <NOTRIM>
```

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**

Example 9.14 presents a simple application of the INTO clause. 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.

```
%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;
%put &findpublisher 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;
36  %put &findpublisher Total Number Sold=&nsold;
Technology Smith Total Sales= $5,503.41
Technology Smith Total Number Sold= 238

```

Notice the leading blanks before the resolved macro variable values on the results of the two %PUT statements. If you add the TRIMMED option to each macro variable reference on the INTO clause, SAS removes the leading blanks.

```
into :totsales trimmed, :nsold trimmed
```

When the INTO clause includes the two TRIMMED options, the results of the %PUT statements are as follows.

```

Technology Smith Total Sales=$5,503.41
Technology Smith Total Number Sold=238

```

### **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 named on the INTO clause. This example demonstrates that action.

Example 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 FIRSTDATE, FIRSTTITLE, and FIRSTPRICE, 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.

```

proc sort data=books.ytdsales out=datesorted;
  by datesold;
run;
proc sql noprint;
  select datesold,booktitle,saleprice
    into :firstdate,:firsttitle,:firstprice
    from datesorted;
quit;

%put One of the first books sold was on &firstdate;
%put The title of this book is &firsttitle;
%put The sale price was &firstprice;

```

```
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.

```
6611  proc sql noprint;
6612    select datesold,booktitle,saleprice
6613      into :firstdate,:firsttitle,:firstprice trimmed
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 &firstdate;
One of the first books sold was on 01/01/2014
6617  %put The title of this book is &firsttitle;
The title of this book is Web Development Title 9
6618  %put The sale price was &firstprice;
The sale price was $23.68
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
```

Output 9.7 shows the PROC PRINT report that verifies the values of the macro variables created by the PROC SQL step.

#### Output 9.7 Output from Example 9.15

First Five Observations of Sorted by Date BOOKS.YTDSALES

Obs	section	booktitle	author	publisher	bookfmt	cost	listprice	saleprice	saleid	datesold
1	Web Development	Web Development Title 9	Hayes, Jacob	Professional House Titles	paper	\$12.38	\$30.95	\$23.68	10003102	01/01/2014
2	Web Development	Web Development Title 10	Gonzales, Angela	Northern Associates Titles	paper	\$12.38	\$30.95	\$23.68	10003177	01/01/2014
3	Certification and Training	Certification and Training Title 9	Reynolds, Gary	AMZ Publishers	ebook	\$12.00	\$29.99	\$25.49	10002728	01/01/2014
4	Networking	Networking Title 1	Bennett, Thomas	Mainst Media	paper	\$13.98	\$34.95	\$26.74	10000876	01/01/2014
5	Networking	Networking Title 9	Roberts, Jacob	Bookstore Brand Titles	paper	\$13.98	\$34.95	\$26.74	10001291	01/01/2014

#### 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 Example 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.

In this example, it is known that BOOKS.YTDSALES has six section values. Example 9.19 shows a way to write the INTO clause if you do not know the number of values the SELECT clause will return.

```
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 $16,227.32
96  %put *** 2: &section2 &sale2;
*** 2: Networking $11,353.04
97  %put *** 3: &section3 &sale3;
*** 3: Operating Systems $15,963.34
98  %put *** 4: &section4 &sale4;
*** 4: Programming $10,160.95
99  %put *** 5: &section5 &sale5;
*** 5: Software $29,374.98
100 %put *** 6: &section6 &sale6;
*** 6: Web Development $12,065.77
```

### **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 Example 9.17 stores all unique section names in the macro variable ALLSECT.

```

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/Networking/Operating Systems/Programming/Software/Web
Development
```

### **Example 9.18: Storing All Values of a PROC SQL Dictionary Table Column in One Macro Variable**

Example 9.18 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. Example 9.18 makes use of the DICTIONARY tables feature of PROC SQL. It saves in one macro variable DATASETNAMES the names of all the SAS data sets in a library specified by the value of macro variable LISTLIB. A

blank separates the data set names assigned to DATASETNAMES. Assume there are three SAS data sets in library BOOKS: YTDSALES, SALES2013, and SALES2012.

```
%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)
SALES2012 SALES2013 YTDSALES
```

## Using the Macro Variables Created by PROC SQL

PROC SQL creates and updates four 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 four PROC SQL macro variables.

**Table 9.2 Macro variables created by PROC SQL**

Macro Variable	Description
SQLEXITCODE	Contains the highest return code that occurred from some types of SQL insert failures. This return code is written to the SYSERR macro variable when PROC SQL terminates.
SQLOBS	Set to the number of rows produced with a SELECT statement
SQLRC	Set to the return code from an SQL statement
SQLLOOPS	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, SQLXMSG and SQLXRC. These macro variables contain information about error conditions that might have occurred in the processing of pass-through facility 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
SQLXMSG	Set to descriptive information and DBMS-specific return code generated by a pass-through facility SQL statement
SQLXRC	Set to the return code generated by a pass-through facility SQL statement

**Example 9.19: Using the PROC SQL SQLOBS Automatic Macro Variable**

Macro program LISTSQLPUB in Example 9.19 uses the SQLOBS macro variable to list the values of a series of macro variables that were created by a SELECT statement and the INTO clause. The goal of the program is to save the names of the publishers in BOOKS.YTDSALES in a series of macro variables.

The INTO clause contains only the reference to the first macro variable in the PUB series of macro variables. This reference is followed by a dash (-). This structure tells SAS to create a macro variable for each value that the SELECT clause returns. The number of publishers in BOOKS.YTDSALES is 12. The SELECT statement returns twelve values and saves the values in macro variables PUB1-PUB12.

After the SELECT statement executes, PROC SQL updates the SQLOBS macro variable. The iterative %DO loop uses this value as its upper index value. The %PUT statement in the iterative %DO loop lists each publisher's name.

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 you execute code that references that original SQLOBS value.

```

options mprint;

%macro listsqlpub;
  proc sql noprint;
    select unique(publisher)
      into :pub1 -
      from books.ytdsales;
  quit;

  %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) into :pub1 -
from books.ytdsales;
MPRINT(LISTSQLPUB): quit;
NOTE: PROCEDURE SQL used (Total process time):
      real time          0.04 seconds
      cpu time           0.04 seconds

```

**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

```

If you specify an upper limit on the series of macro variables on the INTO clause that is *less* than the number of values that the SELECT clause returns, SAS still executes the step. In this situation, SAS creates only the number of macro variables that are specified, which is 2, and assigns the first values it returns to those macro variables.

```

MPRINT(LISTSQLPUB): proc sql noprint;
MPRINT(LISTSQLPUB): select unique(publisher) into :pub1 - :pub2 from
books.ytdsales;
MPRINT(LISTSQLPUB): quit;
NOTE: PROCEDURE SQL used (Total process time):
      real time          0.03 seconds
      cpu time           0.03 seconds

Total number of publishers: 2.
Publisher 1: AMZ Publishers
Publisher 2: Bookstore Brand Titles

```

If you specify an upper limit on the series of macro variables INTO clause that is *greater* than the number of values that the SELECT clause returns, SAS creates only the number of macro variables equal to the number of values that the SELECT clause returns. This SAS log shows that the upper limit is set to 100.

```

MPRINT(LISTSQLPUB): options symbolgen;
MPRINT(LISTSQLPUB): proc sql noprint;

```

```

MPRINT(LISTSQLPUB): select unique(publisher) into :pub1 - :pub100
from books.ytdsales;
MPRINT(LISTSQLPUB): quit;
NOTE: PROCEDURE SQL used (Total process time):
      real time           0.03 seconds
      cpu time            0.00 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

```

A %PUT statement verifies that there is no PUB13 macro variable.

```

186 %put &pub13;
WARNING: Apparent symbolic reference PUB13 not resolved.
&pub13

```

---

## 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. Example 9.18 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 to control 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.

```
proc sql;
  select * from dictionary.options
  where group='MACRO';
quit;
```

The PROC SQL step that follows saves the setting of the macro option MINDELIMITER in a macro variable MYSETTING.

```
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 #
```

The PROC SQL step illustrates a simple example in working with the MACRO group of the dictionary tables. However, it might 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.

```
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_;
```



## **Part 2: Applying Your Knowledge of Macro Programming**

<b>Chapter 10</b>	<b>Storing and Reusing Macro Programs</b>	<b>247</b>
<b>Chapter 11</b>	<b>Building a Library of Utilities</b>	<b>261</b>
<b>Chapter 12</b>	<b>Debugging Macro Programming and Adding Error Checking to Macro Programs</b>	<b>273</b>
<b>Chapter 13</b>	<b>A Stepwise Method for Writing Macro Programs</b>	
	<b>307</b>	



# **Chapter 10 Storing and Reusing Macro Programs**

<b>Introduction.....</b>	<b>247</b>
<b>Saving Macro Programs with the Autocall Facility .....</b>	<b>248</b>
Creating an Autocall Library .....	248
Making Autocall Libraries Available to Your Programs .....	250
Maintaining Access to the Autocall Macro Programs That Ship with SAS .....	251
Defining Filerefs under Windows and Using Them to Identify Autocall Libraries ..	251
Explicitly Specifying the Directory Locations of Autocall Libraries on the OPTIONS Statement.....	252
Identifying Autocall Libraries That Are Stored in SAS Catalogs .....	252
Listing the Names of the Autocall Libraries That Are Defined in the SAS Session	252
Using the Autocall Facility under UNIX and z/OS Systems .....	252
Using the Autocall Facility under UNIX .....	252
Using the Autocall Facility under MVS Batch.....	253
<b>Saving Macro Programs with the Stored Compiled Macro Facility.....</b>	<b>254</b>
Setting SAS Options to Create Stored Compiled Macro Programs .....	254
Creating Stored Compiled Macro Programs.....	255
Saving and Retrieving the Source Code of a Stored Compiled Macro Program ..	257
Encrypting a Stored Compiled Macro Program.....	258
<b>Resolving Macro Program References When Using the Autocall Facility and the Stored Compiled Macro Facility .....</b>	<b>258</b>

---

## **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 resolves 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 resolves 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.

An advantage of the autocall facility is that the macro programs are portable across all operating systems and releases of SAS.

A minor disadvantage to the autocall facility is that the macro program must be searched for and compiled the first time it is used in a SAS session. This takes resources, although usually it is minimal.

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 execute 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.SASMCR is deleted. The code remains in the autocall library unless you delete the macro program or 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 or UNIX, 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, macro program name, and SAS extension must be in lowercase.)

Under z/OS, 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.

Figure 10.1 shows an example of an autocall library where the four macro programs are stored as separate files.

**Figure 10.1 A Windows 7 directory containing four autocall macro programs**

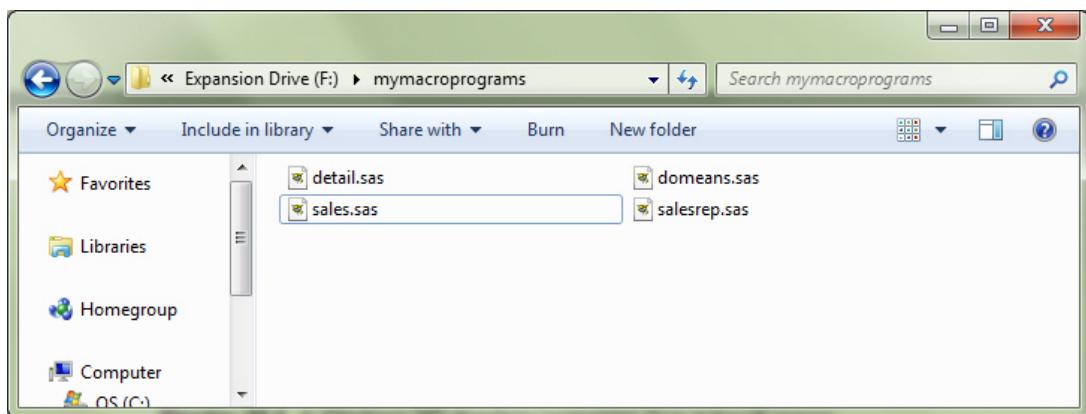
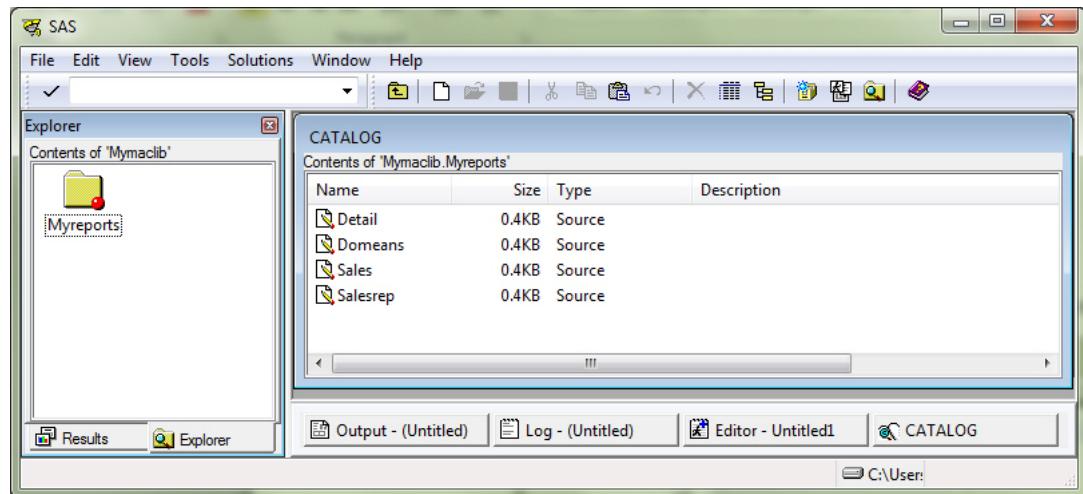


Figure 10.2 shows a SAS catalog that contains the four macro programs stored as SOURCE entries.

**Figure 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.

Additionally, you can reference them in an AUTOEXEC file or modify the config file so that the autocall library is available when your SAS session starts. For more information, see Documentation in the Knowledge Base, at <http://support.sas.com>.

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. While not recommended, a reason you might disable MAUTOSOURCE is if you want to save computing resources when you do not intend to use 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 fileref 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 you list them 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. The SASAUTOS fileref is automatically created by SAS when your SAS session starts.

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 and Using Them to Identify Autocall Libraries

The next statements define two filerefs under Windows 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:\mymacropograms\repmacs' 'c:\mymacropograms\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 in 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 libraries are defined in the current SAS session, submit the following PROC step.

```
proc options option=sasautos;
run;
```

## Using the Autocall Facility under UNIX and z/OS 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.

### Using the Autocall Facility under UNIX

As with Windows, 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 '/mymacropograms/reports'  
-append sasautos sasautos
```

The next example specifies two user-defined autocall libraries and includes the SASAUTOS reference.

```
sas -mautosource -sasautos '/mymacropograms/graphmacs'  
-append sasautos '/mymacropograms/reports'  
-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=('/mymacropograms/reports',  
sasautos)
```

The next OPTIONS statement specifies two user-defined autocall libraries from within a UNIX SAS session, and it includes the SASAUTOS reference.

```
options mautosource sasautos=  
('/mymacropograms/reports', '/mymacropograms/graphmacs',  
sasautos);
```

## 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=MYMACS.REPORTS,DISP=SHR
```

The next example shows how multiple macro libraries can be specified.

```
//MYJOB    JOB account....  
//          EXEC SAS,OPTIONS='MAUTOSOURCE'  
//SASAUTOS DD   DSN=MYMACS.REPORTS,DISP=SHR  
//          DD   DSN=MYMACS.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='mymacs.reports' sasautos;
```

The following statement specifies two user-defined autocall libraries plus the SASAUTOS fileref.

```
options mautosource sasautos= ('mymacs.reports' 'mymacs.grphtmacs'  
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 compilation 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 must maintain the macro source code separately. You will not be able to retrieve the code from the compiled version of the macro program.

A convenient place to store your macro program code is in 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 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 when you want to store a compiled macro program in a catalog. 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.

The SECURE option encrypts the compiled macro program and makes it more difficult for someone to obtain portions of the source code. The %COPY statement will not work with secure macro programs. However, it may be possible to see the non-macro code if the catalog entry is viewed within a text editor.

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**

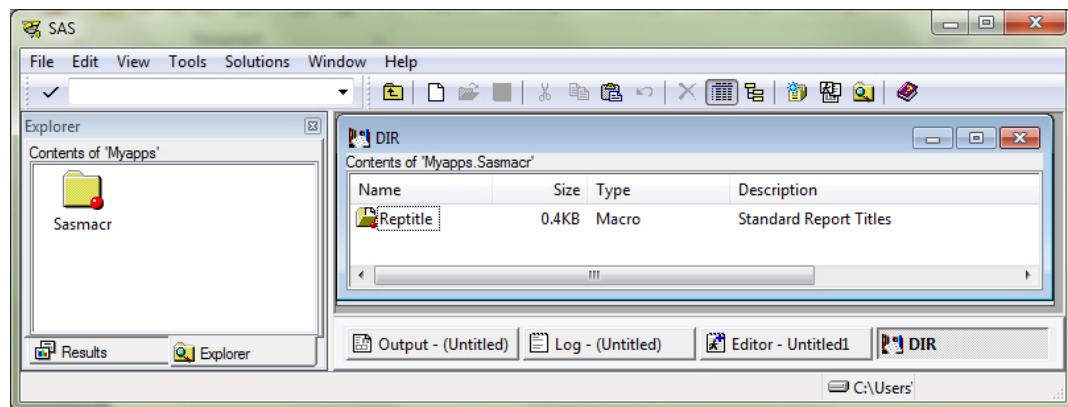
An example of defining a macro program and storing it in a catalog under Windows in SAS®9 follows:

```
libname myapps 'c:\mymacroprograms';
options mstored sasmstore=myapps;

%macro reptile(repprog) / store des='Standard Report Titles';
  title "Bookstore Report &repprog";
  title2 "Processing Date: &sysdate SAS Version: &sysver";
%mend reptile;
```

Figure 10.3 shows the DIR window for the MYAPPS.SASMACR catalog after submitting this program.

**Figure 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 statement has three options.

```
%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**

Example 10.1 is modified in Example 10.2 to save the macro program code along with the compiled macro program.

```
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

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**

Example 10.2 is modified in Example 10.3 so that the stored compiled macro program is secure and encrypted.

```
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;
```

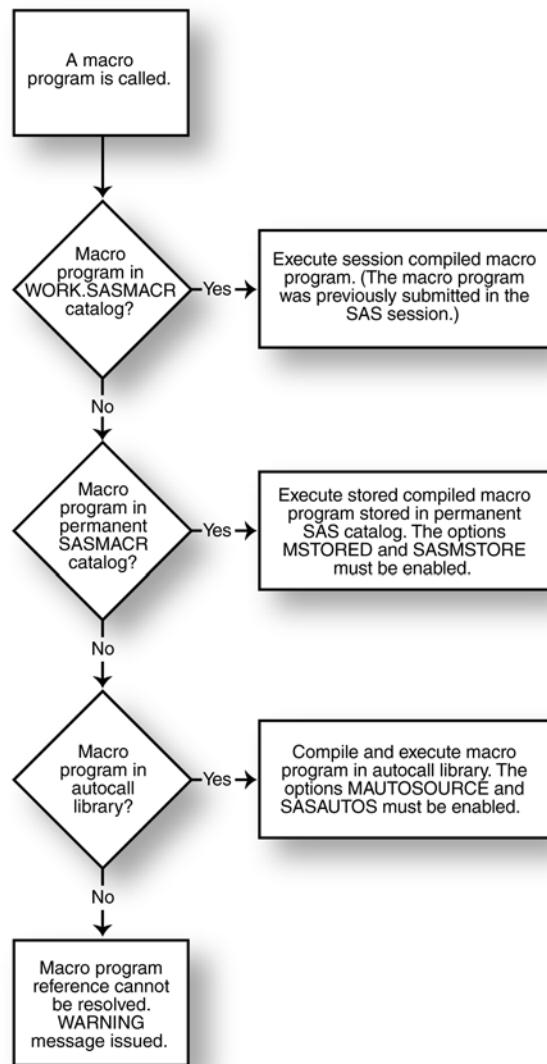
---

## 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. With these features, 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 10.4 to resolve a macro program reference.

**Figure 10.4 How the macro processor resolves calls to macro programs**



# **Chapter 11 Building a Library of Utilities**

<b>Introduction .....</b>	<b>261</b>
<b>Writing a Macro Program to Behave Like a Function.....</b>	<b>261</b>
<b>Programming Routine Tasks .....</b>	<b>266</b>

---

## **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 support website, SAS Community 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 Examples 11.1 and 11.2 demonstrate.

### **Example 11.1: Examining Specific Data Set Characteristics**

Example 11.1 demonstrates that you can call a macro program and have it return a value that you can test to determine the next processing step to take. This action is similar to the way you reference a function and obtain a return code that can be tested. The example is illustrated in two steps. The first step shows how a function can be tested. The second step replaces the function call with a macro program call.

The SAS language function EXIST determines if a data set exists, and it 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. Macro program LISTSAMPLE 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 an error message to the SAS log that the data set does not exist.

```
%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)
```

The program is modified below to replace the EXIST function with a call to macro program MULTCOND. Macro program MULTCOND checks four conditions of a data set, including whether it exists. MULTCOND defines a macro variable RC whose values can be 0 or 1 depending on the macro program's evaluations of the data set. In this manner, MULTCOND acts like a function. 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 is to replace %sysfunc(exist) with %multcond(&dsname).

```
%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";
  %end;
```

```

        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 the following error 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.

Example 11.2 submits a PROC TABULATE step that lists sales by quarter and book format of all titles by 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. Macro program TRIMNAME is called twice. The first time it is referenced is on the WHERE statement of the PROC TABULATE step. The resolved value returned by TRIMNAME is supplied to the WHERE statement. The second reference 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 %QUPCASE 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.

```
%macro trimname(namevalue);
  %cmpres(%qupcase(%sysfunc(
    compress(%superq(namevalue),%str(, ),kA))))
%mend trimname;

proc tabulate data=books.ytdsales
  (where=(upcase(author)=
    "%trimname(%str(wright, LINDA))"))
  ;
  title "Title list for %trimname(%str(wright, linda))";
  class booktitle datesold bookfmt;
  tables booktitle=' ',
    datesold="Quarter Sold" all='**Total Books Sold',
    all="Books Sold"*(bookfmt all='Total')*n=' '*f=3. /
      misstext='0';
  format datesold qtr.;
run;
```

Output 11.1 shows the results of the PROC TABULATE step, including the editing of the author's name for insertion in the title.

#### **Output 11.1 Output from Example 11.2**

<b>Title list for WRIGHT, LINDA</b>			
<b>Networking Title 2</b>			
<b>Quarter Sold</b>	<b>Books Sold</b>		<b>Total</b>
	<b>Book Format</b>		
	ebook	paper	
<b>1</b>	3	5	8
<b>2</b>	3	7	10
<b>3</b>	5	8	13
<b>4</b>	4	7	11
<b>**Total Books Sold</b>	<b>15</b>	<b>27</b>	<b>42</b>

## **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**

Example 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 HTML 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 HTML destination
- resets the orientation to PORTRAIT
- turns on SAS option DATE
- clears the TITLE1 and FOOTNOTE1 statements.

Example 11.3 first submits and compiles the two macro programs. 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 found in SASUSER.TMPLMST.

```
%macro rtf_start(style=,orientation=);
  /* This macro program initializes settings to send reports
   to ODS RTF destination;
  ods html 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 html;
```

```
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
```

A copy of the report follows in Output 11.2.

**Output 11.2 Output from Example 11.3**

Bookstore	
Section	Sale Price
Certification and Training	\$16,227.32
Networking	\$11,353.04
Operating Systems	\$15,963.34
Programming	\$10,160.95
Software	\$29,374.98
Web Development	\$12,065.77
<b>** Totals **</b>	<b>\$95,145.40</b>

Report Prepared 09 JAN 2015

**Example 11.4: Documenting Characteristics of a Data Set**

As a programmer, you might frequently want to list the same specific information about a data set in a specific order to document your work. 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 that you can reference when needed.

Macro program FACTS in Example 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 Example 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. 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.

```
%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 html close;
ods pdf style=statistical;
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 html;

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 Example 11.4.

**Output 11.3 Output from Example 11.4****Data Set Report for BOOKS.YTDSALES Sales for 2014**

Attribute	Value
Creation Date and Time	01JAN14:17:16:01
Last Modification Date and Time	02JAN15:12:26:31
Number of Observations	3346
Number of Variables	10
Variables by Position	cost, listprice, saleprice, saleid, datesold, section, booktitle, author, publisher, bookfmt
Variables Alphabetically	author, bookfmt, booktitle, cost, datesold, listprice, publisher, saleid, saleprice, section

**Data Set Report for BOOKS.YTDSALES Sales for 2014****First 5 Observations**

Obs	section	booktitle	author	publisher	bookfmt	cost	listprice	saleprice	saleid	datesold
1	Web Development	Web Development Title 6	Miller, Kimberly	Technology Smith	ebook	\$9.20	\$22.99	\$17.59	10003021	01/03/2014
2	Web Development	Web Development Title 14	Parker, Timothy	Nifty New Books	ebook	\$9.20	\$22.99	\$17.59	10003338	01/04/2014
3	Web Development	Web Development Title 5	Rodriguez, Joseph	AMZ Publishers	ebook	\$9.20	\$22.99	\$17.59	10002913	01/05/2014
4	Web Development	Web Development Title 9	Hayes, Jacob	Professional House Titles	ebook	\$9.20	\$22.99	\$17.59	10003110	01/05/2014
5	Web Development	Web Development Title 9	Hayes, Jacob	Professional House Titles	ebook	\$9.20	\$22.99	\$17.59	10003112	01/08/2014

# **Chapter 12 Debugging Macro Programming and Adding Error Checking to Macro Programs**

<b>Introduction .....</b>	<b>273</b>
<b>Understanding the Types of Errors That Can Occur in Macro Programming ...</b>	<b>273</b>
<b>Minimizing Errors in Developing SAS Programs That Contain Macro</b>	
<b>Language .....</b>	<b>274</b>
<b>Categorizing and Checking for Common Problems in Macro Programming ....</b>	<b>275</b>
<b>Understanding the Tools That Can Debug Macro Programming .....</b>	<b>278</b>
Using SAS System Options to Debug Macro Programming.....	278
Using Macro Language Statements to Debug Macro Programming .....	279
Using Macro Functions to Debug Macro Programming .....	280
Using Automatic Macro Variables to Debug Macro Programming.....	280
<b>Examples of Solving Errors in Macro Programming.....</b>	<b>281</b>
<b>Improving Your Macro Programming by Including Error Checking .....</b>	<b>298</b>

---

## **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 SAS writes messages related to the syntax error 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 you complete that task you then 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 that you define 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 a checklist of items to check in the code**

Category	Items to Check
Punctuation	<p>Do all statements end with a semicolon?</p> <p>Did you terminate your macro program call with a semicolon? If so, does the semicolon interfere with the resolution of the macro program call?</p> <p>Do you have any unmatched parentheses or quotation marks? If so, is it necessary to mask them?</p> <p>Are you using single quotation marks and double quotation marks correctly? For example, did you enclose title text with double quotation marks so that macro variable references in the text are resolved?</p> <p>Have you terminated your macro statement labels with a colon(:)?</p> <p>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 SAS from interpreting the label as a macro program reference.</p>

Category	Items to Check
<b>Macro Variable Resolution</b>	<p>Do your macro variable references start with an ampersand?</p> <p>Do your macro variable references need a delimiter (a period) so that the macro processor can tell where the reference ends?</p> <p>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?</p> <p>Are you referencing a local macro variable outside of the macro program in which it has been defined?</p> <p>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.</p> <p>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 it is referenced in the DATA step or modify the code to use the RESOLVE SAS language function instead.</p> <p>Do any of your macro variables start with AF, DMS, SYS, or SQL? If so, you should rename them to prevent conflicts in names with automatic macro variables.</p>
<b>Macro Program Resolution</b>	<p>Have you submitted the macro program definition before calling the macro program?</p> <p>Do your macro program references start with a percent sign?</p> <p>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?</p>
<b>System Options Settings</b>	<p>Is SAS option MACRO in effect so that you can access the macro facility?</p> <p>Is option MERROR in effect so that the macro processor displays warning messages when a macro program reference cannot be resolved?</p> <p>Is option SERROR in effect so that the macro processor displays warning messages when a macro variable reference cannot be resolved?</p> <p>Have all appropriate options been set (SASAUTOS, MAUTOSOURCE, MSTORED, SASMSTORE) when working with autocall libraries or stored compiled macro programs?</p> <p>Has MINDELIMITER been specified correctly if any of your macro language statements use the IN operator? Has the MINOPERATOR been set?</p>

Category	Items to Check
%MACRO, %MEND Statements	<p>Do the names on the %MACRO and %MEND statements agree?</p> <p>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.</p>
Macro Parameters	<p>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?</p> <p>If you’re using keyword parameters and you’ve specified defaults for them, have you specified the defaults correctly? Do the default values need to be masked with a macro quoting function?</p> <p>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?</p>
Macro Functions	<p>Does your function call have the correct number of arguments?</p> <p>Have you quoted the arguments to the macro function as you would have with the SAS function counterpart? If so, remove the quotation marks.</p> <p>Does your function argument contain special characters such as commas? If so, you may have to mask the argument with macro quoting functions.</p>
SAS Language vs. Macro Language	<p>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.</p> <p>Are you referencing DATA step variables on a %IF statement? If so, SAS interprets the DATA step variable names as text.</p> <p>Have you forgotten percent signs on macro language keywords so that SAS interprets these as SAS language keywords instead?</p>
%DO, %END statements	<p>Do you have a matching %END for each %DO statement?</p> <p>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.</p>
Processing Special Characters and Mnemonic Operators	<p>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.</p> <p>When you are masking an ampersand or percent sign, are you using the “NR” version of the macro quoting function?</p>
Calculations	<p>Are you using %EVAL for integer arithmetic and %SYSEVALF for calculations that require floating-point arithmetic?</p>

Category	Items to Check
<b>Logical Expressions</b>	Can the operands in your expressions contain special characters or mnemonic operators? If so, you may need to mask them 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.

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, see Documentation in the Knowledge Base, at <http://support.sas.com>.

### 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.

The most important options to remember to use when debugging your macro programs are MPRINT, MLOGIC, and SYMBOLGEN. These three options provide you the most information when debugging.

**Table 12.2 SAS system options useful in debugging macro programming**

Option	Purpose
<b>MLOGIC   NOMLOGIC</b>	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.

---

Option	Purpose
<b>MLOGICNEST   NOMLOGICNEST</b>	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.
<b>MPRINT   NOMPRINT</b>	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.
<b>MPRINTNEST   NOMPRINTNEST</b>	Aligns the nesting level of macro programs and the SAS code generated by the execution of the macro programs in the SAS log.
<b>SYMBOLGEN   NOSYMBOLGEN</b>	Displays the resolution of macro variable references in the SAS log.

---

Chapter 3 presented several examples of how the SYMBOLGEN option can display resolution of macro variable references. Examples in Chapter 4 introduced how you can use the MPRINT option to display the SAS code that a macro program generates.

When processing and debugging macro language, it is important to verify that you have enabled the three system options listed in Table 12.3. 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
<b>MACRO   NOMACRO</b>	Controls whether the macro processor is available to recognize and process macro language
<b>MERROR   NOMERROR</b>	Controls whether the macro processor issues a warning message when a macro program reference cannot be resolved
<b>SERROR   NOSERROR</b>	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 %PUT statement is the primary macro language statement to use when you debug your macro programming. 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 using SYMBOLGEN 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 might 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, might be useful if macro variable references are not resolved the way you expect 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 discusses building your own 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. The values assigned to some of these automatic macro variables may be useful 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, 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 the errors 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 Example 12.1 is to list with PROC PRINT the first ten observations in the data set specified by the DSNAME parameter defined by the macro program. Note that the reference to DSNAME in the title text in PRINT10 is misspelled as DSNAMEE. Assume that option SERROR is disabled when the following program executes.

```
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

Example 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 might be able to free your SAS session if you submit 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 complete copy of BOOKS.YTDSALES.

The DATA step creates temporary data set TEMP. It calls macro program WHSTMT. This macro program constructs a WHERE statement that is applied to the SET statement in the DATA step. In Example 12.2, the goal is to create DATA step TEMP with observations selected from the Software section and publisher AMZ Publishers. The error in the macro program definition for WHSTMT is the missing semicolon on the last %END statement.

```
%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;
  %if &getpub ne %then %do;
    publisher="&getpub"))
  %end
%mend whstmt;
data temp;
  set books.ytdsales
    %whstmt(Software,AMZ Publishers)
  ;
run;
```

When you submit the code, the DATA step does not execute as the following SAS log shows. 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 will 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**

Example 12.3 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 the first group of statements 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 this first group of statements compiles without error. The call to MARKUP specified below, however, does not execute as expected.

```

%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 the above statements 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 a %PUT statement after the first %LET statement so that SAS 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 238 observations read from the data set
      BOOKS.YTDSALES.
      WHERE publisher='Technology Smith';
NOTE: The data set WORK.PUBMARKUP has 238 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.

```
%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 238 observations read from the data set
      BOOKS.YTDSALES.
      WHERE publisher='Technology Smith';
NOTE: The data set WORK.PUBMARKUP has 238 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 the first group of statements 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 the loop 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 executes indefinitely because the value extracted by %SCAN will never equal the text ''.

```

%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 &=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   &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.

```

%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 might 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 RTF file and/or a Microsoft Excel workbook of the selected records. To do this, the program defines three macro programs: MAKERTF, MAKESPREADSHEET, and EXTFILES. Macro program MAKERTF lists observations with PROC PRINT and creates an RTF file of the report in the Meadow style. Macro program MAKESPREADSHEET creates a Microsoft Excel workbook with two worksheets. One worksheet contains the publisher's sales records for books in e-book format. The second worksheet contains the publisher's records for books in paper format. Macro program EXTFILES creates the subset data set, computes new variable LOSTPROFIT, and calls MAKERTF and MAKESPREADSHEET when specified to do so by its parameter values.

Macro program EXTFILES has three parameters: PUBLISHER, RTF, and SPREADSHEET. Positional parameter PUBLISHER specifies the publisher name whose observations should be selected from BOOKS.YTDSALES. Keyword parameters RTF and SPREADSHEET are defined to have one of four values in uppercase or lowercase: Y, YES, NO, or N. Specify the values Y or YES when you want the specific output file produced; specify N or NO when you do not. The program starts by converting the values of RTF and SPREADSHEET to uppercase so that the list of values on the IF statements only need to be in uppercase.

The problem with the following 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 RTF file (RTF=Y) and to create an Excel workbook (SPREADSHEET=Y), the %ELSE statement prevents execution of %MAKESPREADSHEET.

The call to EXTFILES specifies that both an RTF and an Excel workbook should be created for publisher Eversons Books.

```
%macro extfiles(publisher,rtf=,spreadsheet=) / minoperator;
  %let rtf=%upcase(&rtf);
  %let spreadsheet=%upcase(&spreadsheet);

  data temp;
    set books.ytdsales(where=(publisher="&publisher"));
    lostprofit=listprice-saleprice;
  run;

  %if &rtf in Y YES %then %do;
    %makertf
  %end;
  %else %if &spreadsheet in Y YES %then %do;
    %makespreadsheet
  %end;
  %mend extfiles;

%macro makertf;
  ods html close;
  ods rtf style=meadow;
  proc print data=temp(drop=saleid publisher);
    title "Publisher: &publisher";
    format saleprice listprice;
  run;
  ods rtf close;
  ods html;
%mend makertf;

%macro makespreadsheet;
  proc export data=temp(where=(bookfmt='ebook')) 
    file="pubreports.xlsx"
    replace;
    sheet="Ebook &publisher";
  run;
  proc export data=temp(where=(bookfmt='paper')) 
    file="pubreports.xlsx"
    replace;
    sheet="Paper &publisher";
  run;
%mend makespreadsheet;

%extfiles(Eversons Books,rtf=Y,spreadsheet=Y)
```

The above program creates only an RTF 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 MAKESPREADSHEET was not called. The conclusion is that the error is in the specification of the %ELSE statement.

```

185 %extfiles(Eversons Books,rtf=Y,spreadsheet=Y)
MLOGIC(EXTFILES): Beginning execution.
MLOGIC(EXTFILES): Parameter PUBLISHER has value Eversons Books
MLOGIC(EXTFILES): Parameter RTF has value Y
MLOGIC(EXTFILES): Parameter SPREADSHEET has value Y
MLOGIC(EXTFILES): %LET (variable name is RTF)
MLOGIC(EXTFILES): %LET (variable name is SPREADSHEET)

NOTE: There were 119 observations read from the data set
BOOKS.YTDSALES.
      WHERE publisher='Eversons Books';
NOTE: The data set WORK.TEMP has 119 observations and 11
variables.
NOTE: DATA statement used (Total process time):
      real time            0.00 seconds
      cpu time             0.00 seconds

MLOGIC(EXTFILES): %IF condition &rtf in Y YES is TRUE
MLOGIC(MAKERTF): Beginning execution.
NOTE: Writing RTF Body file: sasrtf.rtf
NOTE: There were 119 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(MAKERTF): Ending execution.
MLOGIC(EXTFILES): Ending execution.

```

Replacing the %ELSE statement with %IF corrects the logic error in EXTFILES. The revised EXTFILES macro program follows.

```

%macro extfiles(publisher,rtf=,spreadsheet=) / minoperator;
  %let rtf=%upcase(&rtf);
  %let spreadsheet=%upcase(&spreadsheet);

  data temp;
    set books.ytdsales(where=(publisher="&publisher"));
    lostprofit=listprice-saleprice;
  run;

  %if &rtf in Y YES %then %do;
    %makertf
  %end;

```

```
%if &spreadsheet in Y YES %then %do;
  %makespreadsheet
%end;
%mend extfiles;
```

Now the SAS log shows that both MAKERTF and MAKESPREADSHEET execute and that both the RTF file and the Excel workbook are created.

```
219 %extfiles(Eversons Books,rtf=Y,spreadsheet=Y)
MLOGIC(EXTFILES): Beginning execution.
MLOGIC(EXTFILES): Parameter PUBLISHER has value Eversons Books
MLOGIC(EXTFILES): Parameter RTF has value Y
MLOGIC(EXTFILES): Parameter SPREADSHEET has value Y
MLOGIC(EXTFILES): %LET (variable name is RTF)
MLOGIC(EXTFILES): %LET (variable name is SPREADSHEET)

NOTE: There were 119 observations read from the data set
      BOOKS.YTDSALES.
      WHERE publisher='Eversons Books';
NOTE: The data set WORK.TEMP has 119 observations and 11
      variables.
NOTE: DATA statement used (Total process time):
      real time            0.01 seconds
      cpu time             0.01 seconds

MLOGIC(EXTFILES): %IF condition in Y YES is TRUE
MLOGIC(MAKERTF): Beginning execution.
NOTE: Writing RTF Body file: sasrtf.rtf
NOTE: There were 119 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

NOTE: Writing HTML Body file: sashtml1.htm
MLOGIC(MAKERTF): Ending execution.
MLOGIC(EXTFILES): %IF condition &spreadsheet in Y YES is TRUE
MLOGIC(MAKESPREADSHEET): Beginning execution.
NOTE: "Ebook_Eversons_Books" range/sheet was successfully created.
NOTE: PROCEDURE EXPORT used (Total process time):
      real time            0.35 seconds
      cpu time             0.04 seconds

NOTE: "Paper_Eversons_Books" range/sheet was successfully created.
NOTE: PROCEDURE EXPORT used (Total process time):
      real time            0.33 seconds
      cpu time             0.04 seconds

MLOGIC(MAKESPREADSHEET): 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**

Example 12.6 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 problems in both macro language and SAS language.

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 2014 to 2018. The DATA step that creates data set PROJCOST computes projected costs using a different percentage increase for each of the three years from 2014 to 2018.

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 the program that immediately follows 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, PCOST2014, and PCOST2016.

```
%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} increase2014-increase2018
  (1.12,1.08,1.10,1.15,1.18);
array pcost{5} pcost2014-pcost2018;
drop i;

attrib pcost2014 label="Projected Cost 2014" format=dollar10.2
      pcost2015 label="Projected Cost 2015" format=dollar10.2
      pcost2016 label="Projected Cost 2016" format=dollar10.2
      pcost2017 label="Projected Cost 2017" format=dollar10.2
      pcost2018 label="Projected Cost 2018" format=dollar10.2;

do i=1 to 5;
  pcost{i}=round(cost*increase{i},.01);
end;
run;

%projcost(cost pcost2014 pcost2016)

```

Output 12.1 presents the output produced by the preceding call to PROJCOST. It shows that PROC TABULATE computed only the SUM statistic for the first and second analysis variables, COST and PCOST2014. For the third analysis variable, PCOST2016, PROC TABULATE computed both the mean and the sum statistics and formatted the results with the DOLLAR format.

#### Output 12.1 Output from first group of statements in Example 12.6

#### Projected Costs Report

	Wholesale Cost	Projected Cost 2014	Projected Cost 2016	
	Sum	Sum	Mean	Sum
Section				
Certification and Training	7636.18	8554.84	\$16.03	\$8,401.36
Networking	5935.48	6649.28	\$13.63	\$6,530.44
Operating Systems	9389.85	10518.24	\$17.87	\$10,330.56
Programming	5312.76	5950.54	\$17.34	\$5,845.06
Software	13823.21	15484.31	\$17.75	\$15,208.14
Web Development	6308.00	7066.54	\$12.16	\$6,940.52
All Sections	48405.48	54223.75	\$15.92	\$53,256.08

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 PCOST2014, as Output 12.1 shows.

```
637 %projcost(cost pcost2014 pcost2016)
MPRINT (PROJCOST): proc tabulate data=projcost;
MPRINT (PROJCOST): title "Projected Costs Report";
MPRINT (PROJCOST): class section;
MPRINT (PROJCOST): var cost pcost2014 pcost2016;
MPRINT (PROJCOST): tables section all='All Sections', cost
pcost2014 pcost2016*(mean*f=dollar7.2 sum*f=dollar12.2);
MPRINT (PROJCOST): run;

NOTE: There were 3346 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 includes parentheses around &ANALYSISVARS in the TABLE statement.

```
%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.

```
%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 ERROR- 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(,)$));❸
%if %sysfunc(notdigit(%sysfunc(compress(&minsaleprice,.)))) 
  gt 0
  %then %do; ❹
  %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)); ❺
%let publisher=%qcmpres(%superq(publisher)); ❻

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=$20.95,
  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.

- ③ The value of MONTHSOLD must be numeric and an integer from 1 to 12.
- ④ Remove dollar signs and commas from the value of MINSALEPRICE.
- ⑤ The value of MINSALEPRICE must be numeric and positive.
- ⑥ Uppercase and quote the value of PUBLISHER.
- ⑦ 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**

Automatic macro variables SYSERRORTTEXT 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 Example 12.8 checks whether any text has been stored in either of these two automatic variables. It writes messages to the SAS 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 Example 12.8 are in bold. In this example, assume ODS style BOOKSTORE does not exist. In the PROC FREQ step, variable DATESOLD is misspelled as DATESOLDD. The libref BOOOKS referenced in the DATA step has not been defined. Assume that libref BOOKS was defined before Example 12.8 was submitted.

```
%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 books.ytdsales;

  profit=saleprice-cost;
run;

ods rtf close;
%lastmsg

```

The SAS log for Example 12.8 follows, starting with submission of the first ODS RTF statement. 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, which is 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.**

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:

**The data set WORK.PROFIT may be incomplete. When this step was stopped there were 0 observations and 3 variables.**

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 step 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.

```
%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 ERROR- 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(Henderson, Jeffrey))

/* This author sold more than one book */
%authorreport(%str(Gonzales, Angela))
```

The first call to AUTHORREPORT produces the following message in the SAS log.

```
*****
ERROR: Author ALLAN, MICHAEL not found in data set BOOKS.YTDSALES
      No report produced.
*****
```

Output 12.2 shows the results of the second call to AUTHORREPORT.

**Output 12.2 Output produced by the second call to macro program AUTHORREPORT in Example 12.9**

### Book Sold for Author HENDERSON, JEFFREY

Title of Book	Date Book Sold	Wholesale Cost	Sale Price
Software Title 19	June	\$18.38	\$39.06

Output 12.3 shows the results of the third call to AUTHORREPORT.

**Output 12.3 Output produced by the third call to macro program AUTHORREPORT in Example 12.9**

**Books Sold for Author GONZALES, ANGELA**

Section	Date Book Sold	Title of Book	N	Wholesale Cost	Sale Price
				Total	Total
<b>Web Development</b>	January	Web Development Title 10	5	\$52.35	\$100.13
	February	Web Development Title 10	2	\$18.39	\$35.18
	March	Web Development Title 10	4	\$43.15	\$82.54
	April	Web Development Title 10	5	\$61.90	\$118.40
	May	Web Development Title 10	4	\$43.15	\$82.54
	June	Web Development Title 10	8	\$86.30	\$165.08
	July	Web Development Title 10	2	\$21.58	\$41.27
	August	Web Development Title 10	2	\$21.58	\$41.27
	October	Web Development Title 10	7	\$77.11	\$147.49
	November	Web Development Title 10	8	\$89.49	\$171.17
	December	Web Development Title 10	2	\$18.39	\$35.18
	<b>Total</b>		49	\$533.39	\$1,020.25

# **Chapter 13 A Stepwise Method for Writing Macro Programs**

<b>Introduction .....</b>	<b>307</b>
<b>Building a Macro Program in Four Steps .....</b>	<b>307</b>
<b>Applying the Four Steps to an Example.....</b>	<b>308</b>
Step 1: Write, test, and debug the SAS program(s) that you want the macro program to build.....	309
Step 2: Remove hard-coded programming constants from the program(s) in Step 1 and replace these constants with macro variables .....	317
Step 3: Create macro program(s) from the program(s) in Step 2.....	321
Step 4: Refine and generalize the macro program(s) in Step 3 by adding macro language statements like %IF-%THEN and %DO groups .....	325
Executing the REPORT Macro Program .....	328
Enhancing the Macro Program REPORT .....	338

---

## **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 macro programs developed in this step.

**Step 4. Refine and generalize the macro program(s) in Step 3 by adding macro language statements like %IF-%THEN and %DO groups.**

After you test several macro programs 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 book format (variable BOOKFMT); 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 default HTML destination, 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.
- Bar charts of SALEPRICE and PROFIT by section of the store when the specific time period spans quarters or a 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 default HTML destination, and optionally present the report in a specific 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, PROC SQL, and PROC SGLOT.

## **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, 2014–August 31, 2014.
- **Report B** presents totals and bar charts by SECTION for SALEPRICE and PROFIT for the first quarter of 2014.
- **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 and the report is formatted in the STATISTICAL style that is distributed with SAS software and found in SASHELP.TMPLMST.

### **Program for Report A with No Macro Facility Features**

Report A presents overall totals for COST, LISTPRICE, SALEPRICE, and PROFIT for July 1, 2014, through August 31, 2014.

```

-----REPORT A;
title "Sales Report";
title2 "July 1, 2014 - August 31, 2014";
data temp;
  set books.ytdsales(where=
    ('01jul2014'd le datesold le '31aug2014'd));
  profit=saleprice-cost;

  attrib profit label='Profit' format=dollar11.2;
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 from Step 1 Report A program

## Sales Report

### July 1, 2014 - August 31, 2014

<b>Titles Sold</b>	<b>Wholesale Cost</b>	<b>List Price</b>	<b>Sale Price</b>	<b>Profit</b>
	<b>Total</b>	<b>Total</b>	<b>Total</b>	<b>Total</b>
542	\$7,851.99	\$19,629.98	\$15,408.39	\$7,556.40

#### Program for Report B with No Macro Facility Features

Report B analyzes SALEPRICE and PROFIT for first quarter 2014. It presents a tabular report and two bar charts, one for each of the two analysis variables.

Each TITLE3 statement in the PROC SGPlot steps displays two items that need to be determined before submitting the PROC SGPlot steps. One item is the label of the analysis variable. The second item is the overall total of the analysis variable. The two SELECT statements in the PROC SQL step list these values. After executing the PROC SQL step, you insert the values into the TITLE3 statements. The output from PROC SQL is not shown.

```
*----REPORT B;
```

```

title "Sales Report";
title2 "January 1, 2014 - March 31, 2014";
data temp;
  set books.ytdsales(where=
    ('01jan2014'd le datesold le '31mar2014'd));
  profit=saleprice-cost;

  attrib profit label='Profit' format=dollar11.2;
run;
proc sql;
  select name,label from dictionary.columns
  where libname='WORK' and memname='TEMP' and
    (upcase(name)='SALEPRICE' or upcase(name)='PROFIT');
  select sum(saleprice) as saleprice_sum format=dollar11.2,
    sum(profit) as profit_sum format=dollar11.2
  from temp;
quit;

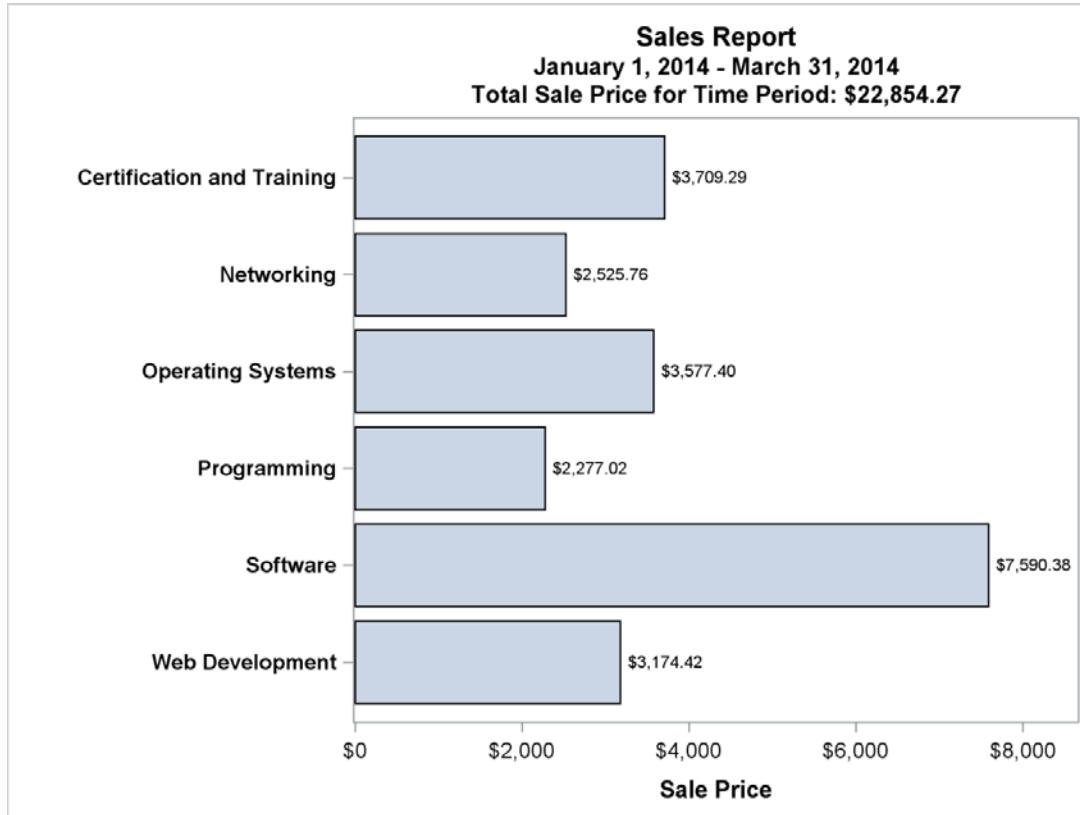
proc tabulate data=temp;
  class section;
  var saleprice profit;
  tables section all,
    n*f=6. (saleprice profit)*sum='Total'*f=dollar11.2;
  keylabel all='Total Sales'
    n='Titles Sold';
run;
proc sgplot data=temp;
  title3 "Total Sale Price for Time Period: $15,408.39";
  hbar section / stat=sum response=saleprice datalabel;
  xaxis labelattr=(weight=bold);
  yaxis label=' ' valueattr=(weight=bold);
run;
proc sgplot data=temp;
  title3 "Total Profit for Time Period: $7,556.40";
  hbar section / stat=sum response=profit datalabel;
  xaxis labelattr=(weight=bold);
  yaxis label=' ' valueattr=(weight=bold);
run;

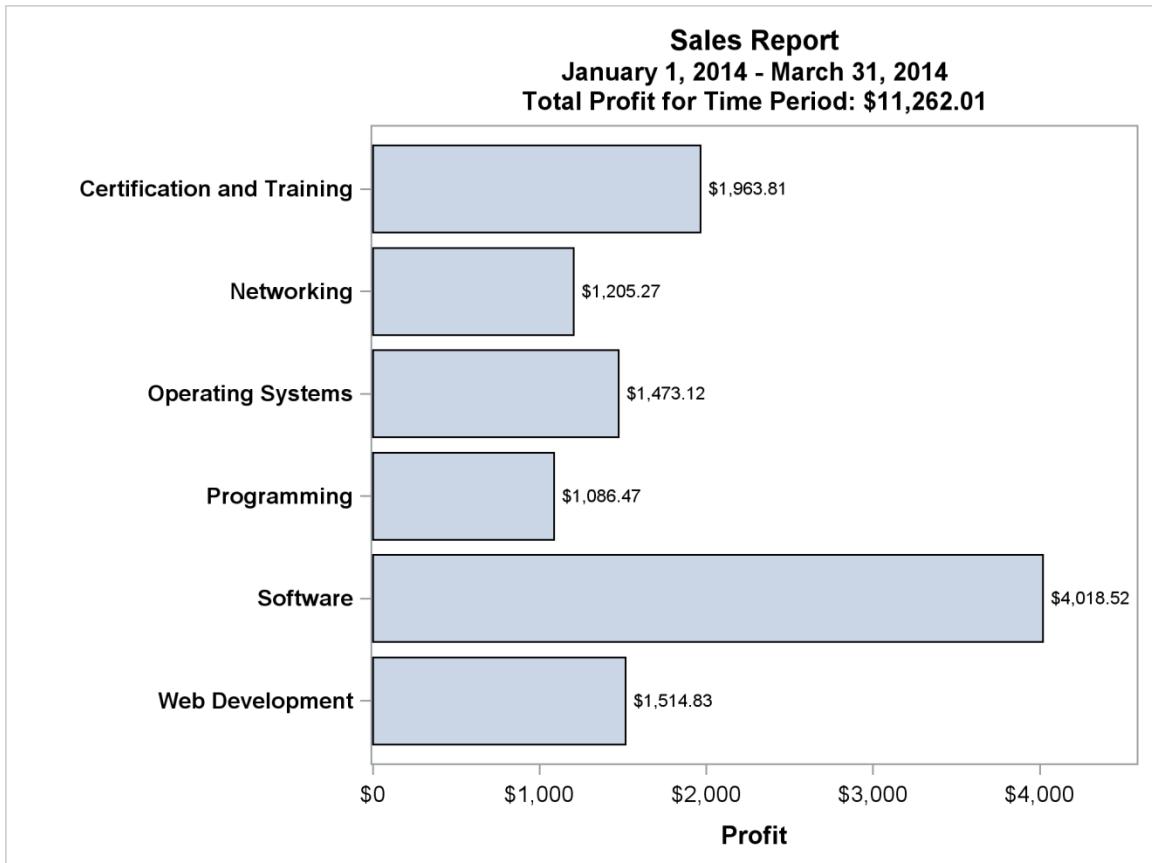
```

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****January 1, 2014 - March 31, 2014**

Section	Titles Sold	Sale Price	Profit
		Total	Total
Certification and Training	117	\$3,709.29	\$1,963.81
Networking	105	\$2,525.76	\$1,205.27
Operating Systems	130	\$3,577.40	\$1,473.12
Programming	74	\$2,277.02	\$1,086.47
Software	220	\$7,590.38	\$4,018.52
Web Development	150	\$3,174.42	\$1,514.83
<b>Total Sales</b>	<b>796</b>	<b>\$22,854.27</b>	<b>\$11,262.01</b>





### 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, 2014. The program sends the report to the RTF destination and formats it with the STATISTICAL style found in SASUSER.TMPLMST. It does not send any output to the default HTML destination.

```
*----REPORT C;
ods html close;
ods rtf style=statistical;

title "Sales Report";
title2 "January 1, 2014 - November 24, 2014";
data temp;
  set books.ytdsales(where=
('01jan2014'd le datesold le '24nov2014'd));
  profit=saleprice-cost;
```

```
attrib profit label='Profit' format=dollar11.2;  
  
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;  
  keylabel all='Total Sales'  
    n='Titles Sold';  
run;  
ods rtf close;  
ods html;
```

Output 13.3 presents the output produced by the Report C program. Since the report is two pages long, Output 13.3 shows just the first and last sections and the grand total.

**Output 13.3 Partial output produced by the Step 1 Report C program**

<b>Sales Report</b>					
<b>January 1, 2014 - November 24, 2014</b>					
<b>Section</b>	<b>Publisher</b>	<b>Titles Sold</b>	<b>Wholesale Cost</b>	<b>Profit</b>	
			<b>Sum</b>	<b>Sum</b>	<b>Sum</b>
<b>Certification and Training</b>	<b>AMZ Publishers</b>	90	\$1,355.83	\$1,525.43	
	<b>Bookstore Brand Titles</b>	67	\$961.56	\$1,081.79	
	<b>Eversons Books</b>	20	\$283.76	\$319.24	
	<b>IT Training Texts</b>	78	\$1,141.74	\$1,284.52	
	<b>Mainst Media</b>	15	\$215.01	\$241.90	
	<b>Northern Associates Titles</b>	25	\$365.66	\$411.39	
	<b>Technology Smith</b>	35	\$511.92	\$575.95	
	<b>Wide-World Titles</b>	144	\$2,078.14	\$2,338.02	
	<b>Total Sales</b>	474	\$6,913.62	\$7,778.24	
•					
•					
•					
<b>Web Development</b>	<b>Publisher</b>				
	<b>AMZ Publishers</b>	55	\$594.93	\$543.04	
	<b>Bookstore Brand Titles</b>	7	\$73.92	\$67.48	
	<b>IT Training Texts</b>	4	\$46.34	\$42.29	
	<b>Nifty New Books</b>	91	\$989.67	\$903.34	
	<b>Northern Associates Titles</b>	156	\$1,740.24	\$1,588.44	
	<b>Professional House Titles</b>	58	\$667.10	\$608.90	
	<b>Technology Smith</b>	148	\$1,625.28	\$1,483.51	
	<b>Total Sales</b>	519	\$5,737.48	\$5,237.00	
<b>Total Sales</b>		3007	\$43,576.15	\$42,071.29	

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=2014;
%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 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.

The TITLE3 statements in the PROC SGPlot steps in this version of Report B do not have to be manually edited with information produced by the PROC SQL step. In this version of Report B, PROC SQL saves the results it generates in macro variables. The TITLE3 statements reference the macro variables that the PROC SQL step creates.

The %SCAN function selects the name of each of the two analysis variables. The suffixes \_SUM and \_LABEL are concatenated to the analysis variable's name, and these concatenated text strings specify the names of the macro variables that the SELECT statements creates.

The TITLE3 statements repeat the %SCAN function calls, and two ampersands precede these function calls. The text strings \_SUM and \_LABEL follow the %SCAN function calls. The two ampersands are required so that the macro processor scans the reference twice. Recall the discussion of resolving multiple ampersands at the end of Chapter 3. On the first pass, the macro processor resolves the two ampersands into one and the %SCAN function finds the analysis variable name. On the second pass, the macro processor sees a single ampersand followed by the name of the macro variable. The macro processor is able to resolve the macro variable reference.

All of Report B has to be submitted only once to produce the output.

```

-----Report B;
%let repyear=2014;
%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 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=dollar11.2;
run;
proc sql noprint;
  select sum(%scan(&vars,1)) format=dollar11.2
    into :%scan(&vars,1)_sum
    from temp;
  select label into :%scan(&vars,1)_label trimmed
    from dictionary.columns
    where libname='WORK' and memname='TEMP' and
      upcase(name)="%upcase(%scan(&vars,1))";
  select sum(%scan(&vars,2)) format=dollar11.2
    into :%scan(&vars,2)_sum
    from temp;
  select label into :%scan(&vars,2)_label trimmed
    from dictionary.columns
    where libname='WORK' and memname='TEMP' and
      upcase(name)="%upcase(%scan(&vars,2))";

quit;
proc tabulate data=temp;
  class &classvar;
  var &vars;
  tables section all,
    n*f=6. (&vars)*sum='Total'*f=dollar11.2;
  keylabel all='Total Sales'
    n='Titles Sold';
run;

proc sgplot data=temp;
  title3
"Total &&%scan(&vars,1)_label for Time Period: &&%scan(&vars,1)_sum";
  hbar &classvar / stat=sum response=%scan(&vars,1) datalabel;
  xaxis labelatrs=(weight=bold);
  yaxis label=' ' valueatrs=(weight=bold);
run;
proc sgplot data=temp;
  title3
"Total &&%scan(&vars,2)_label for Time Period: &&%scan(&vars,2)_sum";
  hbar &classvar / stat=sum response=%scan(&vars,1) datalabel;
  xaxis labelatrs=(weight=bold);
  yaxis label=' ' valueatrs=(weight=bold);
run;;
```

## 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=2014;
%let start=01jan&repyear;
%let stop=&sysdate9;

%let classvar=section publisher;
%let vars=cost profit;

%let titlestart=%sysfunc(putn("&start"d,worddate.));
%let titlestop=%sysfunc(putn("&stop"d,worddate.));

%let outputdest=rtf;
%let outputstyle=statistical;

options symbolgen;

ods html 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=dollar11.2;
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;
  keylabel all='Total Sales'
    n='Titles Sold';
run;

ods &outputdest close;
ods html;
```

---

### 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 into 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 you define 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=2014,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 have to include 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 Step 2 Report B program. The goal, however, will be to produce bar charts only when the values specified for START= and STOP= correspond to quarter start and end dates.

```

options symbolgen mprint;

%macro reportb(repyear=,start=01JAN,stop=31DEC,
               classvar=,vars=);

%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=dollar11.2;
run;

proc sql noprint;
  select sum(%scan(&vars,1)) format=dollar11.2
    into :%scan(&vars,1)_sum

```

```

from temp;
select label into :%scan(&vars,1)_label trimmed
  from dictionary.columns
  where libname='WORK' and memname='TEMP' and
    upcase(name)="%upcase(%scan(&vars,1))";
select sum(%scan(&vars,2)) format=dollar11.2
  into :%scan(&vars,2)_sum
  from temp;
select label into :%scan(&vars,2)_label trimmed
  from dictionary.columns
  where libname='WORK' and memname='TEMP' and
    upcase(name)="%upcase(%scan(&vars,2))";
quit;

proc tabulate data=temp;
  class &classvar;
  var &vars;
  tables section all,
    n*f=6. (&vars)*sum='Total'*f=dollar11.2;
  keylabel all='Total Sales'
    n='Titles Sold';
run;

proc sgplot data=temp;
  title3
  "Total &&%scan(&vars,1)_label for Time Period: &&%scan(&vars,1)_sum";
  hbar &classvar / stat=sum response=%scan(&vars,1)
    datalabel;
  xaxis labelattrs=(weight=bold);
  yaxis label=' ' valueattrs=(weight=bold);
run;
proc sgplot data=temp;
  title3
  "Total &&%scan(&vars,2)_label for Time Period: &&%scan(&vars,2)_sum";
  hbar &classvar / stat=sum response=%scan(&vars,1)
    datalabel;
  xaxis labelattrs=(weight=bold);
  yaxis label=' ' valueattrs=(weight=bold);
run;

%mend reportb;

```

The call to REPORTB is written as follows:

```
%reportb(repyear=2014,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 adds two parameters that control the destination of the output. Macro programs REPORTA and REPORTB sent output to whatever the current destination in the SAS session was when they were called.

```

options symbolgen mprint;

%macro reportc(repyear=,start=01JAN,stop=31DEC,
               classvar=,vars=,
               outputdest=,style=);

%let start=&start&repyear;
%let stop=&stop&repyear;

%let classvar=section;
%let vars=saleprice profit;

%let titlestart=%sysfunc(putn("&start"d,worddate.));
%let titlestop=%sysfunc(putn("&stop"d,worddate.));

%let outputdest=rtf;
%let outputstyle=statistical;

ods html 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=dollar11.2;
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;
keylabel all='Total Sales'
  n='Titles Sold';
run;

ods &outputdest close;
ods html;

%mend reportc;

```

The call to REPORTC is specified as follows:

```
%reportc(repyear=2014,stop=24NOV,classvar=section publisher,
         vars=cost profit,outputdest=rtf,style=statistical)
```

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 and formats the output in the STATISTICAL style.

#### **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 of 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 bar charts.
- The number of PROC SGLOT steps in Report B is equal to the number of analysis variables.
- Report C is sent to a destination other than the default HTML destination.

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 statements in bold indicate where conditional and iterative processing occur as well as changes in parameter specifications.

A %IF statement checks if the time period spans one or more quarters or a year. If either condition is true, two iterative %DO loops execute. The output loop selects a classification variable from CLASSVARS. The inner loop selects an analysis variable from the VARS= parameter. Each iteration of the inner loop executes PROC SQL and PROC SGPLOT for the current value of CLASSVAR and VARNAME. Time period text is also added to the TITLE3 statement.

The parameters START= and STOP= in the Step 3 versions of the macro programs are renamed to STARTDDMM= and STOPDDMM in the following REPORT macro program definition.

Comments describe the processing of the macro program.

```

options mprint mlogic symbolgen;
%macro report(repyear=startddmmm=01JAN,stopddmmm=31DEC,
            classvar=,vars=cost listprice saleprice profit,
            outputdest=html,style=) / minoperator;

      *----Check if a value was specified for report year.
      If no value specified, use current year;
      %if &repyear= %then %let repyear=%substr(&sysdate9,6);
      *----Check if stop date specified. If null, use
          current date as stop date;
      %if &stopddmmm= %then %let stopddmmm=%substr(&sysdate,1,5);

      %let startddmmm=%upcase(&startddmmm);
      %let stopddmmm=%upcase(&stopddmmm);
      %let start=&startddmmm&repyear;
      %let stop=&stopddmmm&repyear;

      %let titlestart=%sysfunc(putn("&start"d,worddate.));
      %let titlestop=%sysfunc(putn("&stop"d,worddate.));

      *----Determine number of classification variables;
      %if &classvar ne %then
          %let nclassvars=%sysfunc(countw(&classvar));
      %else %let nclassvars=0;

```

```

%*----Close default destination and open OUTPUTDEST;
%*----Add STYLE if specified;
ods html close;
ods &outputdest
  %if &style ne %then %do;
    style=&style
  %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;
      %scan(&classvar,1)
    %if &nclassvars ge 2 %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 quarter(s) or year;

```

```

%if &startddmmmm # 01JAN 01APR 01JUL 01OCT and
  &stopddmmmm # 31MAR 30JUN 30SEP 31DEC and
  &startddmmmm ne &stopddmmmm %then %do;

  %-----Specific title text for Quarter(s) and for Year;
  %if &startddmmmm=01JAN and &stopddmmmm=31DEC
    %then %let timeperiod=Year;
  %else %let timeperiod=Quarter(s);

  %let nvars=%sysfunc(countw(&vars));
  %do c=1 %to &nclassvars;
    %let classvarname=%scan(&classvar,&c);
    %do v=1 %to &nvars;
      %let varname=%scan(&vars,&v);
      proc sql noprint;
        select sum(&varname) format=dollar11.2
          into :&varname._sum
        from temp;
        select label into :&varname._label trimmed
          from dictionary.columns
        where libname='WORK' and memname='TEMP' and
          upcase(name)="%upcase(&varname)";
      quit;
      proc sgplot data=temp;
        title3
        "Total &&&varname._label for &timeperiod: &&&varname._sum";
        hbar &classvarname / stat=sum response=&varname
          datalabel;
        xaxis labelattrs=(weight=bold);
        yaxis label=' ' valueattrs=(weight=bold);
      run;
    %end;
  %end;
%end;

%-----Close report output. Open default destination;
ods &outputdest close;
ods html;

%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 2014 is as follows:

```
%report(repyear=2014,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.

The output produced by this call to macro program %REPORT is identical to that in Output 13.1.

```
4496  %report(repyear=2014,startddmmm=01jul,stopddmmm=31aug)
MLOGIC (REPORT): Beginning execution.
MLOGIC (REPORT): Parameter REPYEAR has value 2014
MLOGIC (REPORT): Parameter STARTDDMMM has value 01jul
MLOGIC (REPORT): Parameter STOPDDMMM 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 html
MLOGIC (REPORT): Parameter STYLE has value
SYMBOLGEN: Macro variable REPYEAR resolves to 2014
MLOGIC (REPORT): %IF condition &repyear= is FALSE
SYMBOLGEN: Macro variable STOPDDMMM resolves to 31aug
MLOGIC (REPORT): %IF condition &stopddmmm= is FALSE
MLOGIC (REPORT): %LET (variable name is STARTDDMMM)
SYMBOLGEN: Macro variable STARTDDMMM resolves to 01jul
MLOGIC (REPORT): %LET (variable name is STOPDDMMM)
SYMBOLGEN: Macro variable STOPDDMMM resolves to 31aug
MLOGIC (REPORT): %LET (variable name is START)
SYMBOLGEN: Macro variable STARTDDMMM resolves to 01JUL
SYMBOLGEN: Macro variable REPYEAR resolves to 2014
MLOGIC (REPORT): %LET (variable name is STOP)
SYMBOLGEN: Macro variable STOPDDMMM resolves to 31AUG
SYMBOLGEN: Macro variable REPYEAR resolves to 2014
MLOGIC (REPORT): %LET (variable name is TITLESTART)
SYMBOLGEN: Macro variable START resolves to 01JUL2014
MLOGIC (REPORT): %LET (variable name is TITLESTOP)
SYMBOLGEN: Macro variable STOP resolves to 31AUG2014
SYMBOLGEN: Macro variable CLASSVAR resolves to
MLOGIC (REPORT): %IF condition &classvar ne is FALSE
MLOGIC (REPORT): %LET (variable name is NCLASSVARS)
MPRINT (REPORT): ods html close;
SYMBOLGEN: Macro variable OUTPUTDEST resolves to html
SYMBOLGEN: Macro variable STYLE resolves to
MLOGIC (REPORT): %IF condition &style ne is FALSE
MPRINT (REPORT): ods html ;
NOTE: Writing HTML Body file: sashtml47.htm
MPRINT (REPORT): title "Sales Report";
SYMBOLGEN: Macro variable TITLESTART resolves to July 1, 2014
```

```

SYMBOLGEN: Macro variable TITLESTOP resolves to August 31, 2014
MPRINT(REPORT): title2 "July 1, 2014 - August 31, 2014";
MPRINT(REPORT): data temp;
SYMBOLGEN: Macro variable START resolves to 01JUL2014
SYMBOLGEN: Macro variable STOP resolves to 31AUG2014
MPRINT(REPORT): set books.ytdsales(where= ("01JUL2014'd le datesold
le "31AUG2014"d));
MPRINT(REPORT): profit=saleprice-cost;
MPRINT(REPORT): attrib profit label='Profit' format=dollar10.2;
MPRINT(REPORT): run;

NOTE: There were 542 observations read from the data set
BOOKS.YTDSALES.
      WHERE (datesold>='01JUL2014'D and datesold<='31AUG2014'D);
NOTE: The data set WORK.TEMP has 542 observations and 11 variables.
NOTE: DATA statement used (Total process time):
      real time            0.03 seconds
      cpu time             0.01 seconds

MPRINT(REPORT): proc tabulate data=temp;
SYMBOLGEN: Macro variable CLASSVAR resolves to
MLOGIC(REPORT): %IF condition &classvar ne is FALSE
SYMBOLGEN: Macro variable VARS resolves to cost listprice saleprice
profit
MPRINT(REPORT): var cost listprice saleprice profit;
SYMBOLGEN: Macro variable CLASSVAR resolves to
MLOGIC(REPORT): %IF condition &classvar ne is FALSE
SYMBOLGEN: Macro variable VARS resolves to cost listprice saleprice
profit
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 542 observations read from the data set WORK.TEMP.
NOTE: PROCEDURE TABULATE used (Total process time):
      real time            0.12 seconds
      cpu time             0.01 seconds

SYMBOLGEN: Macro variable STARTDDMMM resolves to 01JUL
SYMBOLGEN: Macro variable STOPDDMMM resolves to 31AUG
SYMBOLGEN: Macro variable STARTDDMMM resolves to 01JUL
SYMBOLGEN: Macro variable STOPDDMMM resolves to 31AUG
MLOGIC(REPORT): %IF condition &startddmmm # 01JAN 01APR 01JUL 01OCT
and      &stopddmmm #
      31MAR 30JUN 30SEP 31DEC and      &startddmmm ne &stopddmmm is
FALSE
SYMBOLGEN: Macro variable OUTPUTDEST resolves to html
MPRINT(REPORT): ods html close;
MPRINT(REPORT): ods html;

```

NOTE: Writing HTML Body file: sashtml48.htm  
MLOGIC(REPORT): Ending execution.

## 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 2014. Since the reporting time period is a quarter, the macro program executes PROC SQL and PROC SGPlot to produce bar charts.

```
%report(repyear=2014,stopddmmm=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.

The output produced by this call to macro program %REPORT is identical to that in Output 13.2.

```
4613  %report(repyear=2014,stopddmmm=31Mar,classvar=section,
MLOGIC(REPORT): Beginning execution.
4614          vars=saleprice profit)
MLOGIC(REPORT): Parameter REPYEAR has value 2014
MLOGIC(REPORT): Parameter STOPDDMMM has value 31Mar
MLOGIC(REPORT): Parameter CLASSVAR has value section
MLOGIC(REPORT): Parameter VARS has value saleprice profit
MLOGIC(REPORT): Parameter STARTDDMMM has value 01JAN
MLOGIC(REPORT): Parameter OUTPUTDEST has value html
MLOGIC(REPORT): Parameter STYLE has value
SYMBOLGEN: Macro variable REPYEAR resolves to 2014
MLOGIC(REPORT): %IF condition &repyear= is FALSE
SYMBOLGEN: Macro variable STOPDDMMM resolves to 31Mar
MLOGIC(REPORT): %IF condition &stopddmmm= is FALSE
MLOGIC(REPORT): %LET (variable name is STARTDDMMM)
SYMBOLGEN: Macro variable STARTDDMMM resolves to 01JAN
MLOGIC(REPORT): %LET (variable name is STOPDDMMM)
SYMBOLGEN: Macro variable STOPDDMMM resolves to 31Mar
MLOGIC(REPORT): %LET (variable name is START)
SYMBOLGEN: Macro variable STARTDDMMM resolves to 01JAN
SYMBOLGEN: Macro variable REPYEAR resolves to 2014
MLOGIC(REPORT): %LET (variable name is STOP)
SYMBOLGEN: Macro variable STOPDDMMM resolves to 31MAR
SYMBOLGEN: Macro variable REPYEAR resolves to 2014
MLOGIC(REPORT): %LET (variable name is TITLESTART)
SYMBOLGEN: Macro variable START resolves to 01JAN2014
MLOGIC(REPORT): %LET (variable name is TITLESTOP)
SYMBOLGEN: Macro variable STOP resolves to 31MAR2014
SYMBOLGEN: Macro variable CLASSVAR resolves to section
MLOGIC(REPORT): %IF condition &classvar ne is TRUE
```

```

MLOGIC (REPORT): %LET (variable name is NCLASSVARS)
SYMBOLGEN: Macro variable CLASSVAR resolves to section
MPRINT (REPORT): ods html close;
SYMBOLGEN: Macro variable OUTPUTDEST resolves to html
SYMBOLGEN: Macro variable STYLE resolves to
MLOGIC (REPORT): %IF condition &style ne is FALSE
MPRINT (REPORT): ods html ;
NOTE: Writing HTML Body file: sashtml51.htm
MPRINT (REPORT): title "Sales Report";
SYMBOLGEN: Macro variable TITLESTART resolves to January 1, 2014
SYMBOLGEN: Macro variable TITLESTOP resolves to March 31, 2014
MPRINT (REPORT): title2 "January 1, 2014 - March 31, 2014";
MPRINT (REPORT): data temp;
SYMBOLGEN: Macro variable START resolves to 01JAN2014
SYMBOLGEN: Macro variable STOP resolves to 31MAR2014
MPRINT (REPORT): set books.ytdsales(where= ("01JAN2014'd le datesold
le "31MAR2014"d));
MPRINT (REPORT): profit=saleprice-cost;
MPRINT (REPORT): attrib profit label='Profit' format=dollar10.2;
MPRINT (REPORT): run;

NOTE: There were 796 observations read from the data set
BOOKS.YTDSALES.
      WHERE (datesold>='01JAN2014'D and datesold<='31MAR2014'D);
NOTE: The data set WORK.TEMP has 796 observations and 11 variables.
NOTE: DATA statement used (Total process time):
      real time           0.04 seconds
      cpu time            0.03 seconds

MPRINT (REPORT): proc tabulate data=temp;
SYMBOLGEN: Macro variable CLASSVAR resolves to section
MLOGIC (REPORT): %IF condition &classvar ne is TRUE
SYMBOLGEN: Macro variable CLASSVAR resolves to section
MPRINT (REPORT): class section;
SYMBOLGEN: Macro variable VARS resolves to saleprice profit
MPRINT (REPORT): var saleprice profit;
SYMBOLGEN: Macro variable CLASSVAR resolves to section
MLOGIC (REPORT): %IF condition &classvar ne is TRUE
SYMBOLGEN: Macro variable CLASSVAR resolves to section
SYMBOLGEN: Macro variable NCLASSVARS resolves to 1
MLOGIC (REPORT): %IF condition &nclassvars ge 2 is FALSE
SYMBOLGEN: Macro variable VARS resolves to saleprice profit
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 796 observations read from the data set WORK.TEMP.
NOTE: PROCEDURE TABULATE used (Total process time):
      real time            0.15 seconds
      cpu time             0.03 seconds

SYMBOLGEN: Macro variable STARTDDMMM resolves to 01JAN
SYMBOLGEN: Macro variable STOPDDMMM resolves to 31MAR
SYMBOLGEN: Macro variable STARTDDMMM resolves to 01JAN
SYMBOLGEN: Macro variable STOPDDMMM resolves to 31MAR
MLOGIC(REPORT): %IF condition &startddmmm # 01JAN 01APR 01JUL 01OCT
and &stopddmmm # 31MAR 30JUN 30SEP 31DEC and &startddmmm ne
&stopddmmm is TRUE
SYMBOLGEN: Macro variable STARTDDMMM resolves to 01JAN
SYMBOLGEN: Macro variable STOPDDMMM resolves to 31MAR
MLOGIC(REPORT): %IF condition &startddmmm=01JAN and &stopddmmm=31DEC
is FALSE
MLOGIC(REPORT): %LET (variable name is TIMEPERIOD)
MLOGIC(REPORT): %LET (variable name is NVARS)
SYMBOLGEN: Macro variable VARS resolves to saleprice profit
SYMBOLGEN: Macro variable NCLASSVARS resolves to 1
MLOGIC(REPORT): %DO loop beginning; index variable C; start value is
1; stop value is 1; by value is 1.
MLOGIC(REPORT): %LET (variable name is CLASSVARNAME)
SYMBOLGEN: Macro variable CLASSVAR resolves to section
SYMBOLGEN: Macro variable C resolves to 1
SYMBOLGEN: Macro variable NVARS resolves to 2
MLOGIC(REPORT): %DO loop beginning; index variable V; start value is
1; stop value is 2; by value is 1.
MLOGIC(REPORT): %LET (variable name is VARNAME)
SYMBOLGEN: Macro variable VARS resolves to saleprice profit
SYMBOLGEN: Macro variable V resolves to 1
MPRINT(REPORT): proc sql noprint;
SYMBOLGEN: Macro variable VARNAME resolves to saleprice
SYMBOLGEN: Macro variable VARNAME resolves to saleprice
MPRINT(REPORT): select sum(saleprice) format=dollar11.2 into
:saleprice_sum from temp;
SYMBOLGEN: Macro variable VARNAME resolves to saleprice
SYMBOLGEN: Macro variable VARNAME resolves to saleprice
MPRINT(REPORT): select label into :saleprice_label trimmed from
dictionary.columns where libname='WORK' and memname='TEMP' and
upcase(name)="SALEPRICE";
MPRINT(REPORT): quit;
NOTE: PROCEDURE SQL used (Total process time):
      real time            0.00 seconds
      cpu time             0.00 seconds

MPRINT(REPORT): proc sgplot data=temp;

```

```

SYMBOLGEN:  && resolves to &.
SYMBOLGEN:  Macro variable VARNAME resolves to saleprice
SYMBOLGEN:  Macro variable SALEPRICE_LABEL resolves to Sale Price
SYMBOLGEN:  Macro variable TIMEPERIOD resolves to Quarter(s)
SYMBOLGEN:  && resolves to &.
SYMBOLGEN:  Macro variable VARNAME resolves to saleprice
SYMBOLGEN:  Macro variable SALEPRICE_SUM resolves to $22,854.27
MPRINT (REPORT):  title3 "Total Sale Price for Quarter(s):
$22,854.27";
SYMBOLGEN:  Macro variable CLASSVARNAME resolves to section
SYMBOLGEN:  Macro variable VARNAME resolves to saleprice
MPRINT (REPORT):  hbar section / stat=sum response=saleprice
datalabel;
MPRINT (REPORT):  xaxis labelattrs=(weight=bold);
MPRINT (REPORT):  yaxis label=' ' valueattrs=(weight=bold);
MPRINT (REPORT):  run;

NOTE: PROCEDURE SGLOT used (Total process time):
      real time           0.90 seconds
      cpu time            0.12 seconds

NOTE: There were 796 observations read from the data set WORK.TEMP.

MLOGIC (REPORT): %DO loop index variable V is now 2; loop will
iterate again.
MLOGIC (REPORT): %LET (variable name is VARNAME)
SYMBOLGEN: Macro variable VARS resolves to saleprice profit
SYMBOLGEN: Macro variable V resolves to 2
MPRINT (REPORT): proc sql noprint;
SYMBOLGEN: Macro variable VARNAME resolves to profit
SYMBOLGEN: Macro variable VARNAME resolves to profit
MPRINT (REPORT): select sum(profit) format=dollar11.2 into
:profit_sum from temp;
SYMBOLGEN: Macro variable VARNAME resolves to profit
SYMBOLGEN: Macro variable VARNAME resolves to profit
MPRINT (REPORT): select label into :profit_label trimmed from
dictionary.columns where libname='WORK' and memname='TEMP' and
upcase(name)="PROFIT";
MPRINT (REPORT): quit;
NOTE: PROCEDURE SQL used (Total process time):
      real time           0.00 seconds
      cpu time            0.00 seconds

MPRINT (REPORT): proc sgplot data=temp;
SYMBOLGEN: && resolves to &.
SYMBOLGEN: Macro variable VARNAME resolves to profit
SYMBOLGEN: Macro variable PROFIT_LABEL resolves to Profit

```

```

SYMBOLGEN: Macro variable TIMEPERIOD resolves to Quarter(s)
SYMBOLGEN: && resolves to &.
SYMBOLGEN: Macro variable VARNAME resolves to profit
SYMBOLGEN: Macro variable PROFIT_SUM resolves to $11,262.01
MPRINT (REPORT): title3 "Total Profit for Quarter(s): $11,262.01";
SYMBOLGEN: Macro variable CLASSVARNAME resolves to section
SYMBOLGEN: Macro variable VARNAME resolves to profit
MPRINT (REPORT): hbar section / stat=sum response=profit datalabel;
MPRINT (REPORT): xaxis labelattrs=(weight=bold);
MPRINT (REPORT): yaxis label=' ' valueattrs=(weight=bold);
MPRINT (REPORT): run;

NOTE: PROCEDURE SGPLOT used (Total process time):
      real time            0.39 seconds
      cpu time             0.03 seconds

NOTE: There were 796 observations read from the data set WORK.TEMP.

MLOGIC(REPORT): %DO loop index variable V is now 3; loop will not
iterate again.
MLOGIC(REPORT): %DO loop index variable C is now 2; loop will not
iterate again.
SYMBOLGEN: Macro variable OUTPUTDEST resolves to html
MPRINT (REPORT): ods html close;
MPRINT (REPORT): ods html;
NOTE: Writing HTML Body file: sashtml52.htm
MLOGIC(REPORT): Ending execution.

```

## 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 classified by SECTION and PUBLISHER. Assume that the program was submitted on November 24, 2014. The program sends output to the RTF destination and formats the report using the STATISTICAL style. The third call to REPORT follows.

```
%report(stop=,
       classvar=section publisher,
       vars=cost profit,
       outputdest=rtf,style=statistical)
```

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.

The output produced by this call to macro program %REPORT is identical to that in Output 13.3.

```
4616 %report(stopddmmm=,
MLOGIC(REPORT): Beginning execution.
4617         classvar=section publisher,
```

```

4618      vars=cost profit,
4619          outputdest=rtf,style=analysis)
MLOGIC (REPORT): Parameter STOPDDMMM 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 analysis
MLOGIC (REPORT): Parameter REPYEAR has value
MLOGIC (REPORT): Parameter STARTDDMMM has value 01JAN
SYMBOLGEN: Macro variable REPYEAR resolves to
MLOGIC (REPORT): %IF condition &repyear= is TRUE
MLOGIC (REPORT): %LET (variable name is REPYEAR)
SYMBOLGEN: Macro variable SYSDATE9 resolves to 24NOV2014
SYMBOLGEN: Macro variable STOPDDMMM resolves to
MLOGIC (REPORT): %IF condition &stopddmmm= is TRUE
MLOGIC (REPORT): %LET (variable name is STOPDDMMM)
SYMBOLGEN: Macro variable SYSDATE resolves to
24NOV2014MLOGIC (REPORT): %LET (variable name is STARTDDMMM)
SYMBOLGEN: Macro variable STARTDDMMM resolves to 01JAN
MLOGIC (REPORT): %LET (variable name is STOPDDMMM)
SYMBOLGEN: Macro variable STOPDDMMM resolves to 24NOV
MLOGIC (REPORT): %LET (variable name is START)
SYMBOLGEN: Macro variable STARTDDMMM resolves to 01JAN
SYMBOLGEN: Macro variable REPYEAR resolves to 2014
MLOGIC (REPORT): %LET (variable name is STOP)
SYMBOLGEN: Macro variable STOPDDMMM resolves to 24NOV
SYMBOLGEN: Macro variable REPYEAR resolves to 2014
MLOGIC (REPORT): %LET (variable name is TITLESTART)
SYMBOLGEN: Macro variable START resolves to 01JAN2014
MLOGIC (REPORT): %LET (variable name is TITLESTOP)
SYMBOLGEN: Macro variable STOP resolves to 24NOV2014
SYMBOLGEN: Macro variable CLASSVAR resolves to section publisher
MLOGIC (REPORT): %IF condition &classvar ne is TRUE
MLOGIC (REPORT): %LET (variable name is NCLASSVARS)
SYMBOLGEN: Macro variable CLASSVAR resolves to section publisher
MPRINT (REPORT): ods html close;
SYMBOLGEN: Macro variable OUTPUTDEST resolves to rtf
SYMBOLGEN: Macro variable STYLE resolves to analysis
MLOGIC (REPORT): %IF condition &style ne is TRUE
SYMBOLGEN: Macro variable STYLE resolves to analysis
MPRINT (REPORT): ods rtf style=analysis ;
NOTE: Writing RTF Body file: sasrtf.rtf
MPRINT (REPORT): title "Sales Report";
SYMBOLGEN: Macro variable TITLESTART resolves to January 1, 2014
SYMBOLGEN: Macro variable TITLESTOP resolves to November 24, 2014
MPRINT (REPORT): title2 "January 1, 2014 - November 24, 2014";
MPRINT (REPORT): data temp;
SYMBOLGEN: Macro variable START resolves to 01JAN2014
SYMBOLGEN: Macro variable STOP resolves to 24NOV2014
MPRINT (REPORT): set books.ytdsales(where= ("01JAN2014"le datesold
le "24NOV2014")) ;

```

```

MPRINT (REPORT) :  profit=saleprice-cost;
MPRINT (REPORT) :  attrib profit label='Profit' format=dollar10.2;
MPRINT (REPORT) :  run;

NOTE: There were 3007 observations read from the data set
BOOKS.YTDSALES.
      WHERE (datesold>='01JAN2014'D and datesold<='24NOV2014'D);
NOTE: The data set WORK.TEMP has 3007 observations and 11 variables.
NOTE: DATA statement used (Total process time):
      real time            0.04 seconds
      cpu time             0.01 seconds

MPRINT (REPORT) :  proc tabulate data=temp;
SYMBOLGEN: Macro variable CLASSVAR resolves to section publisher
MLOGIC (REPORT): %IF condition &classvar ne is TRUE
SYMBOLGEN: Macro variable CLASSVAR resolves to section publisher
MPRINT (REPORT) :  class section publisher;
SYMBOLGEN: Macro variable VARS resolves to cost profit
MPRINT (REPORT) :  var cost profit;
SYMBOLGEN: Macro variable CLASSVAR resolves to section publisher
MLOGIC (REPORT): %IF condition &classvar ne is TRUE
SYMBOLGEN: Macro variable CLASSVAR resolves to section publisher
SYMBOLGEN: Macro variable NCLASSVARS resolves to 2
MLOGIC (REPORT): %IF condition &nclassvars ge 2 is TRUE
MLOGIC (REPORT): %LET (variable name is POS2)
SYMBOLGEN: Macro variable CLASSVAR resolves to section publisher
SYMBOLGEN: Macro variable CLASSVAR resolves to section publisher
SYMBOLGEN: Macro variable CLASSVAR resolves to section publisher
SYMBOLGEN: Macro variable POS2 resolves to 9
SYMBOLGEN: Macro variable VARS resolves to cost profit
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 3007 observations read from the data set WORK.TEMP.
NOTE: PROCEDURE TABULATE used (Total process time):
      real time            0.14 seconds
      cpu time             0.03 seconds

SYMBOLGEN: Macro variable STARTDDMMM resolves to 01JAN
SYMBOLGEN: Macro variable STOPDDMMM resolves to 24NOV
SYMBOLGEN: Macro variable STARTDDMMM resolves to 01JAN
SYMBOLGEN: Macro variable STOPDDMMM resolves to 24NOV
MLOGIC (REPORT): %IF condition &startddmmm # 01JAN 01APR 01JUL 01OCT
and &stopddmmm # 31MAR 30JUN 30SEP 31DEC and &startddmmm ne &stopddmmm
is FALSE
SYMBOLGEN: Macro variable OUTPUTDEST resolves to rtf
MPRINT (REPORT) :  ods rtf close;
MPRINT (REPORT) :  ods html;

```

NOTE: Writing HTML Body file: sashtml53.htm  
MLOGIC(REPORT): Ending execution.

---

## 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 you can analyze other data sets.
- Check that the data set exists and that it contains observations.
- Add more error checking of 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.
- Enhance the output produced by PROC SGLOT output. Add parameters to further customize the output.
- Delete the temporary data set created by the macro program.
- Declare as local all the macro variables that REPORT creates.

## **Part 3 Appendixes**

**Appendix A      Sample Data Set  341**

**Appendix B      Reference to Examples in This Book  343**



## Appendix A Sample Data Set

The following selected PROC CONTENTS output describes data set BOOKS.YTDSALES, which is used in most examples in this book. To create this data set, download the code for this book from the author page located here: <http://support.sas.com/publishing/authors/burlew.html>.

### PROC CONTENTS of BOOKS.YTDSALES

#### The CONTENTS Procedure

Data Set Name	BOOKS.YTDSALES	Observations	3346
Member Type	DATA	Variables	10
Engine	V9	Indexes	0
Label	Sales for 2014		

Alphabetic List of Variables and Attributes						
#	Variable	Type	Len	Format	Informat	Label
3	author	Char	50			First Author
5	bookfmt	Char	5			Book Format
2	booktitle	Char	50			Title of Book
6	cost	Num	8	DOLLAR10.2		Wholesale Cost
10	datesold	Num	4	MMDDYY10.	MMDDYY10.	Date Book Sold
7	listprice	Num	8	DOLLAR10.2		List Price
4	publisher	Char	50			Publisher
9	saleid	Num	8	8.		Sale ID
8	saleprice	Num	8	DOLLAR10.2		Sale Price
1	section	Char	30			Section



## Appendix B Reference to Examples in This Book

Table B.1 lists the programs that are used in the examples in this book. The list includes the names of the macro programs for programs that define them.

**Table B.1 Macro programs used in the examples in this book**

Chapter	Example	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
	1.3		Define macro variables in open code and reference them in TITLE statements, a PROC TABULATE step, and a PROC SGPANEL 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

---

<b>Chapter</b>	<b>Example</b>	<b>Macro Program Name</b>	<b>Function</b>
3	3.1		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, reference them, and reference 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 Example 3.1 with the SYMBOLGEN option enabled.
	3.7		Reference automatic macro variables in WHERE, 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.
	3.10		Construct a PROC FREQ TABLES statement when a macro variable reference precedes text. Demonstrate necessity of macro variable delimiters.
	3.11		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.
	3.13		Demonstrate resolution of concatenated macro variable references. Demonstrate resolution of indirect referencing of macro variables with SYMBOLGEN enabled.
	3.14		Define a series of macro variables in open code and demonstrate resolution of macro variable references when multiple ampersands precede a macro variable reference.

---

---

<b>Chapter</b>	<b>Example</b>	<b>Macro Program Name</b>	<b>Function</b>
4	4.1	PROFITCHART	Specify one PROC SGPLOT step.
	4.2		Specify a PROC CATALOG step for the WORK.SASMACR catalog.
	4.3		Create a data set and call PROFITCHART defined in Example 4.1.
	4.4	LISTPARM	Defined with three positional parameters that specify options for a PROC MEANS step and range in dates for selection of observations to analyze.
	4.5	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 Example 4.4 with positional parameters replaced with keyword parameters.
	4.6	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 Examples 4.4 and 4.5 with differences in types of parameters.
	4.7	PBUFFPARMS	Defined with PARMBUFF option. Contains two PROC SGPLOT steps, which are conditionally executed based on parameter specifications. If parameters specified, one PROC SGPlot 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 Example 5.1 MAKEDS with subset specified with positional parameter. Produces a PROC TABULATE report.
	5.3	MAKEDS	Modifies Example 5.2 MAKEDS to include PROC TABULATE step as well as the DATA step.
	5.4	MAKEDS	Modifies Example 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 a local macro variable not the same-named global macro variable.

---

---

<b>Chapter</b>	<b>Example</b>	<b>Macro Program Name</b>	<b>Function</b>
6	6.1		Use %SUBSTR to extract a substring of text from the value of an automatic macro variable and insert this value in a title and on a WHERE statement.
	6.2		Use %SCAN to extract the nth word from a macro variable and insert this word in a title.
	6.3	LISTTEXT	Define a parameter to LISTTEXT that will be used to select a subset of a data set for processing by PROC PRINT. The text that should be present in a variable's value is the parameter value passed to LISTTEXT. Insert the text string in the title.
	6.4	MAKEDS	Modified version of Example 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	CHECKVARNAM	Use %SYSFUNC and SAS language functions to determine if positional parameter value is a valid SAS name. Results listed by %PUT statements.
	6.8		Compute the mean of four values stored in macro variables using %SYSFUNC and the MEAN SAS language function. 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.
	6.10	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.

---

---

<b>Chapter</b>	<b>Example</b>	<b>Macro Program Name</b>	<b>Function</b>
7	7.1	COMP2VARS	Compare precedence of values of two positional parameters. List results 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. Demonstrates IN operator and MINDELIMITER.
	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 TABULATE step and a PROC SGPLOT step.
	7.6	SUMYEARS	Process an iterative %DO loop that concatenates data sets within range of years specified by the two positional parameters. Specify that PROC SGPLOT plot data from this new data set.
	7.7	MOSALES	Defined with PARMBUFF option in which %DO %UNTIL processes the parameter string and executes a PROC MEANS for each parameter value. Produces an overall PROC MEANS when no parameters specified.
	7.8	MOYRSALES	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 the subset to analyze.
	7.9	DETAIL	Defined with one positional parameter that specifies a data set, one positional parameter that defines a list of variables to analyze, and one keyword parameter. Check if the specified 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.

---

---

<b>Chapter</b>	<b>Example</b>	<b>Macro Program Name</b>	<b>Function</b>
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.
	8.5		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. 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.
	8.11	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	Demonstrate use of %UNQUOTE.
	8.13		Use %QSYSFUNC to mask the result of applying a SAS language function.
	8.14		Use %QSUBSTR to mask the result of taking a substring of a macro variable value.

---

---

<b>Chapter</b>	<b>Example</b>	<b>Macro Program Name</b>	<b>Function</b>
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 SYMGET 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		Sum a variable in a DATA step. Save the sum in a macro variable by executing CALL SYMPUTX 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 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 TITLE statements later reference.
	9.8	LISTAUTOMATIC	Contains %PUT statements and lists automatic macro variables. Demonstrates CALL EXECUTE calling a macro program in a DATA step.
	9.9	LISTLIBRARY	Specifies a PROC DATASETS step on BOOKS library. Demonstrates CALL EXECUTE calling a macro program in a DATA step.
	9.10	REP16K	Specify a PROC REPORT step on a subset specified by the value of the single positional parameter. Before REP16K is called, execute a PROC MEANS step and save 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.

---

---

<b>Chapter</b>	<b>Example</b>	<b>Macro Program Name</b>	<b>Function</b>
9	9.12		Demonstrate how RESOLVE SAS language function obtains a macro variable value during execution of a DATA step. The argument to RESOLVE is a character expression that is based on a variable value in the observation currently being processed.
	9.13	GETSALENAME	GETSALENAME is called from a DATA step by the RESOLVE SAS language function. The call to GETSALENAME by RESOLVE returns a text string that is based on the value of a single positional parameter provided to GETSALENAME. The value passed to GETSALENAME is a variable value. The value that GETSALENAME returns is saved in a different data set variable.
	9.14		Save PROC SQL summarizations in macro variables using the INTO clause.
	9.15		Using PROC SQL and the INTO clause, save the first row of a table in macro variables.
	9.16		Create a macro variable for each row in a table using the PROC SQL INTO clause.
	9.17		Store all unique values of a data set variable in one macro variable using the PROC SQL INTO clause. Separate the values with a specific character.
	9.18		Store all values of a PROC SQL dictionary table column in one macro variable using the PROC SQL INTO clause. 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 macro variable values that were created in a PROC SQL step.
	9.20		Display the value of macro-related options by executing a PROC SQL step on DICTIONARY.OPTIONS.
	9.21		Access macro variable characteristics with PROC SQL and dictionary tables.
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 so that the macro program code is saved with the compiled macro program.
	10.3	REPTITLE	Modifies 10.1 by adding SECURE to %MACRO statement so that the compiled macro program code is encrypted.

---

---

<b>Chapter</b>	<b>Example</b>	<b>Macro Program Name</b>	<b>Function</b>
11	11.1	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.
		MULTCOND	Checks multiple characteristics of a data set and returns a return code value. This macro program is called by LISTSAMPLE.
	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 TABULATE step.
	11.3	RTF_START	Specify ODS characteristics, options, and TITLE and FOOTNOTE statements. Close the HTML 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.

---

---

<b>Chapter</b>	<b>Example</b>	<b>Macro Program Name</b>	<b>Function</b>
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.
	12.3	MARKUP	Include a specific RETAIN statement in a DATA step based on the values of three or four of the positional parameters. Define assignment statements with values from these three parameters. Subset the data set with the fourth parameter.
	12.4	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.
	12.5	MAKERTF	Called by macro program EXTFILES. Close the HTML destination, open the RTF destination, run PROC PRINT, close the RTF destination, and open the HTML destination.
		MAKESPREADSHEET	Called by EXTFILES. Submit PROC EXPORT steps to create Microsoft Excel worksheets.
		EXTFILES	Subset a data set by the value specified by one of the three positional parameters. The other two parameters specify whether an RTF PROC PRINT report and Microsoft Excel spreadsheets should be produced for the subset. Call MAKERTF and/or MAKESPREADSHEET based on these two parameter values.
	12.6	PROJCOST	Specify a PROC TABULATE report on a data set created by a DATA step preceding the call to PROJCOST. The PROC TABULATE analysis variables are specified in the single positional parameter, ANALYSISVARS.
	12.7	SELECTTITLES	Specify 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 not null. List contents of these automatic variables if not null.
	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.

---

---

<b>Chapter</b>	<b>Example</b>	<b>Macro Program Name</b>	<b>Function</b>
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 SGPlot 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, a PROC SQL step, and a PROC SGPlot step for a subset of a data set for specific classification variables and 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 153*t*  
\_ALL\_ option 43*t*  
ALL value 75–76  
ampersand (&) 7, 24, 60–61, 64–65, 136*t*, 137*t*, 180, 232  
AND (&) operator 155*t*  
arithmetic expressions 155–156  
ATTRC function 280  
ATTRIB statement 203, 228  
ATTRIBUTE variable 270–272  
ATTRN function 18, 145–147, 262–264, 280, 346*t*  
AUTHORREPORT macro program 303–306, 352*t*  
autocall facility  
  about 248  
  resolving macro program references when using 258–259  
  saving macro programs with 248–254  
  using under UNIX and z/OS systems 252–254  
autocall library  
  about 19  
  creating 248–250  
  identifying 251–252  
  making available to your programs 250–251  
autocall macro programs 147–150, 198–200, 251–252  
\_AUTOMATIC\_ option 43*t*  
AVERAGE macro variable 214–216

## B

Base SAS application 12*t*  
BOOLEAN conversion type 133–134*t*, 156  
BOX= option 106–108  
%BQUOTE function 136*t*, 178, 180–181, 187–188, 348*t*  
branching, in macro processing 174–176

## C

calculations, category of errors 277*t*  
CALL EXECUTE  
  about 202*t*, 216–226  
  calling macro programs with 220–223  
  calling specific macro programs with 223–226

timing of 217–218, 219–221  
CALL SYMDEL 202*t*  
CALL SYMPUT 202*t*  
CALL SYMPUTX 98, 202*t*, 213–214, 214–216  
catalog names, concatenating with macro variables 58–59  
CATALOG procedure 73, 255, 345*t*  
CATS function 223–226  
CEIL conversion type 134*t*  
character data, editing for comparisons 264–266  
character expressions, using resolution of as arguments 207–209  
CHECKSURVEY macro program 148–150, 346*t*  
CHECKVARNAMES macro program 143–144, 346*t*  
CLASS statement 352*t*  
CLASS\_STRING parameter 289–292  
CLASSVAR macro variable 289–292  
CLOSE language function 18  
CMD option 69*t*  
%CMPRES function 148*t*, 265–266  
%CMPRES macro program 148  
colon (: ) 44, 174, 232  
command-style macros 69*t*  
commas, preventing interpretation of as argument delimiters 183–184  
COMP2VARS macro program 158–160, 347*t*  
compiler, defined 22  
COMPRESS  
  editing data set characters 265–266  
  masking results from applying SAS language functions 199–200  
concatenating  
  catalog names with macro variables 58–59  
  permanent SAS data set names with macro variables 58–59  
concatenation (||) operators 154  
conditional iteration 170–174  
conditional processing 13–15, 158–167  
CONTENTS procedure 341  
%COPY statement 152*t*, 256, 257–258

## D

DAILY macro program 343*t*  
dash (-) 44

DATA= option, SGPLOT procedure 18  
 data set names  
   concatenating with macro variables 58–59  
   referencing 59  
 data sets  
   characteristics of 261–264, 269–272  
   sample 341  
   using macro programs to execute PROC steps on  
     multiple 5  
   using variables as arguments to SYMGET  
     function 204–206  
 DATA statement  
   iterative processing of SAS steps 15  
   in MAKESETS macro program 343*t*  
 DATASETS procedure  
   CALL EXECUTE routine 219–221  
   in DETAILED macro program 347*t*  
   in LISTLIBRARY macro program 349*t*  
   using %GOTO with 174–176  
 %DATATYP function 148*t*  
 date, formatting in TITLE statement with  
   %SYSFUNC function 142  
 DATE() function 346*t*  
 debugging SAS programs 280–281, 309–316  
 DES= option 69*t*, 256  
 DETAILED macro program 347*t*  
 dictionary tables, displaying macro option settings  
   with 241–243  
 DIFFRATE macro variable 286–289  
 %DISPLAY statement 154*t*  
 %DO loops  
   about 15  
   building PROC steps with iterative 167–169  
   building SAS statements within steps with  
     iterative 169–170  
   creating macro programs 325–328  
   iterative 153*t*  
   in LISTSQLPUB macro program 350*t*  
   in MULTREP macro program 347*t*  
   in SALES macro program 343*t*  
   SQL procedure 239–241  
   writing in macro language 167–170  
 %DO statement 153*t*, 156, 277*t*  
 %DO %UNTIL loop  
   errors and 289–292  
   in MOSALES macro program 347*t*  
   in TABLES macro program 352*t*

%DO %UNTIL statement 90, 156, 170–172  
 %DO %WHILE statement 153*t*, 156, 172–174, 347*t*  
 domains 110–114, 138–140  
 DSNAME parameter 145–147, 174–176, 262–264,  
   269–272  
 DSREPORT macro program 343*t*  
  
**E**  
 %ELSE statement 15, 158, 292–295, 294–295, 343*t*  
 %ELSE-%IF statement 287–289  
 encrypting stored compiled macro programs 258  
 %END statement 76, 153*t*, 277*t*, 284–298, 290–292  
 EQ (=) operator 155*t*  
 equal sign (=) 82  
 ERROR: option 43*t*, 44, 76–77  
 errors  
   about 273  
   categorizing 275–278  
   error checking 275–278, 298–306  
   examples in macro programming 281–298  
   minimizing in developing SAS programs that  
     contain macro language 274–275  
   tools for debugging macro programming 278–  
     281  
   types of 273–274  
 %EVAL function 133–135, 144–145, 154, 155–156,  
   277*t*, 285–289  
 execution error 274  
 existence, determining of 138–140  
 EXPORT procedure 352*t*  
 EXTFILES macro program 292–295, 352*t*

**F**  
 FACTS macro program 269–272, 351*t*  
 FCMP procedure 130  
 filerefs, defining under Windows 251–252  
 FINDPUBLISHER macro variable 232–234  
 FIRSTDATE macro variable 234–236  
 FIRSTPRICE macro variable 234–236  
 FIRSTTITLE macro variable 234–236  
 FLOOR conversion type 134*t*  
 FONT= attribute 191–193  
 FOOTNOTE statement  
   about 20, 344*t*

masking special characters and mnemonic operators 193–198  
 modifying and selecting statements 163–165  
 in MYPAGES macro program 348t  
 preventing resolution of special characters in macro variable values 189  
 in RTF\_END macro program 351t  
 in STANDARDOPTS macro program 343t

FREQ procedure  
 about 344t, 349t  
 creating macro variables with CALL SYMPUTX 213–214  
 displaying macro variables 46  
 placing text after macro variables 58  
 referencing permanent SAS data set names and macro variables 59  
 RESOLVE function 229  
 reviewing processing messages 301–303  
 TABLES statement 39–40  
 TITLE statement 39–40  
 functions 137–140, 137t, 261–266  
*See also specific functions*

**G**

GE (>) operator 155t  
 GETOPT macro program 346t  
 GETOPTION function 142–143, 346t  
 GETPUB parameter 283–298  
 GETSALENAMES macro program 350t  
 GETSECTION parameter 283–298  
 GLBSUBSET macro variable 106–108  
 global macro symbol tables 95, 97–101, 97f  
 global macro variables 98–101, 108–110  
 \_GLOBAL\_ option 43t  
 %GLOBAL statement 97, 106–108, 110–114, 152t, 345t  
 %GOTO statement 153t, 174, 275t, 280, 347t  
 GT (==) operator 155t

**H**

hard-coded programming constants, removing 317–320  
 HIGHREPORT macro program 223–226, 349t

**I**

%IF statement 76, 118, 123, 156, 157, 158, 211–212, 277t, 280–281, 294–295  
 %IF-%THEN statement 303–306, 325–328  
 %IF-%THEN-%DO statement 76  
 %IF-%THEN-%ELSE statement 90, 153t, 162–165, 166–167, 216–226, 347t  
 IN operator 154, 157, 166–167  
 IN (#) operator 155t  
 %INDEX function 130t  
 input stack 22, 116  
 %INPUT statement 154t  
 integer arithmetic 134  
 INTEGER conversion type 134t  
 interfaces  
 about 201–202  
 CALL EXECUTE routine 216–226  
 displaying macro option settings with SQL procedure and dictionary tables 241–243  
 macro variables created by SQL procedure 238–241  
 RESOLVE function 227–231  
 SYMGET function 202–209  
 SYMPUT call routine 209–216  
 SYMPUTX call routine 209–216  
 using macro facility features in SQL procedure 231–243  
 INTO clause 18, 98, 232–238, 239–241, 350t  
 iterations, conditional 170–174  
 iterative processing 15–16, 167–174

**J**

JCLDD statement 253–254

**K**

KEYPARAM macro program 345t  
 KEYTEXT parameter 132–133  
 keyword parameters 80, 82–89

**L**

%label: statement 153t  
 LASTMSG macro program 301–303, 352t  
 LASTUPDATE macro variable 145–147

LE ( $\leq$ ) operator 155*t*  
 leading blanks 184–185  
 LEE macro variable 189  
 %LEFT function 148*t*  
 %LEFT macro program 148  
 %LENGTH function 130*t*  
 LENGTH statement 203, 228  
 %LET statement  
   about 136, 152*t*, 344*t*  
   creating macro programs 317–320  
   creating macro variables 52–55  
   data set characteristics 262–264  
   in error checking 277*t*  
   error tracing 287–289  
   masking results from applying SAS language functions 199–200  
   naming macro variables 111–114  
   placing text before macro variable references 56  
   program processing 28, 31  
   quoting functions 179–180, 183–184, 184–185,  
     185–186, 186–187  
   referencing macro variables 39–40, 106–108  
   SQL procedure 242–243  
   SYMGET function 204–206, 207–209  
   using 149–150  
   using %IF-%THEN-%ELSE statement to modify  
     and select statements 163–165  
 library of utilities  
   about 261  
   building and saving routines 19  
   programming routine tasks 266–272  
   writing macro programs to behave like functions 261–266  
 LIBRARY= option 257–258  
 LISTAUTOMATIC macro program 349*t*  
 LISTINDEX macro variable 170–172  
 LISTLIBRARY macro program 219–221, 349*t*  
 LISTPARM macro program 345*t*  
 LISTSAMPLE macro program 262–264, 351*t*  
 LISTSQLPUB macro program 350*t*  
 LISTTEXT macro program 132–133, 346*t*  
 literal token 24  
 local macro symbol tables 96, 101–108  
 local macro variables 108–110  
   \_LOCAL\_ option 43*t*  
 %LOCAL statement 110–114, 153*t*, 270–272  
 LOCLMVAR macro program 345*t*

LOCLMVAT macro program 111–114  
 logical expressions 156, 158–160, 278*t*  
 %LOWCASE function 148*t*  
 %LOWCASE macro program 148  
 LOWREPORT macro program 223–226, 349*t*  
 LT ( $<$ ) operator 155*t*

## M

MAC macro variable 214–216  
 macro activity, SAS processing without 22–23  
 macro character functions 130–133, 198–200  
 macro evaluation functions 133–135  
 macro expressions  
   about 154–155  
   arithmetic 155–156  
   constructing 154–157  
   logical 156  
   IN operator 157  
 %MACRO function 254–258, 277*t*  
 macro functions  
   category of errors 277*t*  
   debugging macro programming with 280  
 %MACRO KEYPARM statement 83–86  
 macro language  
   conditional processing with 158–167  
   interfacing with SAS language functions 18–19  
   iterative processing with 167–174  
   processing SAS programs that contain 27–36  
   SAS language vs. 277*t*  
   selecting SAS steps for processing with 160–162  
   writing iterative %DO loops in 167–170  
 macro language functions  
   about 129–130  
   autocall macro programs 147–150  
   macro character functions 130–133  
   macro evaluation functions 133–135  
   macro quoting functions 135–137  
   macro variable attribute functions 137–140  
   %QSYSFUNC 141–147  
   %SYSFUNC 141–147  
 macro language processing, compared with SAS language processing 26–27  
 macro language statements  
   See also statements  
   about 151–154  
   compared with language statements 52

- debugging macro programming with 279–280
- types of 152
- used in open code or inside macro programs 152–153*t*
- used only inside macro programs 153–154*t*
- used to display windows and prompt users for input 154*t*
- using IN operator in 157
- MACRO option 279*t*, 281–298
- macro parameters 79–93, 277*t*
- macro processing 21, 174–176
  - See also* processing
- macro processor 22, 109*f*, 110*f*, 117*f*, 118
- macro program resolution category of errors 276*t*
- macro programming
  - error examples in 281–298
  - tools for debugging 278–281
- macro programs
  - See also specific macro programs*
  - about 67–68, 247–248
  - autocall 147–150, 198–200, 251–252
  - calling with CALL EXECUTE routine 220–223, 223–226
  - calling with RESOLVE function 230–231
  - compiling 114–119
  - creating 68–73, 255–256
  - defining that accept varying number of parameter values 89–93
  - defining with keyword parameters 83–86, 86–89
  - defining with PARMBUFF option 90–93
  - defining with positional parameters 80–82, 86–89
  - demonstrating 71–73
  - displaying messages about processing of in SAS Log 77–79
  - displaying notes about compilation of in SAS Log 75–77
  - executing 74–75, 120–128
  - generalizing 325–328
  - passing values to through macro parameters 79–93
  - processing 114–128
  - referencing global macro variables in open code and from within 98–101
  - referencing macro variables defined as parameters to 102–108
  - refining 325–328
- resolving references when using autocall facility 258–259
- saving with autocall facility 248–254
- saving with stored compiled macro facility 254–258
- specifying parameters in 80–89, 189–198
- in this book 343–353
- tracing execution of with MLOGIC option 78–79
- using to execute PROC steps on multiple data sets 5
- writing 261–266, 307–338
- macro quoting functions 135–137, 136–137*t*
- %MACRO statement
  - about 68, 68*t*, 152*t*
  - building SAS steps with %DO %UNTIL loops 170–172
  - category of errors 277*t*
  - creating stored compiled macro programs 255–256
  - encrypting stored compiled macro programs 258
  - macro parameter names specified on 79
  - IN operator 157
  - in REPTITLE macro program 350*t*
  - saving and retrieving source code of stored compiled macro programs 257–258
  - specifying keyword parameters in macro programs 82
  - specifying positional parameters 80–82
- macro symbol tables 22, 95–97
- macro triggers 22, 24, 185–186, 187
- macro variable attribute functions 137–140, 137*t*
- macro variable resolution category of errors 276*t*
- macro variable values
  - applying SAS statistical functions to with %SYSFUNC function 144–145
  - converting to uppercase with %UPCASE 132–133
  - extracting Nth item from with %SCAN 131–132
  - extracting text from with %SUBSTR 131
  - obtaining with RESOLVE function 228–229
  - preventing resolution of special characters in 188–189
  - retrieving with SYMGET function 206–207
  - submitting %PUT statement to display 45–46
- macro variables
  - See also specific variables*

- about 37, 38
- assigning results to 142–143
- automatic 48–51
- basic concepts of 38–39
- combining with text 55–56
- concatenating catalog names with 58–59
- concatenating permanent data set names with 58–59
- created by SQL procedure 238–241
- creating with CALL SYMPUTX 213–214, 214–216
- creating with INTO clause 236–237
- creating with %LET statement 52–55
- creating with SQL procedure 232–238
- debugging macro programming with 280–281
- defining 6–11
- defining domains of 110–114
- determining domain and existence of with macro variable attribute functions 138–140
- displaying values 42–48
- global 98–101, 108–110
- illustrating resolution of references when combining them 61–63
- naming 111–114
- referencing 39–41, 59, 60–65, 102–108
- replacing hard-coded programming constants with 317–320
- resolution and use of quotation marks 41–42
- resolving multiple ampersands preceding 64–65
- saving summarizations in 232–234
- selecting observations to process with 5
- storing all values in 237–238
- storing unique values in 237
- updating with SQL procedure 232–238
- user-defined 51–55
  - using 6–11
- MAKEDS macro program 98–101, 102–108, 345*t*, 346*t*
- MAKERTF macro program 292–295, 352*t*
- MAKESETS macro program 343*t*
- MAKESPREADHSEET macro program 292–295, 352*t*
- MAR macro program 198, 348*t*
- MARKUP macro program 285–289
- masking
  - macro triggers 187
  - mnemonic operators 193–198
- special characters 190–191, 193–198
- MAUTOSOURCE option 250–251, 253–254
- MAX operator 154
- MCOMPILENOTE option 75–77
- MEAN function 144–145
- MEANS procedure
  - about 344*t*
  - building SAS steps with %DO %UNTIL loops 170–172
  - building SAS steps with %DO %WHILE loops 173–174
  - CALL EXECUTE routine 220–223, 223–226
  - CALL SYMPUTX 211–212, 214–216
  - conditional processing of SAS steps 13–15
  - defining macro programs with keyword parameters 85
  - defining macro programs with positional parameters 80–82
  - defining macro variables 40
  - extracting text from macro variable values with %SUBSTR 131
  - interfacing macro language and SAS language functions 18–19
  - in LISTPARM macro program 345*t*
  - in MOSALES macro program 347*t*
  - OTHROPTS parameter 86–89
  - preventing resolution of special characters in macro variable values 188–189
  - referencing macro variables 40
  - RESOLVE function 230–231
  - in SALES macro program 343*t*
  - STATS parameter 86–89
  - in STATSECTION macro program 349*t*
  - TITLE statement 41–42
  - WHERE statement 64, 131
- %MEND statement 68, 70*t*, 127, 152*t*, 277*t*, 283–298
- MERROR option 276*t*, 279*t*, 281–298
- messages, displaying about macro program processing in SAS Log 77–79
- MIN operator 154
- MIN variable 214–216
- MINDELIMITER option 69*t*, 157, 166–167
- MINOPERATOR option 69*t*, 154
- MINSALEPRICE parameter 299–301
- MISSING option 87
- MIXDPARM macro program 345*t*
- MLOGIC option

creating macro programs 328  
 in debugging macro programming 278, 278*t*  
 displaying messages 77  
 errors and 292–295  
 executing REPORT macro program 329–331,  
   331–335, 335–338  
 tracing errors in expression evaluation with 285–  
   289  
 tracing execution of macro programs with 78–79  
**m**  
 mnemonic operators  
   about 177–178  
   category of errors 277*t*  
   masking 193–198  
   preventing interpretation of 187–188  
   specifying parameters that contain 189–198  
 MONTHS macro variable 131–132  
 MONTHSOLD parameter 299–301  
 MOSALES macro program 170–172, 347*t*  
 MOSECTDETAIL macro program 190–191, 348*t*  
 MOYRSALES macro program 172–174, 347*t*  
**MPRINT** option  
   CALL EXECUTE routine 221–223  
   calling macro programs with CALL EXECUTE  
     224–226  
   creating macro programs 321–325, 328  
   in debugging macro programming 278, 279*t*  
   displaying messages 77  
   displaying SAS statements with 77–78  
   errors and 290–292, 296–298  
   executing REPORT macro program 329–338  
 MSTORED option 254–255  
 MULTCOND macro program 262–264, 351*t*  
 MULTREP macro program 167–169, 347*t*  
 MVS Batch, using autocall facilities under 253–254  
 MYPAGES macro program 193–198, 348*t*  
 MYREPORT macro program 118, 119*f*, 120–128,  
   122*f*, 127  
 MYSETTING macro variable 242–243  
 MYSQLSTEP macro variable 182–183

**N**

NAMES macro variable 186–187  
 names token 24  
 naming macro variables 39, 111–114  
 NE ( $\neq$ ) operator 154, 155*t*  
 NOAUTOCALL value 75–76

NOMINOPERATOR option 69*t*  
 NOMPRINT option 296–298  
 NONE value 75–76  
 NOSECURE option 70*t*  
 NOT ( $\neg$ ) operator 154, 155*t*  
 NOTE: option 43*t*, 44  
 notes, displaying about macro program compilation in  
   SAS Log 75–77  
 NOTNAME function 143–144  
 %NRBQUOTE function 136*t*, 181  
 %NRQUOTE function 136*t*, 181  
 %NRSTR function  
   about 135–136, 136*t*, 178, 180–181, 348*t*  
   masking macro triggers with 185–186  
   masking results from applying SAS language  
     functions 199–200  
   masking special characters 190–191  
 NSOLD macro variable 232–234  
 numbers token 24  
 numeric data set variables, creating with SYMGET  
   function 206–207  
 NUMOBS= option 174–176  
 NVALID function 143–144

**O**

open code  
   macro variables in 38  
   referencing global macro variables in 98–101  
 OPEN language function 18  
 options 278–279  
   *See also specific options*  
 OPTIONS statement 20, 157, 166–167, 250–252,  
   254, 343*t*  
 </option(s)> element 69*t*, 70*t*  
 OPTS parameter 80–82, 83–86  
 OPTVALUE macro variable 142–143  
 OR () operator 155*t*  
 OTHROPTS parameter 86–89

**P**

parameter values  
   evaluating 299–301  
   masking special characters and mnemonic  
     operators in 193–198

preventing misinterpretation of special characters  
in 191–193

<parameter-list> option, %MACRO element 68*t*

parameters  
*See also specific parameters*  
keyword 80, 82–89  
referencing macro variables defined as  
parameters to macro programs 102–108  
specifying 189–198

PARMBUFF option 69*t*, 80, 89–93, 170–172, 345*t*, 347*t*

PBUFF option 69*t*

PBUFFARMS macro program 345*t*

percent sign (%) 7, 24, 136*t*, 137*t*, 174, 180, 186–187

periods (.) 55

positional parameters 80–82, 86–89

PREFIX macro variable 57

PRINT procedure  
about 5  
in AUTHORITY macro program 352*t*  
INTO clause 234–236  
compiling macro programs 118  
data set characteristics 262, 264  
in DETAIL macro program 347*t*  
documenting data set characteristics 269–272  
evaluating parameter values 299–301  
executing macro programs 125, 126*f*, 127  
in LISTSAMPLE macro program 351*t*  
in LISTTEXT macro program 346*t*  
in MAKERTF macro program 352*t*  
masking special characters 190–191  
in MOSECTDETAIL macro program 348*t*  
quoting functions 179–180  
SYMGET function 205–206  
using automatic variables 50–51  
using %GOTO with 174–176  
using macro language to select SAS steps for  
processing 160–162

PRINT10 macro program 281–298, 352*t*

processing  
conditional 13–15, 158–167  
iterative 15–16, 167–174  
macro programs 114–128  
reviewing messages 301–303  
SAS programs that contain macro language 27–36  
vocabulary of 21–22

without macro activity 22–23

PROFITCHART macro program 345*t*

program option, %MACRO element 68*t*

program steps, passing information between 16–18

programming routine tasks 266–272

PROJCOST macro program 352*t*

PUBLISHER macro variable 136

PUBLISHER parameter 166–167, 299–301

PUBLISHERREPORT macro program 162–165, 347*t*

PUBLISHERSALES macro program 191–193, 348*t*

punctuation, category of errors 275*t*

purpose, of macro variables 39

PUT function 211–212

%PUT LOCAL statement 214–216

%PUT statement  
about 152*t*, 346*t*  
applying quoting functions 182–189  
CALL EXECUTE routine 219–221  
in CHECKVARNAME macro program 346*t*  
in COMP2VAR macro program 347*t*  
debugging macro programming 279–280  
displaying macro variable values with 43–46  
displaying macro variables 42  
%EVAL evaluation function examples 134–135*t*  
in LISTAUTOMATIC macro program 349*t*  
masking results from applying SAS language  
functions 199–200  
quoting functions 183–186  
reviewing processing messages 302–303  
SQL procedure 232–234, 237  
submitting to display text and macro variable  
values 45–46  
%SYSEVALF evaluation function examples  
134–135*t*  
tracing errors in expression evaluation with 285–289  
tracing problems at execution 287–289  
using 142–143, 144–145, 148–150  
using with %LET statement 52–54

%PUT\_AUTOMATIC\_ statement 44, 217–218

%PUT\_GLOBAL\_ statement 45

PUTN function 18, 145–147

**Q**

%QCMPRES function 178  
%QLEFT function 178

%QLOWCASE function 178  
 %QSCAN function 130, 156, 178, 186–187, 348*t*  
 %QSUBSTR function 130, 156, 178, 200, 348*t*  
 %QSYSFUNC function 141–147, 141*f*, 178, 199–  
 200, 280, 348*t*  
 %QTRIM function 178  
 quotation marks 24, 41–42, 186–187  
 %QUOTE function 136*t*, 181  
 quoting functions  
     about 178  
     applying 182–189  
     commonly used 180–181  
     how they work 181–182  
     macro 135–137, 136–137*t*  
     need for 179–180  
 quoting versions, using of macro character functions  
     and autocall macro programs 198–200  
 %QUPCASE function 130, 178, 265–266

## R

REP16K macro program 220–223, 349*t*  
 REPGRP macro variable 28, 31, 33, 34  
 REPMONTH parameter 172–174  
 REPORT macro program  
     about 353*t*  
     enhancing 338  
     executing 328–338  
     in PUBLISHERSALES macro program 348*t*  
     in PUBLISHREPORT macro program 347*t*  
     refining and generalizing macro programs 326–  
         328  
     in REP16K macro program 349*t*  
 REPORT procedure  
     CALL EXECUTE routine 220–223, 223–226  
     preventing misinterpretation of special characters  
         191–193  
     in RTF\_START macro program 351*t*  
     standardizing RTF output 266–269, 267–269  
     using %IF-%THEN-%ELSE statement to modify  
         and select statements 162–165  
 REPORTA macro program 324–325, 353*t*  
 REPORTB macro program 324–325, 353*t*  
 REPORTC macro program 353*t*  
 REPORTS macro program 160–162, 347*t*  
 REPORTTITLE macro variable 185–186  
 REPTITLE macro program 39–40, 350*t*

REPTYPE parameter 160–162, 162–165  
 REPVAR macro variable 39–40  
 RESOLVE function  
     about 202*t*, 227–231  
     ATTRIB statement 228  
     in error checking 276*t*  
     FREQ procedure 229  
     in GETSALENAME macro program 350*t*  
     LENGTH statement 228  
     obtaining macro variable values with 228–229  
     using to call macro programs within DATA steps  
         230–231  
 RESPONSE parameter 148–150, 346*t*  
 RETAIN statement 352*t*  
 %RETURN statement 154*t*  
 RTF output, standardizing 266–269  
 RTF\_END macro program 267–269, 351*t*  
 RTF\_START macro program 267–269, 351*t*  
 RUN statement 26, 35

## S

SALES macro program 343*t*  
 SAS catalogs, identifying autocall libraries stored in  
     252  
 SAS Component Language application 12*t*  
 SAS language  
     functions 18–19, 142–143  
     macro language vs. 277*t*  
     processing, compared with macro language  
         processing 26–27  
 SAS Log  
     displaying messages about macro program  
         processing in 77–79  
     displaying notes about macro program  
         compilation in 75–77  
 SAS macro facility  
     about 4–5  
     advantages of 5–11  
     examples of 12–20  
     where it can be used 11–12  
*SAS Macro Language: Reference* 39, 48, 68, 68*t*, 69*t*,  
 70*t*, 152  
 SAS programs  
     debugging 309–316  
     processing 27–36  
     testing 309–316

- tokenizing 24–26
- writing 309–316
- SAS steps
  - conditional processing of 13–15
  - iterative processing of 15–16
- SASAUTO= option
  - about 20
  - autocall libraries 251
  - maintaining access to autocall macro programs 251–252
- SAS/CONNECT application 12*t*
- SAS/GRAFH application 12*t*
- SASMSTORE option 254–255
- %SCAN function
  - about 130, 130–131*t*, 156, 346*t*
  - building SAS steps with %DO %UNTIL loops 170–172
  - creating macro programs 318–320
  - defining macro programs with PARMBUFF option 90
  - errors and 289–292
  - extracting Nth item from macro variable values 131–132
  - quoting functions 186–187
- SECTION parameter 120–128, 214–216, 220–223
- SECURE option 70*t*, 256, 258
- SELECT statement 98, 232–241, 310–316
- SELECTTITLES macro program 299–301, 352*t*
- semicolon, preventing interpretation of as a SAS statement terminator 182–183
- SENTENCE macro variable 54
- SERROR option 276*t*, 279*t*, 281–298
- SET statement 284–298
- SGPANEL procedure 343*t*
- SGPLOT procedure
  - building PROC steps with iterative %DO loops 167–169
  - creating macro programs 310–316, 318–320, 325–328
- DATA= option 18
- DATA step 74–75
- defining macro programs with PARMBUFF option 90–93
- enhancing REPORT macro program 338
- executing REPORT macro program 331–335
- in MULTREP macro program 347*t*
- in PROFITCHART macro program 345*t*
- in REPORTB macro program 353*t*
- TITLE statement 16–18
- source code 257–258
- SOURCE option
  - creating stored compiled macro programs 255–256
  - </option(s)> element 70*t*
  - saving and retrieving source code of stored compiled macro programs 257–258
- special characters
  - about 177–178
  - category of errors 277*t*
  - masking 190–191, 193–198
  - preventing misinterpretation of 191–193
  - preventing resolution of in macro variable values 188–189
  - specifying parameters that contain 189–198
- special token 24
- SQL procedure
  - about 50, 348*t*, 350*t*
  - in AUTHORREPORT macro program 352*t*
  - computing sum of scales with 18
  - creating macro programs 310–316, 318–320, 326–328
  - creating macro variables with 232–238
  - default action of INTO clause 234–236
  - displaying macro option settings with 241–243
  - documenting data set characteristics 270–272
  - executing REPORT macro program 331–335
  - %LET statement 242–243
  - macro variables created by 238–241
  - %PUT statement 232–234, 237
  - quoting functions 182–183
  - in REPORT macro program 353*t*
  - saving summarizations in macro variables 232–234
- SELECT statement 98, 232–241
- SQLOBS automatic macro variable 239–241
- storing all values in macro variables 237–238
- storing unique values in macro variables with 237
- updating macro variables with 232–238
- using INTO clause to create macro variables 236–237
- using macro facility features in 231–243
- SQLEXITCODE macro variable 238*t*
- SQLOBS macro variable 238*t*, 239–241

- SQLOOP macro variable 238  
 SQLRC macro variable 238  
 SQLXMSG macro variable 238, 239t  
 SQLXRC macro variable 238, 239t  
 SRC option 70t  
 standardizing RTF output 266–269  
 STANDARDOPTS macro program 343t  
 START parameter 80–82, 83–86, 86–89, 326–328  
 STATE macro variable 187–188  
 statements  
*See also specific statements*  
 compared with SAS macro language statement 52  
 displaying with MPRINT option 77–78  
 modifying 162–165  
 selecting 162–165  
 transferring from input stack to word scanner 116  
 STATS parameter 86–89  
 STATSECTION macro program 214–216, 349t  
 STMT option 69t  
 STOP parameter 80–82, 83–86, 86–89, 326–328  
 STORE option 70t, 257–258  
 stored compiled facility 248  
 stored compiled macro programs 257–259  
 %STR function  
     about 136t, 178, 180–181, 348t  
     masking special characters 190–191  
     masking unbalanced quotation marks and preceding percent signs 186–187  
     preserving leading and trailing blanks 184–185  
     preventing interpretation of commas as argument delimiters 183–184  
     preventing interpretation of semicolon as SAS statement terminator with 182–183  
     quoting functions 185–186  
 STYLE parameter 191–193  
 STYLEHEADER parameter 191–193  
 STYLEREPORT parameter 191–193  
 SUBSET macro variable 98–101, 102–108, 111–114  
 %SUBSTR function 130, 131, 131t, 156, 183–184, 200, 346t, 348t  
 SUMSOFTPROG macro variable 18  
 SUMYEARS macro program 169–170, 347t  
 %SUPERQ function 136t, 137t, 178, 180–181, 188–189, 265–266, 348t  
 SURVLIB macro variable 59  
 %%SYMBLOBL macro variable 214–216  
 symbol tables, specifying 214–216  
 SYMBOLGEN option  
     about 344t  
     creating macro programs 321–325, 328  
     debugging macro programming 278, 279t, 280  
     displaying macro variables 42  
     displaying messages 77  
     enabling to display macro variable values 46–48  
     quoting functions 182  
     resolving multiple ampersands that precede macro variables 64  
     tracing resolution of indirect macro variable references 63  
 %SYMDEL statement 102–108, 138–140, 152t, 346t  
 SYMEXIST tool 202t  
 %SYMEXIST function 137t, 280  
 SYMGET function  
     about 202–209, 202t, 349t  
     creating numeric data set variables with 206–207  
     retrieving macro variables with 206–207  
     similarity to RESOLVE function 227  
     using data set variables as arguments to 204–206  
     using resolution of character expressions as arguments to 207–209  
 %SYMGLOBL function 137t, 280, 346t  
 %SYMLOCAL function 137t, 202t, 280, 346t  
 %SYMLOCAL macro variable 214–216  
 SYMPUT call routine 209–216  
 SYMPUTX call routine 16–18, 209–216  
 syntax error 274  
 %SYSCALL statement 152t  
 SYSDATE macro variable 49t, 131  
 SYSDATE9 macro variable 49t, 97–101  
 SYSDAY macro variable 39–40, 49t, 97–101, 125, 343t  
 SYSDEN macro variable 49t  
 SYSERR macro variable 49t  
 SYSERRORTEXT macro variable 49t, 301–303, 352t  
 %SYSEVALF function  
     about 133–135, 154, 155–156, 346t  
     in error checking 277t  
     error tracing 285–289  
     preventing interpretation of mnemonic operators 187–188  
     using 144–145  
 %SYSEXEC statement 153t

- SYSFILRC macro variable 49*t*  
**%SYSFUNC** function  
 about 141–142, 346*t*  
 applying SAS language functions 145–147  
 applying SAS statistical functions 144–145  
 assigning results to macro variables 142–143  
 data set characteristic 262–264  
 debugging macro programming 280  
 determining if values are valid SAS variable names with 143–144  
 in DSREPORT macro program 343*t*  
 executing SAS language functions with 142–143  
 formatting date in TITLE statement with 142  
 masking results from applying SAS language functions 199–200  
 using 141–142, 141–147, 141*f*, 149–150  
**%SYSGET** function 141*f*  
**SYSLAST** macro variable 49*t*  
**SYSLIBRC** macro variable 49*t*  
**%SYSPUT** statement 153*t*  
**%SYSMACDELETE** statement 73  
**SYSMACRONAME** macro variable 49*t*  
**SYSPBUFF** macro variable 89–93  
**%SYSPROD** function 141*f*  
**%SYSRPUT** statement 153*t*  
 system information, displaying 12–13  
 system option settings category of errors 276*t*  
**SYSTIME** macro variable 49*t*  
**SYSVER** macro variable 49*t*, 97–101  
**SYSWARNINGTEXT** macro variable 49*t*, 301–303, 352*t*
- T**
- TABLES** macro program 289–292, 352*t*  
**TABLES** statement  
 errors and 296–298  
**FREQ** procedure 39–40  
 placing text after macro variable references 57–58  
 in TABLES macro program 352*t*  
**TABULATE** procedure  
 about 3, 343*t*  
 in AUTHORREPORT macro program 352*t*  
 building PROC steps with iterative %DO loops 167–169  
 editing data set characters 264–266
- enhancing REPORT macro program 338  
 errors and 289–292, 296–298  
 in MAKEDS macro program 345*t*  
 in PROJCOST macro program 352*t*  
 referencing macro variables defined as  
   parameters to macro programs 103–108  
 in REPORTA macro program 353*t*  
 in REPORTS macro program 347*t*  
 in TABLES macro program 352*t*  
 in TRIMNAME macro program 351*t*  
 using macro language to select SAS steps for processing 160–162  
 testing SAS programs 309–316  
**text**  
 combining macro variables with 55–56  
 extracting from macro variable values with  
   %SUBSTR 131  
 placing after macro variable references 57–58  
 placing before macro variable references 56  
 unmasking 198  
**text values**  
 macro variables as 38–39  
 submitting %PUT statement to display 45–46  
**<text>** element 70*t*  
 timing, of CALL EXECUTE routine 217–218, 219–221  
**TITLE** statement  
 about 3, 133, 343*t*, 344*t*, 346*t*  
 building SAS steps with %DO %UNTIL loops 171–172  
**CALL SYMPUTX** 214–216  
 formatting date with %SYSFUNC function in 142  
**FREQ** procedure 39–40  
 masking special characters and mnemonic operators 193–198  
**MEANS** procedure 41–42  
 in MYPAGES macro program 348*t*  
 passing information between program steps 16–18  
 in REPTITLE macro program 350*t*  
 resolving macro variables enclosed in quotation marks 41–42  
 in RTF\_END macro program 351*t*  
 in STANDARDOPTS macro program 343*t*  
 in STATSECTION macro program 349*t*  
 using %UNQUOTE function 198

in VENDORTITLES macro program 347*t*  
 TITLE1 statement 20  
 TITLE2 statement 166–167  
 tokenizing SAS programs 24–26  
 tokens  
   about 23–24  
   defined 22  
   transferring to macro processor 117*f*, 118  
 tools, for debugging macro programming 278–281  
 TOTSALES macro variable 232–234  
 trailing blanks 184–185  
 %TRIM function 148*t*  
 TRIMNAME macro program 264–266, 351*t*

**U**

UNIX 20, 252–254  
 unmasking text 198  
 %UNQUOTE function 137*t*, 178, 198, 348*t*  
 %UPCASE function 130, 131*t*, 132–133, 148–150  
 %UPCASE macro program 346*t*  
 uppercase, converting macro variable values to with  
   %UPCASE 132–133  
 \_USER\_ option 43*t*  
 user-defined macro variables 38, 51

**V**

VALIDRESPONSES macro variable 149  
 VALUE variable 270–272  
 values  
   determining if they are valid SAS variable names 143–144  
   macro variable 42–48  
   passing to macro programs through macro parameters 79–93  
 VAR statement 179–180  
 variables  
   See *specific variables*  
 VARLIST parameter 174–176  
 VARNUM macro variable 290–292  
 VARS= parameter 326–328  
 VENDORTITLES macro program 166–167, 347*t*  
 %VERIFY function 148–150, 148*t*, 346*t*

**W**

WARNING: option 43*t*, 44  
 WHERE statement  
   about 133, 343*t*, 344*t*, 346*t*  
   building SAS steps with %DO %UNTIL loops 170–172  
   editing data set characters 264–266  
   errors and 283–298  
   evaluating parameter values 299–301  
   MEANS procedure 64, 131  
   in WHSTMT macro program 352*t*  
 WHEREVAR macro variable 64  
 WHSTMT macro program 283–298, 284–298, 352*t*  
 %WINDOW statement 154*t*  
 WONTWORK macro variable 179–180  
 word scanner 22, 26, 116

**Y**

YEARIDX macro variable 173–174  
 YEARLIST parameter 172–174  
 YEARSTRING macro variable 169–170

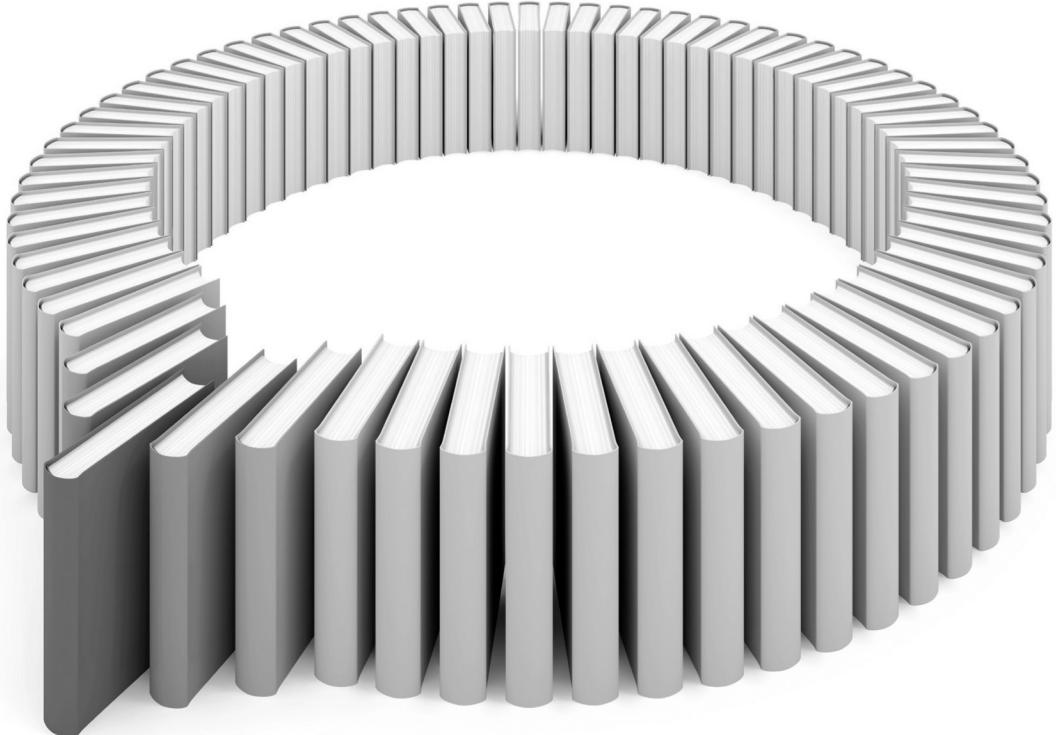
**Z**

z/OS systems, using autocall facilities under 252–254

**Symbols**

%\* comment; statement 152*t*  
 - operator 155*t*  
 / operator 155*t*  
 \* operator 155*t*  
 \*\* operator 155*t*  
 ^ operator 155*t*  
 + operator 155*t*  
 ~ operator 155*t*  
 || (concatenation) operators 154  
 >= operator 155*t*  
 & (ampersand) 7, 24, 60–61, 64–65, 136*t*, 137*t*, 180, 232  
 & (AND) operator 155*t*  
 : (colon) 44, 174, 232  
 - (dash) 44  
 = (EQ) operator 155*t*  
 = (equal sign) 82  
 > (GE) operator 155*t*

$\sim=$  (GT) operator 155*t*  
 $\#$  (IN) operator 155*t*  
 $\leq=$  (LE) operator 155*t*  
 $<$  (LT) operator 155*t*  
 $\hat{=}$  (NE) operator 155*t*  
 $\neg$  NOT operator 154, 155*t*  
 $|$  (OR) operator 155*t*  
 $\%$  (percent sign) 7, 24, 136*t*, 137*t*, 174, 180, 186–187  
. (periods) 55



# Gain Greater Insight into Your SAS® Software with SAS Books.

Discover all that you need on your journey to knowledge and empowerment.



[support.sas.com/bookstore](http://support.sas.com/bookstore)  
for additional books and resources.



