

# **Carpenter's Complete Guide to the SAS Macro Language**

**Third Edition**

**Art Carpenter**



Carpenter, Art. 2016. *Carpenter's Complete Guide to the SAS® Macro Language Third Edition*. Cary, NC: SAS Institute Inc.

**Carpenter's Complete Guide to the SAS® Macro Language, Third Edition**

Copyright © 2016, SAS Institute Inc., Cary, NC, USA

ISBN 978-1-62959-268-8 (Hard copy)

ISBN 978-1-62960-237-0 (EPUB)

ISBN 978-1-62960-238-7 (MOBI)

ISBN 978-1-62960-239-4 (PDF)

Produced in the United States of America.

August 2016

# Contents

<b>About This Book .....</b>	
<b>Part 1: Macro Basics.....</b>	<b>1</b>
<b>Chapter 1: What the Language Is, What It Does, and What It Can Do .....</b>	<b>3</b>
1.1 Introduction .....	3
1.2 Stages of Macro Language Learning.....	5
1.2.1 Code Substitution.....	5
1.2.2 Macro Language Elements.....	6
1.2.3 Dynamic Programming .....	7
1.3 Terminology.....	7
1.4 Sequencing Events—It's All about the Timing.....	8
1.5 Scopes or Referencing Environments .....	12
1.5.1 Use of Symbol Tables .....	12
1.5.2 Nested Symbol Tables .....	13
<b>Chapter 2: Defining and Using Macro Variables.....</b>	<b>17</b>
2.1 Naming Macro Variables.....	18
2.2 Defining Macro Variables .....	18
2.3 Using Macro Variables .....	19
2.4 Displaying Macro Variables by Using the %PUT Statement.....	21
2.5 Resolving Macro Variables .....	24
2.5.1 Using the Macro Variable as a Suffix .....	25
2.5.2 Using the Macro Variable as a Prefix .....	26
2.5.3 Using Macro Variables as Building Blocks—Appending Macro Variables.....	27
2.5.4 Understanding Results When Macro References Are Not Resolved.....	28
2.6 Using Automatic Macro Variables.....	29
2.6.1 &SYSDATE, &SYSDATE9, &SYSDAY, and &SYSTIME .....	29
2.6.2 &SYSLAST and &SYSDSN .....	30
2.6.3 &SYSERR and &SYSCC .....	31
2.6.4 &SYSRC .....	32
2.6.5 &SYSSITE, &SYSSCP, &SYSSCPL, and &SYSUSERID .....	32
2.6.6 &SYSMACRONAME.....	33
2.7 Removing Macro Variables .....	33
2.8 Testing Your Knowledge with Chapter Exercises .....	33

<b>Chapter 3: Defining and Using Macros .....</b>	<b>35</b>
3.1 Creating a Macro .....	35
3.1.1 Defining a Macro .....	37
3.1.2 Commenting a Block of Code with Use of %MACRO and %MEND .....	37
3.1.3 Using the /DES Macro Statement Option .....	39
3.2 Invoking a Macro.....	39
3.3 Using System Options with the Macro Facility .....	40
3.3.1 General Macro Options.....	41
3.3.2 Debugging Options .....	41
3.3.3 Use of the Debugging Options.....	41
3.3.4 Autocall Facility Options.....	42
3.4 Testing Your Knowledge with Chapter Exercises.....	44
<b>Chapter 4: Using Macro Parameters.....</b>	<b>45</b>
4.1 Introducing Macro Parameters .....	45
4.2 Using Positional Parameters .....	46
4.2.1 Defining the Macro's Parameters .....	46
4.2.2 Passing Parameter Values into the Macro .....	46
4.3 Using Keyword Parameters .....	48
4.3.1 Defining the Parameters and Their Default Values.....	48
4.3.2 Passing Parameter Values When Calling the Macro .....	48
4.3.3 Documenting Your Macro .....	49
4.4 Choosing between Keyword and Positional Parameters .....	50
4.4.1 Selecting Parameter Types .....	50
4.4.2 Using Keyword and Positional Parameters Together .....	50
4.4.3 Naming Keyword Parameters without the Equal Sign .....	51
4.5 Testing Your Knowledge with Chapter Exercises.....	51
<b>Part 2: Using Macros .....</b>	<b>53</b>
<b>Chapter 5: Controlling Programs with Macros .....</b>	<b>55</b>
5.1 Macros That Invoke Macros .....	55
5.1.1 Passing Parameters between Macros .....	56
5.1.2 Passing Parameters When Macros Call Macros.....	57
5.1.3 Passing Macro Parameters through Macro Calls—An Illustrated Example .....	58
5.1.4 Controlling Macro Calls .....	62
5.1.5 Nesting Macro Definitions .....	63
5.2 Conditional Execution Using %IF-%THEN/%ELSE Statements.....	64
5.2.1 Executing Macro Statements.....	65
5.2.2 Building SAS Code Dynamically .....	66
5.2.3 Using the IN Comparison Operator .....	69
5.3 Iterative Execution of Macro Statements.....	70
5.3.1 %DO Block.....	70
5.3.2 Iterative %DO Loops .....	73
5.3.3 %DO %UNTIL Loops.....	76

5.3.4 %DO %WHILE Loops.....	77
<b>5.4 Additional Macro Program Statements .....</b>	<b>78</b>
5.4.1 Macro Comments.....	79
5.4.2 %GLOBAL and %LOCAL.....	81
5.4.3 %SYSEXEC .....	84
5.4.4 Termination of Macro Execution with %ABORT .....	84
5.4.5 Normal Termination of Macro Execution with %RETURN.....	85
<b>5.5 Testing Your Knowledge with Chapter Exercises.....</b>	<b>86</b>
<b>Chapter 6: Interfacing with Data Set Values .....</b>	<b>89</b>
<b>6.1 Using the SYMPUTX Routine to Create Macro Variables.....</b>	<b>90</b>
6.1.1 Introducing SYMPUTX Syntax .....	91
6.1.2 Comparing SYMPUTX with SYMPUT .....	93
6.1.3 Using a Macro Variable in the Same Step That Created It .....	95
6.1.4 Building a List of Macro Variables .....	96
<b>6.2 Defining Macro Variables in a PROC SQL Step .....</b>	<b>98</b>
6.2.1 Placing a Single Value into a Single Macro Variable .....	98
6.2.2 Building a List of Values .....	99
6.2.3 Placing a List of Values into a Series of Macro Variables.....	102
6.2.4 Understanding Automatic SQL-Generated Macro Variables.....	105
<b>6.3 Moving Text from Macro Variables into Code .....</b>	<b>106</b>
6.3.1 Assignment and RETAIN Statements.....	106
6.3.2 SYMGET and SYMGETN Functions .....	107
6.3.3 The RESOLVE Function .....	109
6.3.4 Comparison of the SYMGET and RESOLVE Functions .....	111
6.3.5 Less-Than-Optimal Uses of SYMGET and RESOLVE.....	115
<b>6.4 Using Data to Control Program Flow.....</b>	<b>116</b>
6.4.1 Assigning Macro Variable Values .....	117
6.4.2 Assigning Macro Variable Names as well as Values .....	119
<b>6.5 Executing Macro Code Using CALL EXECUTE .....</b>	<b>121</b>
6.5.1 Executing Non-Macro Code.....	122
6.5.2 Executing Macro Code .....	123
6.5.3 Addressing Timing Issues .....	125
<b>6.6 Testing Your Knowledge with Chapter Exercises.....</b>	<b>129</b>
<b>Chapter 7: Using Macro Functions .....</b>	<b>131</b>
<b>7.1 Quoting Functions.....</b>	<b>132</b>
7.1.1 Using the %BQUOTE Function .....	135
7.1.2 %STR .....	137
7.1.3 Considerations When Quoting .....	137
7.1.4 Basic Types of Quoting Functions and Why We Care .....	142
7.1.5 A Bit about the %QUOTE and %NRQUOTE Functions .....	145
7.1.6 Removing Masking Characters.....	145
7.1.7 The %SUPERQ Quoting Function.....	146
7.1.8 Quoting Function Summary.....	148
7.1.9 Quoting Mismatched Symbols with the %STR and %QUOTE Functions .....	149

<b>7.2 Text Functions .....</b>	<b>150</b>
<b>7.2.1 %INDEX .....</b>	<b>152</b>
<b>7.2.2 %LENGTH .....</b>	<b>153</b>
<b>7.2.3 %SCAN and %QSCAN .....</b>	<b>154</b>
<b>7.2.4 %SUBSTR and %QSUBSTR.....</b>	<b>157</b>
<b>7.2.5 %UPCASE and %QUPCASE .....</b>	<b>158</b>
<b>7.2.6 %LEFT and %QLEFT.....</b>	<b>159</b>
<b>7.2.7 %LOWCASE and %QLOWCASE.....</b>	<b>160</b>
<b>7.2.8 %TRIM and %QTRIM .....</b>	<b>161</b>
<b>7.3 Evaluation Functions .....</b>	<b>162</b>
<b>7.3.1 Explicit Use of %EVAL .....</b>	<b>162</b>
<b>7.3.2 Implicit Use of %EVAL .....</b>	<b>164</b>
<b>7.3.3 Using %SYSEVALF.....</b>	<b>166</b>
<b>7.4 Using DATA Step Functions and Routines.....</b>	<b>169</b>
<b>7.4.1 Using %SYSCALL.....</b>	<b>169</b>
<b>7.4.2 Using %SYSFUNC and %QSYSFUNC.....</b>	<b>170</b>
<b>7.4.3 Taking Advantage of Less Commonly Used DATA Step Functions.....</b>	<b>173</b>
<b>7.5 Building Your Own Macro Functions .....</b>	<b>176</b>
<b>7.5.1 Introduction.....</b>	<b>176</b>
<b>7.5.2 Building the Function .....</b>	<b>177</b>
<b>7.5.3 Using the Function .....</b>	<b>180</b>
<b>7.5.4 Returning a Value .....</b>	<b>181</b>
<b>7.6 Other Useful User-Written Macro Functions .....</b>	<b>182</b>
<b>7.6.1 One-Liners.....</b>	<b>182</b>
<b>7.6.2 Macro Functions with Logic.....</b>	<b>187</b>
<b>7.6.3 Functions for the DATA Step.....</b>	<b>190</b>
<b>7.7 Testing Your Knowledge with Chapter Exercises .....</b>	<b>193</b>
<b>Chapter 8: Discovering Even More Macro Language Elements.....</b>	<b>195</b>
<b>8.1 Even More Macro Functions .....</b>	<b>196</b>
<b>8.1.1 Accessing System Environmental Variables Using %SYSGET .....</b>	<b>196</b>
<b>8.1.2 %SYSMEXECDEPTH and %SYSMEXECNAME.....</b>	<b>199</b>
<b>8.1.3 Assessing Macro Existence and Execution Status with %SYSMACEXEC and %SYSMACEXIST .....</b>	<b>200</b>
<b>8.1.4 Determining Product Availability Using %SYSPROD .....</b>	<b>201</b>
<b>8.1.5 Checking Up on Macro Variable Scopes .....</b>	<b>203</b>
<b>8.2 Even More Macro Statements .....</b>	<b>204</b>
<b>8.2.1 Extending the Use of %SYMDEL .....</b>	<b>204</b>
<b>8.2.2 Using the %GOTO and %label Statements Appropriately.....</b>	<b>206</b>
<b>8.2.3 Using %WINDOW and %DISPLAY.....</b>	<b>208</b>
<b>8.2.4 Extending %SYSEXEC with Examples.....</b>	<b>211</b>
<b>8.2.5 Deleting Macro Definitions with %SYSMACDELETE .....</b>	<b>212</b>
<b>8.2.6 Making Macro Variables READONLY .....</b>	<b>213</b>
<b>8.3 Even More Automatic Macro Variables .....</b>	<b>214</b>
<b>8.3.1 Passing VALUES into SAS Using &amp;SYSPARM.....</b>	<b>214</b>

8.3.2 Learning More about Deciphering Errors .....	216
8.3.3 Taking Advantage of the Parameter Buffer .....	220
8.3.4 Using &SYSNOBS as an Observation Counter.....	223
8.3.5 Using &SYSMACRONAME.....	224
8.3.6 Using &SYSLIBRC and &SYSFILRC.....	224
8.4 Even More System Options.....	225
8.4.1 Memory Control Options .....	225
8.4.2 Preventing New Macro Definitions with NOMCOMPILE .....	226
8.5 Even More DATA Step Functions and Statements.....	226
8.5.1 DOSUBL Function .....	226
8.5.2 Deleting Macro Variables with CALL SYMDEL .....	228
8.5.3 Using SYMEXIST, SYMGLOBL, and SYMLOCAL .....	229
<b>Chapter 9: Exploring Some Less Common Intermediate Topics.....</b>	<b>231</b>
9.1 Building Macro Calls.....	231
9.1.1 Building Macro Calls %&name .....	231
9.1.2 Calling Macros from the Display Manager .....	233
9.2 Working with Macro Variables.....	236
9.2.1 Determining Macro Variable Existence and Scope .....	236
9.2.2 Creating a Large Number of Macro Variables.....	238
9.3 Using the Macro Language to Form Simple Hash Tables .....	241
9.4 Using the Macro Language for Formatted Table LookUps.....	243
9.5 Making Comparisons to Null Values—Some Considerations .....	244
9.6 Evaluating Expressions Stored in a Data Set.....	245
9.7 Using Macro Language Elements on Remote Servers .....	246
9.8 Working with Macro Variables That Contain Special Characters.....	248
9.8.1 Quoting Review.....	248
9.8.2 The Problem with Quotes .....	248
9.8.3 Ampersands and Percent Signs.....	249
9.8.4 Lists and Nested Functions—The Comma Problem.....	251
<b>Chapter 10: Building and Using Macro Libraries .....</b>	<b>253</b>
10.1 Establishing Macro Libraries .....	254
10.2 Using %INCLUDE as a Macro Library .....	254
10.3 Using Stored Compiled Macro Libraries .....	256
10.3.1 Stored Compiled Macro Library Overview.....	256
10.3.2 Defining and Using a Stored Compiled Macro Library.....	256
10.3.3 Storing and Retrieving the Source Code for Compiled Macros.....	258
10.3.4 Recovering Compiled Macro Source Code .....	260
10.3.5     Using the %SYSMACDELETE Statement.....	260
10.3.6     Changing the SASMSTORE= libref Location.....	260
10.4 Using the Autocall Facility.....	261
10.4.1 Autocall Library Review .....	262
10.4.2 Tracking Autocall Macro Locations .....	262
10.4.3 Options Used with Macro Libraries.....	265
10.5 Macro Library Essentials.....	265

10.5.1 The Macro Library Search Order .....	265
10.5.2 Establishing a Macro Library Structure and Strategy .....	266
10.5.3 Interactive Macro Development.....	266
10.5.4 Modifying the SASAUTOS System Variable.....	267
<b>10.6 Autocall Macros Supplied by SAS.....</b>	<b>268</b>
10.6.1 %VERIFY and %KVERIFY.....	270
10.6.2 %LEFT and %QLEFT.....	270
10.6.3 %CMPRES and %QCMPRES.....	271
10.6.4 %LOWCASE and %QLOWCASE.....	272
10.6.5 %TRIM and %QTRIM .....	273
10.6.6 %DATATYP .....	273
10.6.7 %COMPSTOR .....	274
10.6.8 Autocall Macros That Assist with Color Conversions .....	275
10.6.9 Surfacing Other Autocall Macros Supplied by SAS .....	277

## **Part 3: Dynamic Macro Coding Techniques..... 279**

<b>Chapter 11: Writing Dynamic Programs.....</b>	<b>281</b>
<b>11.1 Dynamic Programming Introduction and Design Elements .....</b>	<b>282</b>
11.1.1 A Short Macro Language Review from the Perspective of a Dynamic Programmer	282
11.1.2 Elements of a Dynamic Program .....	287
11.1.3 Creating Data Independence .....	289
11.1.4 Elements for Making a Program Dynamic .....	289
11.1.5 Controlling the Program with Data.....	290
11.1.6 List Processing Basics.....	291
11.1.7 Iterative Step Execution.....	291
11.1.8 Building Statements .....	291
<b>11.2 Information Sources .....</b>	<b>293</b>
11.2.1 Using SASHELP Views .....	293
11.2.2 Using SQL DICTIONARY Tables .....	296
11.2.3 Automatic Macro Variables .....	297
11.2.4 %SYSFUNC and DATA Step Functions.....	297
11.2.5 Retrieving Operating System Information .....	300
11.2.6 Using Data Set Metadata.....	300
11.2.7 Using Data Tables to Control a Process.....	303
11.2.8 Creating and Using Control Files.....	304
11.2.9 Using SET Statement Options.....	306
<b>11.3 Using &amp;&amp;VAR&amp;I Constructs as Vertical Macro Arrays .....</b>	<b>307</b>
11.3.1 Creating the List of Macro Variables.....	308
11.3.2 Resolving &&VAR&i .....	308
11.3.3 Stepping through a List of Data Sets .....	309
<b>11.4 Horizontal Lists .....</b>	<b>309</b>
11.4.1 Creating Horizontal Lists .....	310
11.4.2 Resolving Horizontal Lists .....	310

11.4.3 Stepping through the Horizontal List .....	311
11.4.4 Counting the Items in a List .....	312
11.5 Using CALL EXECUTE.....	313
11.6 Writing %INCLUDE Programs .....	315
11.7 Writing Applications without Hardcoded Data Dependencies.....	317
11.7.1 Generalized and Controlled Repeatability .....	318
11.7.2 Setting Up Project Control Files .....	319
11.7.3 Using Control Files to Build Macro Variable Lists .....	321
11.7.4 Using Control Files to Create Empty Data Sets .....	322
11.7.5 Using Control Files to Create Data Validation Checks Dynamically.....	324
11.8 Building SAS Statements Dynamically .....	327
11.9 More Than Just the Macro Coding .....	328
11.9.1 Naming Conventions.....	328
11.9.2 Directory Structure.....	330
11.9.3 Using the AUTOEXEC File .....	333
11.9.4 Unifying fileref and libref Definitions .....	334
<b>Chapter 12: Examples of Dynamic Programs.....</b>	<b>335</b>
12.1 File Management.....	335
12.1.1 Copy an Unknown Number of Catalogs.....	336
12.1.2 Appending Unknown Data Sets .....	336
12.2 Controlling Output .....	342
12.2.1 Coordinating Titles (or Footnotes).....	342
12.2.2 Auto Display of ODS Styles .....	344
12.2.3 Consolidating ODS OUTPUT Destination Data Sets .....	345
12.3 Adapting Your SAS Environment.....	346
12.3.1 Maintaining System Options .....	346
12.3.2 Building and Maintaining Formats.....	347
12.3.3 Working with Libraries and Directories .....	350
12.4 Working with Data Sets and Variables .....	351
12.4.1 Splitting a Data Set Vertically.....	352
12.4.2 Creating a List of Variable Names from Procedure Output .....	353
12.4.3 Parsing Individual Values from an Existing Horizontal List .....	360
12.4.4 Placing Commas between Words .....	364
12.4.5 Quoting Words in a List .....	365
12.4.6 Checking for Existence of Variables .....	366
12.4.7 Removing Repeated Words from a List .....	367
12.4.8 Controlled Data Corrections and Manipulations .....	369
<b>Part 4: Miscellaneous Topics and Examples .....</b>	<b>373</b>
<b>Chapter 13: Examples and Utilities to Perform Various Tasks .....</b>	<b>375</b>
13.1 Working with Operating System Operations.....	375
13.1.1 Write the First N Lines of a Series of Flat Files .....	375
13.1.2 Storing System Clock Values in Macro Variables .....	378
13.1.3 Executing a Series of SAS Programs .....	379

<b>13.2 Working with the Output Delivery System.....</b>	<b>381</b>
13.2.1 Why You Might Need to Automate with Macros .....	382
13.2.2 Controlling Directories.....	382
13.2.3 Controlling Hyperlinks .....	384
<b>13.3 Working with Data.....</b>	<b>389</b>
13.3.1 Selection of a Top Percentage of Observations .....	389
13.3.2 Selection of Top Percentage Using the POINT Option.....	390
13.3.3 Random Selection of Observations.....	391
13.3.4 Building a WHERE Clause Dynamically .....	394
<b>Chapter 14: Miscellaneous Topics.....</b>	<b>397</b>
14.1 More on Triple Ampersand Macro Variables .....	397
14.1.1 Overview of Triple-Ampersand Macro Variables .....	398
14.1.2 Selecting Elements from Macro Arrays .....	398
14.2 Doubly Subscripted Macro Arrays .....	399
14.2.1 Subscript Resolution Issues for a Simple Case .....	400
14.2.2 Naming Row and Column Indicators .....	400
14.2.3 Using the &&&VAR&I Variable Form .....	402
14.2.4 Using the %SCAN Function to Identify Array Elements.....	404
14.3 Programming Smarter.....	405
14.3.1 Efficiency Issues.....	405
14.3.2 Programming with Style .....	407
14.3.3 Macro Programming Best Practices .....	409
14.3.4 Debugging Your Macros.....	411
14.3.5 Traps: DATA Step Code versus the Macro Language.....	412
14.3.6 Little Things with a Big Bite.....	417
14.4 Understanding Recursion in the Macro Language .....	425
14.5 Determining Macro Variable Scopes .....	427
14.5.1 Nested or Layered Symbol Tables.....	427
14.5.2 Macro Parameters.....	427
14.5.3 Macro Variables Created with %LET and %DO.....	428
14.5.4 Macro Variables Created with the SYMPUT and SYMPUTX Routines .....	428
14.5.5 Macro Variables Created in a PROC SQL Step Using the INTO: Operator .....	429
14.6 Controlling System Initialization and Termination .....	429
14.6.1 Controlling AUTOEXEC Execution.....	430
14.6.2 Saving the Global Symbol Table .....	431
14.6.3 Executing Initialization and Termination Statements.....	431
14.7 Protecting Macros and Controlling Their Execution.....	432
<b>Appendix 1: Exercise Solutions .....</b>	<b>433</b>
Chapter 2.....	433
Chapter 3.....	435
Chapter 4.....	436
Chapter 5.....	437
Chapter 6.....	440

<b>Chapter 7.....</b>	<b>444</b>
<b>Section 14.3.6 Quizlette.....</b>	<b>447</b>
<b>Appendix 2: Using the Macro Language with Compiled Programs .....</b>	<b>449</b>
<b>A2.1 The Problem: Macro Variable Resolution during Compilation .....</b>	<b>450</b>
<b>A2.2 Using Macro Variables .....</b>	<b>451</b>
<b>A2.2.1 Defining Macro Variables .....</b>	<b>451</b>
<b>A2.2.2 Macro Variables in SCL SUBMIT Blocks .....</b>	<b>452</b>
<b>A2.2.3 Using Macro Variables in SCL .....</b>	<b>453</b>
<b>A2.2.4 Passing Macro Values between SCL Entries .....</b>	<b>453</b>
<b>A2.2.5 Using &amp;&amp;VAR&amp;I Macro Arrays in SCL Programs .....</b>	<b>454</b>
<b>A2.3 Calling Macros from within Compiled Programs.....</b>	<b>454</b>
<b>A2.3.1 Run-Time Macros .....</b>	<b>454</b>
<b>A2.3.2 Compile-Time Macros .....</b>	<b>455</b>
<b>A2.4 Using the Macro Language with FCMP Functions .....</b>	<b>457</b>
<b>A2.4.1 Compile-Time Execution.....</b>	<b>457</b>
<b>A2.4.2 Executing a Macro during Function Execution.....</b>	<b>457</b>
<b>Appendix 3: Utilities and Examples Locator.....</b>	<b>461</b>
<b>Data Set / File Manipulation.....</b>	<b>461</b>
<b>Data Variable Manipulation.....</b>	<b>461</b>
<b>Data Value Manipulation .....</b>	<b>461</b>
<b>Date / Time .....</b>	<b>462</b>
<b>Library / Directory Tools.....</b>	<b>462</b>
<b>Macro Techniques .....</b>	<b>462</b>
<b>Macro Variable Tools.....</b>	<b>462</b>
<b>SAS Execution .....</b>	<b>462</b>
<b>SAS/GRAFH Tools .....</b>	<b>462</b>
<b>System and Environment .....</b>	<b>463</b>
<b>Text Manipulation.....</b>	<b>463</b>
<b>Appendix 4: Code Sample Locator .....</b>	<b>465</b>
<b>A4.1 Macro Variable Constructs.....</b>	<b>465</b>
<b>A4.2 Macro Language Statements, Functions, and Autocall Macros .....</b>	<b>466</b>
<b>A4.3 %MACRO Statement Options .....</b>	<b>469</b>
<b>A4.4 Automatic Macro Variables .....</b>	<b>469</b>
<b>A4.5 DATA Step and Other Non-Macro-Language Elements.....</b>	<b>470</b>
<b>A4.6 SASHELP Views and DICTIONARY Tables .....</b>	<b>473</b>
<b>Appendix 5: Glossary .....</b>	<b>475</b>
<b>Bibliography .....</b>	<b>479</b>
<b>Index .....</b>	<b>505</b>

# About This Book

---

## Purpose

This book was written to provide a comprehensive look at the SAS Macro Language. It takes the reader from the most basic introduction through the most advanced topics in the macro language. Regardless of your current level of understanding of the macro language, this book contains new ways of looking at the language as a tool for improving your SAS programs.

---

## Is This Book for You?

Written for all levels of macro language understanding, this book takes the reader from an introduction that assumes no macro language knowledge—and indeed little SAS knowledge—to advanced topics for the advanced SAS programmer. Regardless of where you are in your journey with SAS, you will find this book helpful if you are at all interested in improving your coding skills and in incorporating macro language elements in your programs.

---

## Prerequisites

Although SAS programming strength is not a prerequisite per se, the stronger you are in the overall use of SAS, the easier it will be for you to learn the macro language. That said, even someone just starting to learn SAS can take advantage of the fundamentals of the macro language. The book is written with the expectation that readers will come to it with widely varying levels of understanding of SAS. If you are just starting with the macro language, the concepts in this book should provide you challenge for years to come.

---

## What's New in This Edition

The third edition not only incorporates recent changes and additions to the macro language, but, more important, it greatly expands the sections on using list processing techniques, writing dynamic applications, and writing data-driven macros.

---

## Scope of This Book

This is a comprehensive book on the SAS macro language. It covers all aspects of the macro language from basic concepts, to how the macro language thinks and interacts with the rest of SAS, and then on to the most advanced macro programming techniques.

The book itself is divided into four primary parts, the first three of which roughly translate into levels of complexity.

---

### Part 1

Part 1 is a very basic introduction to the macro language, including explanations of why one should want to go to the trouble of learning the language. It contains basic explanations and basic syntax. This part of the book is designed for the user who is new to the macro language, and it is written to be read sequentially. It is *not* written as a syntax or reference guide.

---

## **Part 2**

Part 2 covers the beginning and intermediate usage and organization of the macro language. In these chapters macro language statements and functions are introduced and demonstrated. Macro quoting is explained, and you will learn to write and use your own macro functions.

---

## **Part 3**

Part 3 presents in detail the writing of dynamic applications. Various aspects of list processing are covered in detail. Throughout the examples in these chapters, you will discover numerous data-driven programming techniques.

---

## **Part 4**

Part 4 includes more examples and many miscellaneous macro language topics.

---

## **About the Examples**

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

Although this book was written using the latest maintenance release, SAS 9.4 M4, virtually all of the content and examples will be applicable to all users of SAS, regardless of SAS release, their operating system, or their SAS interface (batch, SAS Display Manager, SAS Enterprise Guide, SAS Studio, or SAS University Edition).

---

### **Example Code and Data**

There are nearly 400 example programs that accompany this book. These programs use either standard SAS data sets such as those in the SASHELP library, or data sets that are included with the downloaded programs. The SAS example programs are organized by chapter, and the names of the programs correspond to the section number. For example, the first program in Section 6.3.2 is Program 6.3.2a, which can be found in the downloadable Chapter 6 SAS programs with the filename Carpenter\_17835TW\_Program6.3.2a.sas.

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 “Art Carpenter.” Then look for the cover thumbnail of this book, and select “Example Code and Data” to display the SAS programs that are included in this book.

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

---

### **SAS University Edition**



This book is compatible with SAS University Edition.

If you are using SAS University Edition you can use the code and data sets provided with this book. This helpful link will get you started: [http://support.sas.com/publishing/import\\_ue.data.html](http://support.sas.com/publishing/import_ue.data.html).

---

## **Exercises and Solutions**

Exercises designed to test your understanding of the material discussed in this book can be found at the end of Chapters 2 through 7. Programs used in these exercises can be found among the other example programs (see “Example Code and Data” above). Answers to the exercise questions, as well as solutions to programming tasks, can be found in Appendix 1 of this book.

---

## **Additional Help**

Although this book illustrates many analyses regularly performed in businesses across industries, questions specific to your aims and issues may arise. To fully support you, SAS Institute and SAS Press offer you the following resources:

- For questions about topics covered in this book, contact the author through SAS Press:
  - Send questions by email to [saspress@sas.com](mailto:saspress@sas.com); include the book title in your correspondence.
  - Submit feedback on the author's page at [http://support.sas.com/author\\_feedback](http://support.sas.com/author_feedback).
  - More information on the book is available on sasCommunity.org at <http://www.sascommunity.org/wiki/Category:Carpenters Complete Guide to the SAS Macro Language, Third Edition>.
- For questions about topics in or beyond the scope of this book, post queries to the relevant SAS Support Communities at <https://communities.sas.com/welcome>.
- A great deal of macro-language-related content can be found on [http://www.sascommunity.org/wiki/Category:Macro\\_Language](http://www.sascommunity.org/wiki/Category:Macro_Language).
- SAS Institute maintains a comprehensive website with up-to-date information. One page that is particularly useful to both the novice and the seasoned SAS user is its Knowledge Base. Search for relevant notes in the “Samples and SAS Notes” section of the Knowledge Base at <http://support.sas.com/resources>.
- Registered SAS users or their organizations can access SAS Customer Support at <http://support.sas.com>. Here you can pose specific questions to SAS Customer Support; under “Support” click “Submit a Problem.” You will need to provide an email address to which replies can be sent, identify your organization, and provide a customer site number or license information. This information can be found in your SAS logs.

---

## **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.

---

### **Contact the Author through SAS Press**

- By email: [saspress@sas.com](mailto:saspress@sas.com)
- Via the Web: [http://support.sas.com/author\\_feedback](http://support.sas.com/author_feedback)

---

### **Purchase SAS Books**

For a complete list of books available through SAS, visit [sas.com/store/books](http://sas.com/store/books).

- Phone: 1-800-727-0025
- Email: [sasbook@sas.com](mailto:sasbook@sas.com)

---

### **Subscribe to the SAS Learning Report**

Receive up-to-date information about SAS training, certification, and publications via email by subscribing to the SAS Learning Report monthly eNewsletter. Read the archives and subscribe today at <http://support.sas.com/community/newsletters/training>!

---

### **Publish with SAS**

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

## **Part 1: Macro Basics**

<b>Chapter 1 What the Language Is, What It Does, and What It Can Do .....</b>	<b>3</b>
<b>Chapter 2 Defining and Using Macro Variables .....</b>	<b>17</b>
<b>Chapter 3 Defining and Using Macros .....</b>	<b>35</b>
<b>Chapter 4 Using Macro Parameters.....</b>	<b>45</b>

# **Chapter 1: What the Language Is, What It Does, and What It Can Do**

<b>1.1 Introduction.....</b>	<b>3</b>
<b>1.2 Stages of Macro Language Learning .....</b>	<b>5</b>
1.2.1 Code Substitution .....	5
1.2.2 Macro Language Elements .....	6
1.2.3 Dynamic Programming.....	7
<b>1.3 Terminology .....</b>	<b>7</b>
<b>1.4 Sequencing Events—It's All about the Timing .....</b>	<b>8</b>
<b>1.5 Scopes or Referencing Environments .....</b>	<b>12</b>
1.5.1 Use of Symbol Tables.....	12
1.5.2 Nested Symbol Tables.....	13

The macro language enables the SAS programmer to greatly increase the power, flexibility, and capability of SAS. The macro language can be used to generalize programs and to make them reusable. It is a complex language with many nuances, and although it is not easily mastered, even its simplest tools and capabilities can have a great impact on your workflow and relative efficiencies.

Within Base SAS software, the SAS Macro Facility is a tool that contains the essential elements that enable you to use the macro language. The macro facility contains the macro language and a macro processor that translates macro code into statements that can be used by SAS. The macro language provides the means to communicate with the macro processor.

For the examples in this chapter and indeed for all of the code examples throughout the book, if you want to execute these sample programs, then be sure to follow the setup instructions. Remember that all of the data sets and programs are available for download, so you do not need to retype either the code or the data. For instructions on accessing and setting up the programs and data, see the “Example Code and Data” section within this edition’s “About This Book” front matter.

---

## **1.1 Introduction**

The macro language consists of its own set of commands, options, syntax, and compiler. Although many macro statements have similarities to the statements in the DATA step, you must understand the differences in behavior in order to effectively write and use macros. The macro language provides tools that enable you to do the following:

- Pass information between SAS steps
- Dynamically create code after the user submits the program for execution
- Conditionally execute DATA or PROC steps
- Create generalizable and flexible code

The tools made available through the macro facility include macro (or symbolic) variables, macro statements, and macro functions. These tools are included as part of the SAS code, or program, where they are detected when the code is sent to the SAS Supervisor for execution.

First and foremost, you should always keep in mind that, although it can do much more, the macro facility is primarily a source code generator. Whether you are substituting the name of a data set, or you are having the macro language write a complex DATA step, the macro facility will be writing code. It works with text as input and writes source code as output.

There is a lot to learn concerning the SAS macro language. This is true even for those of us who have been using the macro language for years. This means that learning the language requires no small investment in time and energy. So is the reward worth the investment? For myself I would say, *absolutely!* But before you start the learning process, there are some points that can be helpful as you gather knowledge.

As complicated as the macro language is to learn, there are very strong reasons for doing so. As just mentioned, at its heart the macro language is a code generator. In its simplest uses, it can substitute simple bits of code like variable names and the names of data sets that are to be analyzed. In more complex situations, it can be used to create entire statements and steps based on information that might be unavailable to the person writing or even executing the macro. At the time of execution, the macro language can be used to make queries of the SAS environment, as well as the operating system, and use the gathered information to make informed decisions about how it is to further function and execute.

In the olden days, typewriters (you might remember seeing one in a museum) were used to write text to paper, and the modern keyboard mimics the keys of the typewriter. When we write a SAS program, we generally type in the code from our keyboard. Our programs consist of DATA steps and PROC steps (along with the occasional global statement). We know the names of the data sets and variables that are to be included in the code, and we use them in our DATA steps and PROC steps. The macro language can be used in this typing process. The macro language can serve as our intelligent typist, which generates all or portions of our code for us.

Because the macro language is at its heart a code generator, it can act as a typewriter that we can control. We can use it to write all or any portions of our code for us. In a simple PROC PRINT step, we might want to print a selected number of lines of the specified SAS data set. The code must name the data set to be printed and any options that are to be applied.

In the following PROC PRINT step, we have named the data set of interest and the number of observations that are to be printed:

```
proc print data=sashelp.class(obs=10);  
run;
```

Clearly, if this is our entire program, then there is little motivation to automate or expand on its capabilities, but let's use it as a metaphor for a much longer, more complex program. Perhaps the data set name appears multiple times throughout the program, and when the data set name changes, then each occurrence must be edited separately. Instead, the macro language can be used to make these changes for us. It can do so through the use of code substitution techniques, which use macro variables.

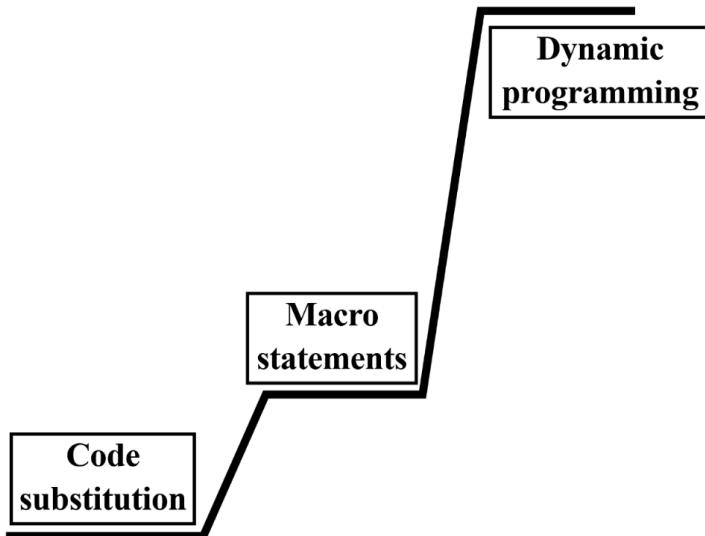
Macro variables are stored in memory and are used to hold text values. They can contain a single letter, a word, a list of words, SAS statements, or even complete steps. There are more than a dozen ways to create macro variables, and the simplest, and probably the most common, is through the use of the %LET statement (for details about the %LET statement, see Chapter 2). Cynthia Zender, renowned trainer for SAS Institute, uses the concept of the macro language as a typewriter extensively when she teaches the macro language. If you found this analogy helpful, then I probably stole the idea from her.

**SEE ALSO:** Lougee (2016) has a brief introduction to the macro language with discussion of whether or not a macro is needed. Carpenter (2014b) also discusses the macro language in general terms with much of that text appearing in this chapter. The essential introduction to statements and macro variables is provided by Lafler (2015).

## 1.2 Stages of Macro Language Learning

Most programmers who learn the macro language go through three stages, or levels, of learning (see Figure 1.2). Code substitution is the first, and generally the most straightforward, of the three. Generally, even those new to SAS can easily make use of code substitution techniques without encountering any of the macro language issues that can impede the learning process.

**Figure 1.2: Stages of Macro Language Learning**



The second level makes use of macro statements, macro functions, and macro logic. The learning curve between the first and second learning step is fairly short and not too steep. With practice, and a reasonably good understanding of the basic SAS programming language without the use of macros, most programmers are able to start to use the basic aspects of these elements of the macro language fairly quickly.

The learning curve for the third level, dynamic programming techniques, is much steeper and longer. Often programmers who have gained proficiency in the second level will take months of practice before even starting to become comfortable with using the techniques of this third learning level. Indeed, I have known SAS programmers who are quite adept at the macro language that have little or no understanding of the elements of the third level.

Actually—true confession time—I programmed in the macro language for years, blissfully unaware that this third level even existed. Then I was confronted by a problem that could be solved only with these techniques, so I was dragged, with much resistance (read *kicking, screaming, gnashing of teeth*), into this new (to me) world of dynamic programming. However, the difference in power and capability between the techniques in the second level and those in the third level make the effort to acquire the necessary knowledge well worthwhile. I cannot imagine being unable to take advantage of dynamic programming techniques, and I am still hoping that I will one day find that there is a fourth level.

The first three parts of this book roughly correspond to these three levels of macro learning:

- Part 1 (Code Substitution)—Chapters 1–4
- Part 2 (Macro Statements and Functions)—Chapters 5–10
- Part 3 (Dynamic Programming)—Chapters 11–12

### 1.2.1 Code Substitution

Analogous to the assignment statement in the DATA step, the %LET statement is a macro language statement that is used to assign a text value to a macro variable. The %LET statement, like all macro statements, begins with a percent sign (%), which precedes a keyword, and ends with a semicolon. Macro

variables will be discussed in more detail in Chapter 2. However, for now it is important to know only that macro variables store text values in memory during the execution of a SAS program. Program 1.2.1 demonstrates two simple %LET statements and a PROC PRINT.

#### Program 1.2.1: Creating Macro Variables with the Use of %LET

```
%let dsn = sashelp.class;  
%let cnt = 5;
```

In order to take advantage of macro variables once they are created, you need to be able to call them by name when they are needed. In our SAS programs, we name macro variables by using an ampersand. Essentially, these names are used in our program as symbolic references, dummy values, for the true values of the items that will be substituted later during program execution.

#### Program 1.2.1 (continued): Using the Macro Variables for Code Substitution

```
proc print data=&dsn(obs=&cnt);  
run;
```

When we write our program, using macro variables, we give ourselves a deeper level of programming flexibility. Imagine for a moment that the simple PROC PRINT shown here in Program 1.2.1 is actually a program consisting of many steps, perhaps of several hundred lines in length. If the items specified in the macro variables appear throughout the program, we need change only the %LET statements to radically alter what the program will do. Essentially, before the PROC PRINT step can execute, the macro variables must be resolved. The macro facility, the typewriter, replaces the macro variables with their stored values. Because this code substitution is done before the PROC step is scanned, the macro language is essentially typing code for us:

```
proc print data=sashelp.class(obs=5);  
run;
```

---

## 1.2.2 Macro Language Elements

In the second level shown in Figure 1.2, the programmer goes beyond code substitution by taking advantage of more of the elements of the macro language. Some of these elements bear a strong resemblance to similar elements in the DATA step. This resemblance reinforces the notion that you cannot be a better macro language programmer than you are a DATA step programmer. Similar elements include the following:

- **Functions.** Many macro functions have analogous counterparts to the DATA step functions.
- **Statements.** Some executable DATA step statements, such as the DO and IF, can also be found with similar syntax in the macro language.
- **Options.** System options can be used to affect the behavior of the macro language.

Mastering this level of the macro language enables you to control the flow of your program at the step level. The DATA step IF statement gives you control within a DATA step, but not across steps. The macro %IF statement operates on the program-code level (remember macro language is a code generator) and writes the steps that are to be executed. You can use macro functions to determine the status of our program and then, using logic, apply that information to guide the execution of the program itself. Imagine a program that creates a data set and then, depending on the number of observations, performs either a PROC PRINT or a PROC MEANS on that data set.

You can also use macro logic and macro language elements to control the statements and flow of your program within a step. These could include things like which data set to read, which observations to remove, and the naming of DATA step variables.

As is so often the case when you are learning something new, it is not immediately obvious why it is worth the trouble to learn the macro language. But once you start to accumulate an understanding of the macro language elements, you will find more and more uses for them. Soon you will be wondering how you ever managed to program without them.

---

### 1.2.3 Dynamic Programming

The true power and flexibility of the macro language becomes available to you as you master the techniques at the third level shown in Figure 1.2. Dynamic programming is less about the macro language itself, and more about the nuances of how the macro language elements are applied. Although there are some macro language elements that are used in dynamic programs that you are less likely to use in the first two levels, for the most part you will just be using these elements in different ways in the third level.

The construction and use of lists is crucial in these kinds of programs. List processing enables you to execute a program many times, with each iteration being based on different items, such as data sets or variables. There are four distinct types of list processing (Fehd and Carpenter, 2007), and the two most common and flexible approaches strongly utilize many of the macro language elements learned in Level 2 (Rosenbloom and Carpenter, 2014).

Dynamic programs adjust to their environment. Rather than telling the program which data sets or variables to process against, these programs tend to make these determinations at execution time. This means that you can write programs that will execute against data sets that have not yet been created, or perhaps even named, when you do your coding.

In our PROC PRINT example, we have selected one data set to print. What if we want to print every data set in a given directory, and we do not know the data set names or even how many data sets there are in that directory? A dynamic program can determine the names, make a list, count the items in the list, and then “type” the code to generate the PROC PRINT steps for each of the data sets. If the list of names changes, our program will not care; it will dynamically adjust to different data set names and different numbers of data sets. Dynamic programs tend not to have hardcoded information in the program when that information can be determined by the program at execution time.

There are a great many tools in SAS and in the macro language that can be used by the programmer to help the macro language to dynamically determine the information that it needs to execute. SAS has special data sets and tables that contain all kinds of run-time information about the operating system, about the SAS environment, and about the data sets themselves. There are functions and statements that can retrieve this information and position it so that the macro language can take advantage of it.

---

## 1.3 Terminology

The statement and syntax structure that is used by the macro facility is known as the *macro language*, and like any language, it has its own terminology. The SAS user who understands the programming language used in Base SAS, however, will discover quickly that much of the syntax and content of the macro language is familiar.

The following terms will be used throughout this book:

- **Text.** This term denotes a collection of characters and symbols that can contain variable names, data set names, SAS statement fragments, complete SAS statements, or even complete DATA and PROC steps. Text forms the primary building blocks used by the macro language.
- **Macro variable.** The names of macro variables are almost always preceded by an ampersand (&) when used in SAS code. Macro variables are generally used to store text.
- **Macro program statement.** A macro statement controls what actions take place during the macro execution. Like Base SAS language statements, it starts with a keyword that in the macro language is always preceded by a percent sign (%) and is often syntactically similar to statements used in the DATA step.
- **Macro facility.** This term refers to the software responsible for interpreting and executing macro language statements and elements.
- **Macro references.** When encountered by the SAS parser, these references to the macro language invoke the macro facility.
- **Macro triggers.** The symbols & and % are known as *macro language triggers* and are used to designate macro language references and elements. These symbols cause the macro facility to be invoked.

- **Macro.** Also known as a *macro program*, a macro is a stored collection of macro language statements and text.
- **Macro expression.** One or more macro variable names, text, and macro functions, combined together by one or more operators or parentheses. Macro expressions are closely analogous to the expressions used in Base SAS programming.
- **Macro function.** This term denotes predefined routines for processing text in macros and macro variables. Many macro functions are similar to functions used in the DATA step.
- **Operators.** These symbols are used for comparisons, logical operation, or arithmetic calculations. The operators are the same ones used in Base SAS language comparisons.
- **Automatic macro variables.** These special-purpose macro variables are automatically defined and provided by SAS. These variable names should be considered as reserved.
- **Open code.** These SAS program statements exist outside of any macro definition. Not all macro statements can be used in open code.
- **Resolving macro references.** During the resolution process, elements of the macro language (or references) are replaced with text.

When SAS statements are submitted for processing, they are broken up into their component parts so that SAS can understand them. This is done by the *word scanner*. The basic component parts are known as *tokens*. There are several types of these tokens, but two have special meaning to the macro language. These two types of tokens are designated with the percent sign (%) and the ampersand (&). These two symbols are macro processor *triggers*. When the word scanner detects one of these macro triggers (followed by a letter or underscore), the macro processor is invoked. The statement or macro language element is then turned over to the macro facility for processing.

**MORE INFORMATION:** For additional terminology, see the glossary in the back matter of this book.

**SEE ALSO:** For more information on macro language terminology, see Burlew (2014, Chapter 2).

## 1.4 Sequencing Events—It's All about the Timing

When you run a SAS program, it is executed in a series of DATA and PROC steps, one step at a time. GLOBAL statements (for example, TITLE, FOOTNOTE, and OPTIONS), which can exist outside of these steps, are executed immediately when they are encountered. For each step, SAS first checks whether macro references exist. *Macro references* might be macro variables, macro statements, macro definitions, or macro calls. If the program does not contain any macro references, then processing continues with the DATA step compiler or the PROC step processor. If the program does contain macro references, then the macro processor intercepts and resolves them prior to any further processing of that step. The resolved macro references then become part of the SAS code that is passed to the DATA or PROC step processor.

Before code is passed to the SAS supervisor, the following takes place for each step:

- A check is made to see whether there are any macro statements, macro variables, or macro calls. If there are, then the following occur:
  - Macro variables are resolved.
  - Macro definitions are compiled and stored.
  - Called macros are executed (resolved).
  - Macro statements and functions are executed.
- The DATA or PROC step that contains resolved macro references (if there were any) is compiled and executed.

Because the macro language is primarily a code generator, it makes sense that the code that it creates must be generated before it can be executed. Logically this implies that execution of the macro language comes first. As simple as this notion is conceptually, timing issues and timing conflicts are often not so simple to

recognize in application. But as you use the macro language to take on more complex tasks, it becomes even more critical that you have an understanding of these timing relationships.

First, remember that SAS is a *parsed language*, meaning that the code is processed one portion at a time. If a program contains two DATA steps, a syntax error in the second will not even be noticed until the first step has been completed. The same is true at the statement level. The first statement is parsed before the second is looked at. The submitted code is parsed at both the word and statement levels. Each statement is compiled, then statements are grouped into a step, and then the step is executed. These steps always happen and they must do so in this order. In reality, there is more than one parser, including a separate one for macro language elements.

Think of the parser as a Pac-Man game that gobbles up code instead of energy dots, as shown in Figures 1.4a and 1.4b below. Let's say that we submit a PROC step for execution, without using any macro language elements in our program. When a statement starts with PROC, as is shown in Figure 1.4a, SAS knows that the next word will be the name of the procedure, in this case PRINT. The name of the procedure will then be followed by a limited number of statement options—limited, specifically, by the name of the selected procedure. Effectively, a decision tree is established as the statement is parsed.

**Figure 1.4a: Parsing a PROC Statement**



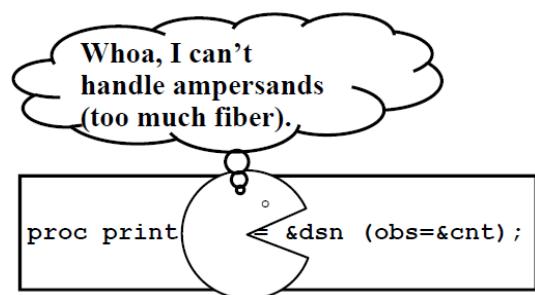
When macro language elements are present (remember that macro language elements are always marked by either an ampersand or percent sign), the process is interrupted. When the parser encounters a macro language element, it cannot continue until the element is resolved or executed. This resolution or execution of the macro language element takes place in the macro facility, while the statement parser waits patiently for a response.

In the PROC PRINT step shown here and back in Section 1.2.1, “Code Substitution,” there are two macro language elements (macro variables):

```
proc print data=&dsn(obs=&cnt);
run;
```

Before the PROC statement can be fully parsed, these macro variables must be resolved. The parser encounters the &DSN, and the &DSN (which is known as a *token*) is then passed to the macro facility. Within the macro facility the &DSN is resolved, in our example, to SASHELP.CLASS. Effectively the SASHELP.CLASS is “typed” (replaces the &DSN in our PROC statement), and it is this “typed” value that the parser sees. This same process then repeats a few characters later when the &CNT is encountered.

**Figure 1.4b: Detecting Macro Language Triggers**



Think of the program that we have written as stream of water, with us as the parser standing on the bank and watching the stream flow by, as shown in Figure 1.4c. Slightly upstream of us is a partner on a bridge, scanning for macro elements that might be floating downstream. These macro language elements are recognizable by the macro triggers, the ampersand (&) and percent sign (%). If the scanner detects a macro language trigger, then the water upstream of us immediately freezes (ceases to flow—hey, this is an analogy, so it can only go so far). The macro language element (token) is lifted from the water and processed, and then the resulting value is placed (“typed”) back into the stream at the same spot. The stream now thaws (in Alaska we call this process *breakup*), starts to flow, and the process continues. This means that, necessarily, by the time that the water reaches us, there are *no macro language elements in the water*.

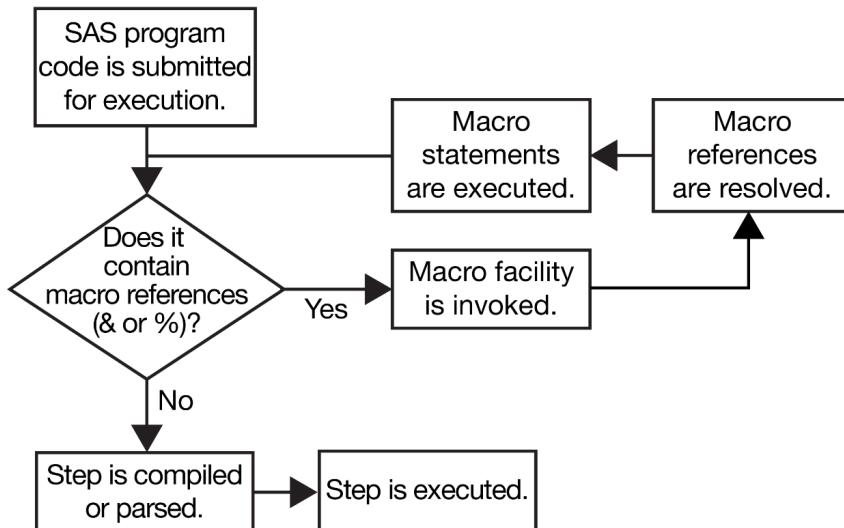
**Figure 1.4c: The Parser as a Watcher Standing on the Stream Bank with the Scanner Upstream**



Source: © Kenna Bates, Kenai Productions

This description is very much a simplification of this process. However, for ease of understanding the basic process, a simplification is called for, and an understanding of this process is absolutely essential, if the programmer is to make full use of the macro language. Reality is of course more complex. In fact, the SAS parser handles some of these steps in what is essentially a simultaneous manner.

If, however, we can live with this simplification, then we can make good use of the process flowchart shown in Figure 1.4d. It can be applied at the step level, statement level, expression level, and even word level. Regardless of the level of application, the code is scanned for macro language elements *before* it can be parsed. This timing is crucial. There are some things in the macro language that just will not make sense to you if you do not understand this process. In fact, although it is not uncommon for successful macro programmers to have no understanding of the order of these events, if you do not understand these timing relationships, then there are some things that you will simply not be able to do in the macro language.

**Figure 1.4d: Macro Language Event Sequencing**

The most important information to glean from this flowchart is the timing of the execution and resolution of macro language elements relative to Base SAS language elements. Not understanding this relationship causes users new to the macro language to ask questions like the following:

- Can macro %IF statements be used interchangeably with DATA step IF statements?
- In a DATA step, why can't I assign the value of a DATA step variable to a macro variable by using the %LET statement?
- Why can't I use a DATA step IF to conditionally execute a %LET?
- Why don't data set variables have values when I'm using them in %IF statements?

You can answer each of these questions by looking at Figure 1.4d. It shows the interaction between the macro and Base SAS language, as well as the order in which the different types of statements are executed. The trouble is that the discernment of the timing of events is not always obvious.

A simple, and unfortunately not uncommon, application of this timing issue and its related confusion can be seen in the DATA step in Program 1.4a. We would like to retrieve the value in the variable AGE and write it into the macro variable &J\_AGE, using the %LET statement, but this plan simply *cannot* work.

#### **Program 1.4a: Showing a Macro Language Timing Issue**

```

data _null_;
  set sashelp.class(keep=name age
                    where=(name='Jane'));
  %let j_age = age;
run;
  
```

During the DATA step's compilation phase, a storage buffer known as the Program Data Vector (PDV) is created. As DATA step variables are identified, named, and associated with attributes, this information is stored in the PDV. During the execution of the DATA step, DATA step variable values are temporarily stored in the PDV where they are available to the functions and assignment statements of the DATA step.

According to the timing shown in Figure 1.4d, the %LET will execute before the DATA step has compiled, which will be before the variable AGE exists on the PDV, and this is before the PDV even exists, and certainly before any data has been read. Fortunately, there are *DATA step tools*, such as the SYMPUTX routine (see Section 6.1), that can be used instead of the %LET statement, that will solve this problem—emphasis on DATA step tools, *not* macro language elements.

It is important for you to keep in mind that, if there are macro references in your code (% or &), these will be resolved *before* the step is even compiled.

Macro %IFs and DATA step IFs are not interchangeable, because the %IF can *never* compare values of variables on the PDV. Indeed, the PDV does not yet exist when the %IF is executed. The %IF statement is described in Section 5.2.

For the same reason, you cannot conditionally assign a value to a macro variable by using an IF statement and the %LET statement, because the %LET is a macro statement and is therefore executed long before the IF statement is even compiled. The %LET statement is first described in the Section 2.2, “Defining Macro Variables.” The application of the timing illustrated in Figure 1.4d is not always this obvious, and it takes practice to look for instances where it applies, but remember—macro language elements will execute *before* the DATA step even exists. Macro language elements are therefore not used to access values on the PDV.

**MORE INFORMATION:** Additional comparisons between the macro language and the DATA step are made in Section 14.3.5.

**SEE ALSO:** *SAS 9.4 Macro Language: Reference, Fourth Edition* (Chapter 2) contains a detailed discussion of how SAS processes statements with and without macro activity.

A very readable and detailed explanation of the internal processes of the macro facility from the SAS developer’s point of view is offered by O’Connor (1998). Li (2010) and Jaffee (1999) both restate this process in simple terms. Burlew (2014) spends all of Chapter 2 and some of Chapter 5 discussing several variations on this series of events.

## 1.5 Scopes or Referencing Environments

Macro variables and the text values that they hold are stored in symbol tables, which in turn are held in memory. Not only is there a number of ways to create macro variables, but they can also be created in a wide variety of situations. How they are created, as well as under what circumstances, affects the variable’s scope—how and where the macro variable is stored and retrieved. There are a number of misconceptions about macro variable scope and about how the macro variables are assigned to symbol tables. These misconceptions can cause problems that the new, and sometimes even the experienced, macro programmer does not anticipate. Understanding the basic rules for macro variable assignment can help you solve some of these problems that are otherwise quite mystifying.

### 1.5.1 Use of Symbol Tables

Unlike the values of data set variables, the values of macro variables are stored in memory in *symbol tables*. Each macro variable’s definition in the symbol table is also associated with a *referencing environment* or *scope*, which is determined by where and how the macro variable is defined. The terms *referencing environment* and *scope* are interchangeable; however, *scope* is currently the preferred term.

Although macro variables can be created in a number of different ways (Carpenter, 2004), they are always stored in symbol tables, and there are only two basic types of symbol tables—global and local. The assignment of a macro variable to a symbol table is not, however, this straightforward. A given macro variable is written to a specific table, and the rules that determine which table is to receive a macro variable can be very arcane (for the specifics, see Section 14.5). Often, when you are writing macro code, even if you know the arcane rules, it is not always possible to anticipate which symbol table will receive the macro variable. To make matters more complicated, there can be any number of local symbol tables (thankfully, there is always only one global symbol table).

A *global* macro variable has a single value available that can be available throughout the remainder of the program and SAS session. Macro variables that are defined outside of any macro will be global.

*Local* macro variables have values that are available only within the macro in which they are defined. Because macros can call other macros, there might be multiple levels of nested local tables.

We know that data sets stored in the WORK directory will be deleted at the end of the SAS session, while data sets stored in user-defined libraries can be stored permanently. How long the data set is stored is known as its *scope* or *persistence*. Once a TITLE is defined in a SAS program, its definition remains until the end of the SAS session or until it is redefined. Its scope is the SAS session. PROC PRINT options defined within a PROC PRINT step are not carried forward to the next step, so their persistence, or scope, is at the step level. Macro variables also have scope, or persistence, levels. The global symbol table and its macro variables have a scope consisting of the SAS session. A local symbol table, however, exists only during the execution of its associated macro. The scope of a local symbol table is limited to its macro's execution; therefore, macro variables stored in a local table are no longer available after the macro terminates. Further discussion of global and local macro variables can be found in Section 5.4.2.

## 1.5.2 Nested Symbol Tables

Imagine that you are executing a macro that has a local symbol table. There will be two different symbol tables available to store macro variables—the global symbol table and the local one associated with the macro. If you create a macro variable and it is written to the global symbol table, it will be available after your macro terminates; however, if it is written to the local table, the macro variable and its value will be available only during macro execution. These two symbol tables are said to be *nested*, with the local table nested within the global table (the global table is said to be “higher”). From within the macro, you can use macro variables in either table—that is, you can always look outward (or to a higher table).

Because each macro creates its own local scope, macro variable values that are defined in one macro might be undefined in another. Indeed, macro variable names need not be unique even among nested macros. This means that the specific value associated with a given macro variable might depend on how the macro variable is used in the program.

When macro calls are nested, their associated local symbol tables are also nested. This means that macro variables known to one macro might also be known within the macros that it calls.

Have you ever seen the Russian dolls known as *matrioshkas*? They are a series of dolls, which are usually carved from wood so that they nest within each other, as shown in the picture below (Figure 1.5.2a). The number of dolls in the set depends on the skill of the artisan, and each doll traditionally is painted as a woman wearing a colored scarf and apron. As you open the doll set for the first time, you can see the colors of the dolls that have been opened, including the one unopened doll, but you have no idea how many more dolls are in the set or the colors of their scarves. In the picture of matrioshka dolls below, we can see that the fourth doll is unopened—is there a fifth? We have no way of knowing, but at any point in the process, we do know how many have already been opened and the colors associated with each doll that has been revealed.

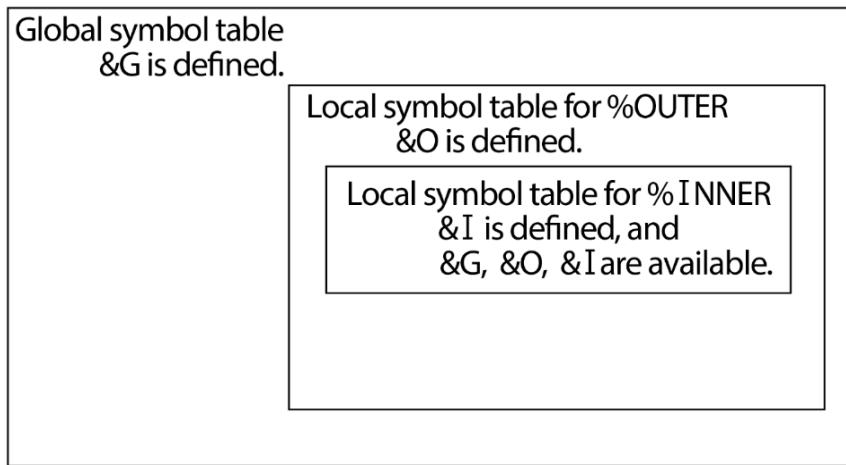
**Figure 1.5.2a: Matrioshka as a Visual Metaphor of Nested Symbol Tables**



Although you might not recognize it as such, one of the powers of the macro language is the ability to have macros that call other macros that call macros, and so on. These nested macro calls can result in nested symbol tables. As with the matrioshkas, where you can see the scarves of the opened dolls, you can use the macro variables in the current local table (the revealed but unopened doll) or any higher symbol table (opened doll), but you cannot access any aspect of the dolls that have yet to be revealed.

In Figure 1.5.2b, the macro variable &G is defined in the global symbol table. The %OUTER macro is called (macros are named and the macro is called, executed, by preceding its name with a percent sign), and it defines &O in its local table. The %INNER macro is then called from within %OUTER, and &I is defined. Although each of these macro variables resides in a different symbol table, during the execution of the %INNER macro, all three macro variables are available to be used.

**Figure 1.5.2b: Nested Symbol Tables**



When macros are executing, nested symbol tables are usually available, and your program can use macro variables from any of the currently existing symbol tables from the most local to the highest (global).

As Figure 1.5.2c shows, once the %INNER macro has completed executing, its symbol table is removed, and any macro variables that it contained are no longer available. Essentially, we have closed up one of the nesting dolls. The macro variable &I is no longer defined, nor is its symbol table.

**Figure 1.5.2c: Symbol Tables after the Macro %INNER Terminates**

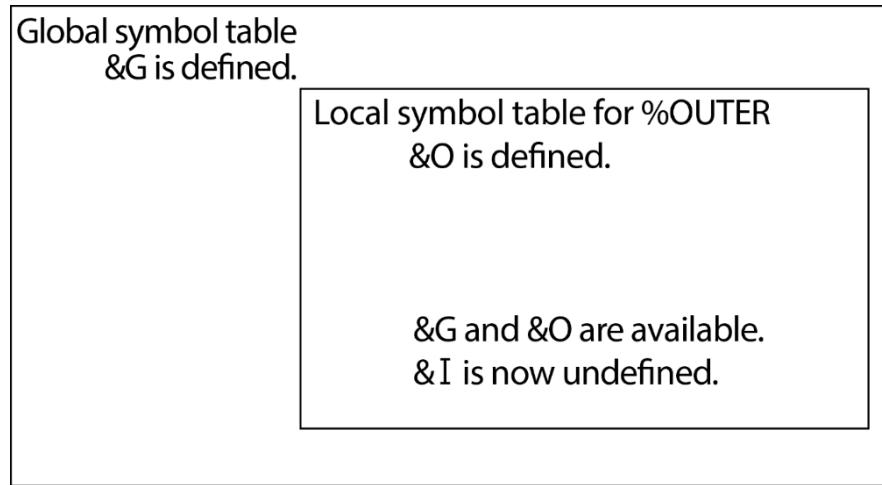


Figure 1.5.2d shows that, after all the macros have completed executing, only the global symbol table and its macro variable is available.

**Figure 1.5.2d: Only the Global Symbol Table Is Available in Open Code**

Global symbol table  
&G is defined.

Only &G is defined.

Because macro variables are always referenced by name, this means that, when a macro variable is defined in the local environment, only the local version of that macro variable can be accessed—even if the same name is used in a higher table. Even when the same name is used in multiple environments, you will be able to access only the most local version. This also means that a macro variable in a higher symbol table will act like a global variable when in the local environment.

**MORE INFORMATION:** You can control the referencing environment for a macro variable through the use of the %GLOBAL and %LOCAL statements, which are described in Section 5.4.2.

**SEE ALSO:** For extensive examples, see *SAS 9.4 Macro Language Reference, Fourth Edition*, (Chapter 5).

Articles that specifically cover referencing environments include Bercov (1993) and Hubbell (1990).

An example of a macro variable that takes on more than one value at the same time is given in Carpenter (1996, p. 1637).

# **Chapter 2: Defining and Using Macro Variables**

<b>2.1 Naming Macro Variables.....</b>	<b>18</b>
<b>2.2 Defining Macro Variables .....</b>	<b>18</b>
<b>2.3 Using Macro Variables.....</b>	<b>19</b>
<b>2.4 Displaying Macro Variables by Using the %PUT Statement.....</b>	<b>21</b>
<b>2.5 Resolving Macro Variables .....</b>	<b>24</b>
2.5.1 Using the Macro Variable as a Suffix .....	25
2.5.2 Using the Macro Variable as a Prefix .....	26
2.5.3 Using Macro Variables as Building Blocks—Appending Macro Variables .....	27
2.5.4 Understanding Results When Macro References Are Not Resolved .....	28
<b>2.6 Using Automatic Macro Variables.....</b>	<b>29</b>
2.6.1 &SYSDATE, &SYSDATE9, &SYSDAY, and &SYSTIME .....	29
2.6.2 &SYSLAST and &SYSDSN.....	30
2.6.3 &SYSERR and &SYSCC.....	31
2.6.4 &SYSRC .....	32
2.6.5 &SYSSITE, &SYSSCP, &SYSSCPL, and &SYSUSERID .....	32
2.6.6 &SYSMACRONAME .....	33
<b>2.7 Removing Macro Variables .....</b>	<b>33</b>
<b>2.8 Testing Your Knowledge with Chapter Exercises .....</b>	<b>33</b>

For most SAS programmers, their first encounter with the macro language is through the use of macro variables. Indeed, macro variables are very powerful all by themselves. Even if you know nothing else about the macro language other than the information contained in this chapter, you will be able to accomplish a great deal.

This chapter introduces macro variables by showing you how they are named, defined, and used in SAS programs. These symbolic variables can be used as a part of any SAS program. Macros and other macro statements need not be present for you to take advantage of the power of macro variables.

For the examples in this chapter and indeed for all of the code examples throughout the book, if you want to execute these sample programs, then be sure to follow the setup instructions. Remember that all of the data sets and programs are available for download, so you do not need to retype either the code or the data. For instructions on accessing and setting up the programs and data, see the “Example Code and Data” section within this edition’s “About This Book” front matter.

**SEE ALSO:** An introductory tutorial to various aspects of the macro language can be found in Leighton (1997). Li (2010) introduces the creation and use of macro variables and discusses the resolution process. Information about macro variables and their resolution, debugging information, and simple examples can be found in Widawski (1999, 2002).

## 2.1 Naming Macro Variables

*Macro variables*, which are also known as *symbolic variables*, are not data set variables. Instead, macro variables belong to the SAS macro language. Once they are defined, they can take on many different values during the execution of a SAS program.

You can use the same basic rules to name macro variables as are used to name data sets and data set variables:

- A name can be 1 to 32 characters in length.
- A name must begin with a letter or underscore (\_).
- Any combination of letters, numbers, and underscores can follow the first character.

Here are some basic rules that apply to the use of macro variables:

- Text that is stored in macro variables can range in length from 0 to 65,534 bytes. The exact number of bytes available depends on the host system.
- Reference macro variables inside or outside of a macro by prefixing the macro variable's name with an ampersand (&).
- When referencing macro variables, you can put a period immediately after the name. The period can be used to avoid any confusion that might occur when you are concatenating text onto the resolved value of the macro variable.
- The macro processor replaces, or substitutes, the name of the symbolic variable with its value.

Another important difference between DATA step variables and macro variables is that there *are* reserved names for macro variables, macro names, and macro labels. As a general rule, you should avoid names that have other usages within the macro language. For example, since there is a %EVAL function, you cannot create a macro named %EVAL. Also, there are a number of automatic macro variables (see Sections 2.6 and 8.3), and most of them start with the letters SYS. You should not create a macro variable with the same name as an automatic macro variable. In addition, when you name your macro variable, you should avoid using names that start with SYS altogether.

**SEE ALSO:** Appendix 1, “Reserved Words in the Macro Facility” in *SAS 9.4 Macro Language Reference, Fourth Edition* provides a full list of reserved names and words.

---

## 2.2 Defining Macro Variables

One of the easiest and most commonly used ways to define a macro variable is through the %LET statement. (Macro language statements always start with a %). This statement works much like an assignment statement in the DATA step.

The %LET statement is followed by the macro variable name, an equal sign (=), and then the text value to be assigned to the macro variable. Notice in the syntax of the %LET statement that quotation marks are not used:

**SYNTAX:**

```
%LET macro-variable-name = text-or-text-value;
```

Unlike data set variables, macro variables are not distinguished as either *character* or *numeric*; they always store only text. While learning the macro language, SAS programmers familiar with DATA set variables might find it easier to think of them as character variables. Because SAS knows that whatever is to the right of the equal sign is to be assigned to the macro variable, quotes are unnecessary. Indeed, when they are used they become part of the value that is stored.

The following statement assigns the text string *clinics* to the macro variable DSN:

```
%LET dsn = clinics;
```

If the %LET statement is outside of any macro, then its value will be available throughout the entire program, and it is said in that case to be a *global macro variable*. On the other hand, if the macro variable is defined inside a macro, then it might be local, in which case its value will be available within only that macro. For more detail on these issues, see Sections 1.3, 5.4.2, and 14.5.

**SEE ALSO:** A short introduction to the macro language and the use of the %LET statement can be found in Sissing (2014). Ottesen and Goldstein (2015) discuss the use of macro variables in PROC TEMPLATE and the Graphics Template Language.

## 2.3 Using Macro Variables

You can use SAS Program 2.3a below to determine the contents and general form of the data set WORK.CLINICS. It uses PROC CONTENTS and PROC PRINT, limiting the print to the first ten observations.

### Program 2.3a: An Ungeneralized Program

```
title1 'Data Set clinics';
proc contents data=clinics;
run;

proc print data=clinics (obs=10);
run;
```

Macro variables are especially useful when you want to generalize your programs. Program 2.3a works for only one data set (WORK.CLINICS). If you want to apply it to a different data set, then you will need to edit it in three different places. Doing so is trivial in this situation, but edits of existing production programs can be a serious problem in actual applications.

Fortunately, the program can be rewritten and is generalized in Program 2.3b.

### Program 2.3b: A Generalized Program

```
%let dsn = clinics; ①
title "data set &dsn"; ②
proc contents data=&dsn; ③
run;

proc print data=&dsn ③ (obs=10);
run;
```

- ① The %LET statement defines the macro variable.
- ② Macro variables inside double quotes will be resolved.
- ③ A macro variable (&DSN) replaces the data set name.

To change the data set name in Program 2.3b, you still need to edit the %LET statement. At least doing so is now a simpler task. Examples later in the book (see Chapter 4, “Macro Parameters”) show easier and even more general ways of accomplishing this same sort of task.

Notice that, in the rewritten code, quotes in the TITLE statement ② were changed from single to double quotes. Macro variables that appear inside a quoted string will not be resolved unless you use double quotes (""). This is because within single quotes, ampersands and percent signs are interpreted as just text, not as macro language triggers.

You can change the value of a macro variable simply by issuing a new %LET statement. The most recent definition of &DSN will be used at any given time.

The period, or dot, can be used to terminate the name of the unresolved macro variable. Although the macro variable name &DSN is interchangeable with &DSN., most macro programmers add the period only when it is needed to minimize confusion.

The macro language does not support the concept of a missing value. Unlike data set variables, which at least contain a missing value, macro variables can actually contain nothing. In the macro language this is often referred to as a *null value*. Each of the %LET statements defining the macro variable &NADA are equivalent, and in this case the value stored in &NADA is actually nothing—a null value:

```
%let nada =;
%let nada =      ;
```

The %LET statement does not typically store nonembedded blanks, so each of these definitions of the macro variable &DSN will be the same:

```
%let dsn =clinics;
%let dsn =      clinics      ;
```

If you do want to store a blank, as opposed to a null value, then you will need to use one of several quoting functions, and these functions are described in Section 7.1.

In Program 2.3c PROC CONTENTS and PROC PRINT might be useful to you during the debugging phase of the development of a larger program. It will be more flexible if you set up the code so that it can be turned on or off at the flip of a debugging switch. The code in Program 2.3c will execute exactly as it did in Program 2.3b, because the macro variable &DEBUG ④ has been assigned a null value (remember that null values are less than blank values; they truly are nothing).

#### Program 2.3c: Use of &DEBUG to Comment Code

```
%let dsn = clinics;
%let debug =; ④
&debug title "Data Set &dsn";
&debug proc contents data=&dsn;
&debug run;
&debug proc print data=&dsn (obs=10);
&debug run;
```

In each of these statements, &DEBUG resolves to a null value, and the statements are executed just as they were in Program 2.3b. It is as if each of the &DEBUG variables were not even there. Notice that the blank between the &DEBUG (which resolves to a null value) and the statement that follows it, is preserved.

```
title "Data Set clinics";
proc contents data=clinics;
run;
proc print data=clinics (obs=10);
run;
```

However, when you redefine &DEBUG as an asterisk as it is in Program 2.3d ⑤, each of the statements becomes an asterisk-style comment.

#### Program 2.3d: Use of &DEBUG to Comment Code

```
%let dsn = clinics;
%let debug =*; ⑤
&debug title "Data Set &dsn";
&debug proc contents data=&dsn;
&debug run;
&debug proc print data=&dsn (obs=10);
&debug run;
```

The resolved code for Program 2.3d becomes the following:

```
* title "Data Set clinics";
* proc contents data=clinics;
*   run;
* proc print data=clinics (obs=10);
*   run;
```

The /\* style comment can also be used similarly; however, because of the way that the macro language is interpreted, extra care must be exercised. This time the macro variable &DEBUG will take on the value of “/” when the code is to be commented, as shown in Program 2.3e.

#### Program 2.3e: Use of &DEBUG to Comment Code

```
%let dsn = clinics;
%let debug = /;
&debug* *;
proc contents data=&dsn;
title "Data Set &dsn";
run;
proc print data=&dsn (obs=10);
run;
*/ *;
```

When &DEBUG is null (blank or empty), a valid \* style comment appears before and after the block of code; otherwise, when &DEBUG is set to a slash (/), as is shown in Program 2.3e, the block of code becomes surrounded by a /\* --- \*/ style comment. Notice the use of the asterisk-style comments to protect the syntax when &DEBUG is set to null.

This technique would, of course, fail if there were already /\* --- \*/ style comments within the block of code to be commented (this style of comment cannot be nested). You might think of changing the example to store /\* in &DEBUG:

```
%let dsn = clinics;
%let debug = /*;
&debug *;
```

But this would also fail, because the %LET statement would not store the anticipated values. (The /\* symbols have special meaning and are not stored. See Section 7.1).

**MORE INFORMATION:** For other ways to use macros to comment out blocks of code, see Section 3.1.2.

**SEE ALSO:** A quick summary of the use of macro variables can be found in Werner (2014). The DATA step debugger is switched on and off with the use of a macro variable described in Riba (2000).

## 2.4 Displaying Macro Variables by Using the %PUT Statement

The %PUT statement, which is analogous to the DATA step PUT statement, writes text and the current values of macro variables to the SAS Log. As a macro statement, the %PUT statement (unlike the PUT statement) does not need to be inside of a DATA step:

```
%let dsn = clinics;

%put ***** selected data set is &dsn;
```

Notice that, unlike the PUT statement, the text string is not enclosed in quotes. The quotes are unnecessary, because, unlike in the DATA step, the macro facility does not need to distinguish between variable names

and character strings. Everything is a text string, a macro language reference, or some other macro language trigger. The macro language can easily recognize macro variables, for instance, because they are preceded by an ampersand.

The %PUT statement can be especially useful when you are debugging macro code. If you want to write the macro variable name, as well as the value of the macro variable, then you can follow the & with an equal sign in the %PUT statement, which resembles named output in the DATA step's PUT statement:

```
%put &=dsn;
```

The result of the %PUT statement is written to the SAS Log:

```
49   %let dsn = clinics;
50
51   %put ***** selected data set is &dsn;
***** selected data set is clinics
52   %put &=dsn;
DSN=clinics
```

Because macro statements are executed before the DATA step statements are even compiled (see Figure 1.4d), you might need to get used to their execution order. The DATA step in Program 2.4a is also rather silly, but is included to illustrate this point. It contains both a %PUT and a PUT statement, and both are inside of a DO loop. The associated SAS Log illustrates the differences between these two statements. The distinction between the %PUT and the PUT statements is an important one.

#### **Program 2.4a: Comparing PUT and %PUT Statements**

```
data _null_;
do j = 1 to 5;
  put j ' Placed by PUT';
  %put j ' Placed by macro PUT'; ①
end;
run;
```

Examine this DATA step, particularly the PUT and the %PUT statement, noting that the PUT statement is first. With regard to these two statements, answer the following questions:

- Which of these two statements will be executed first?
- How many times will each of these statements be executed?
- The DO loop index variable is j, and both the PUT and %PUT statements have an unquoted j; how will this j be interpreted in each statement?

When this DATA step is submitted for execution, it must first be parsed and compiled (remember the timing diagram—Figure 1.4d). For this particular example, let's parse the step at the statement level. The first three statements do not contain macro language elements, and they are scanned and compiled without invoking the macro facility. However, the fourth statement is a %PUT ① statement. Because of the % sign, the macro facility is triggered, and the %PUT is immediately executed by the macro facility. At this point the DATA step has not yet been compiled. The DO loop has not been executed and the PUT has not been executed. After the %PUT is executed ②, the remaining statements in the DATA step are compiled.

#### **SAS Log 2.4a: Showing Execution of the DATA Step**

```
1   data _null_;
2     do j = 1 to 5;
3       put j ' Placed by PUT';
4       %put j ' Placed by macro PUT';
j ' Placed by macro PUT' ②
5     end;
6   run;
```

```

1 Placed by PUT ❸
2 Placed by PUT
3 Placed by PUT
4 Placed by PUT
5 Placed by PUT
NOTE: The DATA statement used 1.26 seconds.

```

- ❷ Notice that the %PUT statement is executed (and writes to the SAS Log) as soon as it is encountered; because there are quotes in the statement, they are also displayed. Moreover, %PUT statement does not recognize the j as a variable name, so it just includes it as part of the rest of the text that is to be displayed.
- ❸ Now the DO loop and the associated PUT statement are executed five times. In this PUT statement the j is seen as a variable, and the value of j is printed in the SAS Log.

The code making up the compiled DATA step has no macro language elements, so the DATA step code that is executed is as follows:

```

data _null_;
do j = 1 to 5;
  put j ' Placed by PUT';
end;
run;

```

So, although it initially appears that the %PUT statement is inside of the DO loop, it actually is not. Remember, it is all about the timing: The macro language executes first. In this case it executes before the DATA step exists and long before the DO loop executes.

When you are creating macro variables, it is easy to lose track of which macro variables have been defined and what values they contain. The %PUT statement can be used to help you determine the values taken on by macro variables. Fortunately, there are several options that can be used on the %PUT statement. When the following %PUT statement options are used, without specifying other text, information about your symbol tables is written to the SAS Log:

**\_all\_** lists all macro variables in all referencing environments.

**\_automatic\_** lists all of the macro variables that are automatically defined by SAS. The list of these variables can vary by site, operating system, and version of SAS. Automatic macro variables are described further in Section 2.6, “Using Automatic Macro Variables.”

**\_global\_** lists user-created macro variables that will be available in all of the referencing environments.

**\_local\_** lists user-defined macro variables that are available only in the current or most local referencing environment. When this option is not used from within a macro, it will be unable to list any macro variables.

**\_user\_** lists all of the user-created macro variables in each of the referencing environments. This option can be especially useful during the debugging process for complicated macros.

When any of these options are used with the %PUT statement, the SAS Log will indicate the macro variable’s symbol table ❹, its name ❺, and its value ❻. The next two statements create the SAS Log that follows:

**Program 2.4b: Using %PUT to Show Macro Variable Values**

```
%LET dsn = clinics;
%PUT _all_;
```

A partial listing of the SAS Log shows the one user-defined macro variable &DSN and some of the automatic macro variables defined by SAS.

**SAS Log 2.4b: Showing Macro Variable Values**

```
1   %let dsn = clinics;
2   %put _all_;
GLOBAL④ DSN⑤ clinics⑥
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 15OCT14
AUTOMATIC SYSDATE9 15OCT2014
AUTOMATIC SYSDAY Wednesday

. . . Portions of the SAS Log not shown . . .
```

The macro variable &DSN ⑤ has been created in the GLOBAL symbol table ④ with a value of *clinics* ⑥.

Messages that mimic those written to the SAS Log by the Base SAS language can also be generated by using the %PUT statement. When the %PUT statement is followed by ERROR, WARNING, or NOTE, then the text associated with the %PUT will be written to the SAS Log in the color appropriate for that message type. Under the default settings in an interactive environment, the following %PUT statement generates a red error message in the SAS Log:

```
%PUT ERROR: Files were not copied as expected.;
```

The keywords must be in all caps, must immediately follow the %PUT statement, and must be immediately followed by a colon. If the keyword is followed by a dash, the message is written to the SAS Log without the keyword:

```
1   %put ERROR: test;
ERROR: test
2   %put ERROR- test2;
      test2
```

**SEE ALSO:** The additional options for the %PUT statement are described fully in *SAS 9.4 Macro Language Reference, Fourth Edition* (Chapter 19).

## 2.5 Resolving Macro Variables

Prior to the execution of the SAS code, macro variables are resolved. The resolved values are then substituted back into the code. It is, therefore, important to understand the rules associated with how macro variables are resolved.

The use of single macro variables, as shown in the following example, is fairly straightforward:

#### **Program 2.5: Resolution of Single Macro Variables**

```
%let dsn = IAX;
title "Looking at the &dsn data";
proc contents data= &dsn;
run;

proc print data= &dsn;
run;
```

The macro variables in this code will be resolved, and the resultant code will be executed:

```
title "Looking at the IAX data";
proc contents data= IAX;
run;

proc print data= IAX;
run;
```

Remember that, when the resolved value of a macro variable is to be within quotes (as in the TITLE statement), you must use double, not single, quotes.

It is when you start combining macro variables with text and other macro variables that the fun begins. It is not at all unusual for macro variables to be used as building blocks to create other items. Macro variables can be concatenated to text or even to other macro variables.

---

#### **2.5.1 Using the Macro Variable as a Suffix**

You can append a macro variable to SAS code that includes variables, text strings, and data set names. When resolved, the value of the macro variable is concatenated to the string that precedes it. In the macro language there is no concatenation operator. It is unnecessary because there are only text and macro language elements. Once the macro language elements have been resolved, there is only text.

In Program 2.5.1 the macro variable &DSN contains the text characters IAX. In each usage these letters are to be appended onto the word *protocol* to form a data set name. Once &DSN is resolved, the data set name becomes *protocolIAX*.

#### **Program 2.5.1: Use of a Macro Variable as a Suffix**

```
%let dsn = IAX;
title "Looking at protocol&dsn";
proc contents data= protocol&dsn;
run;

proc print data= protocol&dsn;
run;
```

The executed code becomes the following:

```
title "Looking at protocolIAX";
proc contents data= protocolIAX;
run;

proc print data= protocolIAX;
run;
```

In order for the macro word scanner to determine the macro token, it must be able to detect the start and the end of the token. For macro variables the start of the token is clearly indicated by the ampersand (&). The end of a macro variable token is detected when a character is encountered that cannot be a part of the name.

Because the macro facility can easily determine by the & character where the macro variable name starts, there is no confusion when the macro variable is used as a suffix (as in Program 2.5.1), for the name is followed by either a semicolon or a double quote. The determination of where the macro variable name ends is not always so easy when you are using the macro variable as a prefix.

## 2.5.2 Using the Macro Variable as a Prefix

A macro variable may also precede portions of SAS code. Because there are no concatenation operators in the macro language, text that is to be appended to the end of the resolved value of a macro variable can become confused with the macro variable name. To separate a macro variable name from the text that follows it, a macro variable reference can be followed by a period to designate the end of the variable name.

In Program 2.5.2a, because the parser cannot determine where the name of the macro variable ends, the token holding the macro variable &DSN cannot be determined. The scanner will incorrectly determine that &DSNPROTOCOL is the name of the macro variable that is to be resolved.

### Program 2.5.2a: Indeterminable Macro Tokens

```
%let dsn = IAX;
title "Looking at &dsnprotocol";
proc contents data= &dsnprotocol;
run;

proc print data= &dsnprotocol;
run;
```

If the programmer is lucky, then the program will fail, and a warning will be written to the SAS Log, indicating that the macro variable &DSNPROTOCOL could not be resolved:

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

Of course if the programmer is unlucky, then the macro variable &DSNPROTOCOL will exist, and the incorrect macro variable will be resolved. Section 2.5.4 discusses the resolution failure in Program 2.5.2a in more detail.

By following the macro variable name with a dot, you avoid any ambiguity. In Program 2.5.2b the dot, or period, immediately follows each instance of &DSN, and the dot is used by the word scanner to determine the end of the macro variable name.

### Program 2.5.2b: Use of a Dot after Macro Variable Names

```
%let dsn = IAX;
title "Looking at &dsn.protocol";
proc contents data= &dsn.protocol;
run;

proc print data= &dsn.protocol;
run;
```

When a dot is used as a part of the macro variable name, it is also absorbed when the macro variable is resolved. After macro variable resolution, the submitted code in Program 2.5.2b becomes the code in Program 2.5.2c.

### Program 2.5.2c: Resolved Code

```
title "Looking at IAXprotocol";
proc contents data= IAXprotocol;
run;

proc print data= IAXprotocol;
run;
```

You might want to think of the &DSN. form (with the dot) of the macro variable as its formal name, whereas &DSN (without the dot) is less formal. There is no consensus among macro programmers as to which form should be preferred. You can always include the dot, and therefore not worry about whether or not it is needed, or you can learn when it is needed and include the dot only as it is required.

Sometimes the first character of the string that is to be appended to the macro variable's value is a period. As shown in the previous code, the period will be absorbed during the resolution process and will not appear in the resolved text. To get around this, you can use a double period (..) when a single period (.) is desired in the final code.

This is illustrated in Program 2.5.2d, where we would like to append two macro variables, a *libref* and the name of a data set, in such a way as to have them separated by a period. By using two periods between the two macro variables, you ensure that the resultant code will have the two resolved values separated by a period.

#### **Program 2.5.2d: Use of a Double Dot**

```
%let lib = macro3;
%let dsn = clinics;

title "Looking at &lib..&dsn";
proc contents data= &lib..&dsn;
run;

proc print data= &lib..&dsn;
run;
```

The macro references *&lib..&dsn* will resolve to *macro3.clinics* in the program. The first period is seen as part of the macro variable name (&LIB.), and the second is just a text character.

---

### **2.5.3 Using Macro Variables as Building Blocks—Appending Macro Variables**

In addition to combining macro variables with text, you can join two or more macro variables to form a single result. Consider, for example, the following four macro variable definitions:

```
%let dsn=CLINICS;
%let n=5;
%let dsn5=MAY;
%let dset=DSN;
```

Using the rules that are discussed in the previous section, you can use various combinations of these macro variables as building blocks, and they will resolve as shown in Table 2.5.3a.

**Table 2.5.3a: Macro Variables as Building Blocks**

Combination	Value Resolved To
&DSN&N	CLINICS5
&DSN.&N	CLINICS5
&DSN..&N	CLINICS.5

The macro processor scans for macro variables and resolves them as encountered. In each of the previous macro variable combinations, the resolution is possible in a single pass. When two or more ampersands (&) appear next to each other, successive passes, or rescans, are required to make the final resolution. You can think of the double ampersand (&&) as a special reference that resolves to a single ampersand. This is demonstrated in Table 2.5.3b.

**Table 2.5.3b: Double- and Triple-Ampersand Macro Variable References**

Combination	Resolved Value after First Scan	Resolved Value after Second Scan
&&DSN&N	&DSN5	MAY
&&&DSET	&DSN	CLINICS
&&&DSN&N	&CLINICS5	&CLINICS5

The macro variable reference &CLINICS5 does not exist, so the following message is written to the SAS Log:

WARNING: Apparent symbolic reference CLINICS5 not resolved.

A common mistake is to assume that the resolution process proceeds from right to left as follows:

**Incorrect:** &&DSN&N → &CLINICS5

It is very important to remember that resolution is from left to right. Consequently, the double ampersand (&&) must be resolved to a single ampersand (&) before the &DSN5 is resolved:

**Correct:** &&DSN&N → &DSN5 → MAY

For the same reasons, &&&DSN&N is taken as && &DSN &N, which resolves to &CLINICS5. This macro variable does not exist on the symbol table and therefore remains unresolved, so a warning is written to the SAS Log. For a summary of the resolution process, see Table 2.5.3c.

**Table 2.5.3c: Macro Variable Rescan Resolution**

Scanning Pass	&DSN&N <u>&amp;DSN</u> <u>&amp;N</u>	&&DSN&N <u>&amp;&amp;</u> <u>DSN</u> <u>&amp;N</u>	&&&DSN&N <u>&amp;&amp;&amp;</u> <u>DSN</u> <u>&amp;N</u>	&&&DSET <u>&amp;&amp;&amp;</u> <u>DSET</u>	&&&DSET&N <u>&amp;&amp;&amp;</u> <u>DSET</u> <u>&amp;N</u>
First	CLINICS5	&DSN5	&CLINICS5	&DSN	&DSN5
Second	—	MAY	&CLINICS5	CLINICS	MAY

Using multiple ampersands establishes the ability to create a type of vector or an array of macro variables. Examples that use this notation are shown in Program 6.4.1b and various other sections in this book.

**MORE INFORMATION:** Extensive use is made of the macro variable form &&VAR&I in list processing. Numerous examples can be found in the latter portions of this book (see Chapters 11 and 12). The form &&&VAR&I is less commonly used, but it enables you to create a type of doubly subscripted list (see Section 14.2.3).

**SEE ALSO:** Several examples and variations of the process of resolving macro variables can be found in Yindra (1998). Matise (2015) gives a careful explanation of the resolution process, especially for references involving multiple ampersands.

## 2.5.4 Understanding Results When Macro References Are Not Resolved

In Program 2.5.2a, the macro variable &DSN is incorrectly used as a prefix to the constant text PROTOCOL, resulting in the text string &DSNPROTOCOL, which will in turn be interpreted as a macro language reference. The &DSNPROTOCOL becomes a token that is passed to the macro facility for resolution. Because the macro variable &DSNPROTOCOL does not exist, it cannot be resolved. If the program is executed, then a series of warnings and errors are generated and written to SAS Log 2.5.4.

**SAS Log 2.5.4: Showing an Unresolved Macro Reference**

```

1  %let dsn = IAX;
2  title "Looking at &dsnprotocol";
WARNING: Apparent symbolic reference DSNPROTOCOL not resolved.❶
3  proc contents data= &dsnprotocol; ❷
-
22
200
NOTE: Writing HTML Body file: sashtml.htm
WARNING: Apparent symbolic reference DSNPROTOCOL not resolved. ❶
ERROR: File WORK.DSNPROTOCOL.DATA does not exist.
ERROR 22-322: Expecting a name. ❷
ERROR 200-322: The symbol is not recognized and will be ignored.
4      run;

```

- ❶ The unresolvable macro variable causes this warning to be issued. Since it cannot be resolved, the token cannot be replaced and is left in place with the ampersand (&) marked to prevent further attempts at resolution. Because the ampersand (&) has not been removed by the macro language, it remains in the code, where it will probably cause additional problems.
- ❷ Because the macro reference was unresolved, the ampersand (&) remains and is now part of the data set name. But data set names cannot start with an &, so an error is also issued in the SAS Log.

When a macro variable cannot be resolved, the ampersand (&) will not be absorbed by the macro facility. Usually, this “left behind” ampersand will cause syntax problems when the statement is compiled.

## 2.6 Using Automatic Macro Variables

Several macro variables are automatically created for you by the macro processor. You can use these variables as you would any other macro variable. Although the list of available automatic macro variables can vary quite a bit, depending on your release of SAS and your operating environment, a number of them can be very useful in your daily programming, and a few of these are shown in this section.

Some automatic macro variables are write-protected and cannot be modified by the user; others can be changed. As a general rule, be cautious about modifying the values of any automatic macro variables. I am not saying do not do so, but just to be careful. Because of possible conflicts with automatic macro variables, you should either never create a macro variable with a name starting with the letters SYS, or you should at least be very careful when doing so.

There are a few automatic macro variables whose names start with other letters than SYS. When a PROC SQL step is executed, a series of macro variables starting with SQL are generated (see Section 6.2.4). Simply because of their availability, some SAS products will also generate automatic macro variables. The SAS Log for Program 2.4b was generated using a version of SAS that included SAS/AF and SAS/FSP, and both of these products generated automatic macro variables starting with the letters AF and FSP, respectively.

**MORE INFORMATION:** Additional automatic macro variables are introduced and discussed in Section 8.3.

**SEE ALSO:** The *SAS 9.4 Macro Language: Reference, Fourth Edition* (Table 3.1) lists many of the automatic variables. A list and description of selected automatic macro variables is provided by First (2001b).

### 2.6.1 &SYSDATE, &SYSDATE9, &SYSDAY, and &SYSTIME

At the start of the current SAS session (for batch execution this might also be known as a “job”), the following four automatic macro variables are loaded; they note the date, day, and time of the start of the session:

#### SYSDATE

Date that the session began executing (DATE7. form)

#### SYSDATE9

Date that the session began executing displayed with a four-digit year (DATE9. form)

#### SYSDAY

Day of the week that the session began executing

#### SYSTIME

Time of the day that the SAS session began executing (TIME8. form)

These macro variables can be used to insert date and datetime values into various aspects of your program. These might include incorporation of the current date into a title:

```
title3 "Processing Initiated at: &sysdate9 &systime";
```

The resulting title will include the date and time that the current SAS session was initiated. You can also use these macro variables in programming statements to insert values into the data itself:

```
* Add time stamp when updating data;
data new;
  set old;
  if batch = 2 then do;
    conc=2.5;
    concmoddate=&sysdate9"d"; ①
    concmodtime=&systime"t";
  end;
run;
```

- ① The &SYSDATE9 and &SYSTIME values are written to data set variables as SAS date and time values by converting the values of the macro variables into date and time constants. The macro variables are resolved before the conversion to SAS date and time constants. This resolution and execution sequence enables the &SYSDATE9 and &SYSTIME values to be converted from text values to SAS date and time values by treating them as date/time constants.

**MORE INFORMATION:** The use of macro variables in DATA step assignment statements is discussed further in Section 6.3.1.

## 2.6.2 &SYSLAST and &SYSDSN

&SYSLAST stores the name of the last data set that was created or modified. You can use this variable as you would use any data set name. This variable is useful when you are generating dynamic code and you do not necessarily know the name of the data set that was just created.

&SYSDSN also stores the name of the last modified data set; however, the name is stored in two words (one word if the *libref* uses all 8 characters), with the library first.

#### SYSLAST

Name of the last SAS data set created with the library and data set name separated with a period (.)

#### SYSDSN

Name of the last SAS data set created with the library stored in the first 8 characters and the data set name starting in the 9<sup>th</sup> position

If no data sets have been created in the current SAS session, then the macro variables will contain the value `_NULL_`:

```
data myclass;
  set sashelp.class;
  run;

%put &syslast;
%put &sysdsn;
```

The resulting SAS Log is given:

```
47 %put &syslast;
SYSLAST=WORK.MYCLASS
48 %put &sysdsn;
SYSDSN=WORK      MYCLASS
```

Notice that the name of the data set stored in `&SYSLAST` includes the associated libref.

**SEE ALSO:** The automatic macro variable `&SYSLAST` is used by Farriola (2015).

### 2.6.3 &SYSERR and &SYSCC

It is sometimes handy to be able to determine whether a particular PROC or DATA step executed successfully. Each step has a return code that measures the existence and severity of any errors. The return code is stored in `&SYSERR`, where it can be checked during the execution of the job.

Because each step determines a new value for `&SYSERR`, it is reset at each step boundary.

#### SYSERR

Stores the return codes of PROC and DATA steps

#### SYSCC

Stores the overall (highest) session return code

The system condition code (`&SYSCC`) is not reset at step boundaries and can be used to determine the error conditions across steps.

The various steps (procedures and DATA steps) do not all return the same set of values for `&SYSERR`. The value that is assigned to `&SYSERR` depends on both the type of error and the type of step. You might need to do some experimenting to determine the values returned for the step that you are interested in. As a general rule, successful steps assign a 0 to `&SYSERR`.

The following PROC DATASETS copies the data sets in the COMBINE library into the COMBTEMP library. In a shared data environment, the copy might not be fully successful if another user currently has write access to a data set in the library COMBINE. When this happens, `&SYSERR` will take on a value of greater than 0. Writing the value of `&SYSERR` to the SAS Log enables you to detect the success or failure of the copy:

```
* Copy the current version of the
* COMBINE files to COMBTEMP;
proc datasets memtype=data;
  copy in=combine out=combtemp;
  quit;
%put SYSERR is &syserr;
```

**MORE INFORMATION:** Section 8.3.2 has a more in-depth discussion about &SYSERR, and Program 8.3.2a shows a more sophisticated version of this example.

**SEE ALSO:** Beverly (2001) also discusses the automatic macro variable &SYSERR. The documentation of &SYSERR gives a list of common return codes. Billings (2015) describes error trapping using &SYSERR.

## 2.6.4 &SYSRC

The SYSRC macro variable captures the last return code from system operations that are executed during a system subsession, which would have been spawned with the use of something like the X command, X statement, or the %SYSEXEC macro statement.

### SYSRC

Indicates the last return code from your operating environment

The value will be an integer, but it will vary according to the operating environment. Successful operations will not always return a value of 0. The SAS Companion for your operating system has additional information about what values can be returned to &SYSRC.

## 2.6.5 &SYSSITE, &SYSSCP, &SYSSCPL, and &SYSUSERID

These macro variables contain information about your site and operating system. Primarily, you can use them when a particular SAS job or application might be executed in multiple environments and its behavior needs to be altered accordingly.

### SYSSITE

Contains the current site number

### SYSSCP

Gives the name of the host operating environment

### SYSSCPL

In some operating environments can be more specific than &SYSSCP.

### SYSUSERID

For operating systems that require a login user ID, the user ID is placed in this macro variable (&SYSUSERID receives the value “default” when the operating system has not captured the user identification)

This snippet of the SAS Log was generated on a machine running Windows 7 and SAS 9.4.

```
23  %put &=sysuserid;
SYSUSERID=Art
24  %put &=syssite &=sysscp &=sysscpl;
SYSSITE=0123456789 SYSSCP=WIN SYSSCPL=X64_7PRO
```

The site number returned by &SYSSITE is usually required by SAS Technical Support when you are requesting help to solve a problem.

**SEE ALSO:** Michelsen (2014) uses &SYSSCPL to detect the operating system. &SYSSITE and &SYSSCPL are documented in *SAS 9.4 Macro Language: Reference, Fourth Edition* (Chapter 14). Davis (1997) and Hessel (1998) both include examples that use &SYSSCP. Glass and Hadden (2016) load a DATA set variable using &SYSUSERID. Rasheed and Vijayarangan (2014) use &SYSUSERID, as well as &SYSPROCESSNAME, in a program that emails the user with its status.

## 2.6.6 &SYSMACRONAME

When a macro is executing, &SYSMACRONAME will contain the name of that macro. This macro variable can be very useful when you would like to document the progress of your application. If &SYSMACRONAME is used in open code (outside of any macro definitions), it will have a null value, and when macros are nested, &SYSMACRONAME will contain the name of the innermost macro.

**MORE INFORMATION:** &SYSMACRONAME is described in more detail in Section 8.3.5.

## 2.7 Removing Macro Variables

As you work with macro variables, you might occasionally want to remove them from the global symbol table. You can use the %LET statement to reset the value to null; however, doing so does not remove the variable. Fortunately, the macro statement %SYMDEL can be used to totally remove the global macro variable:

### SYNTAX:

```
%SYMDEL macro_variable_list;
```

In the following example the macro variables &NADA and &DSN are removed from the global symbol table:

```
%symdel nada dsn;
```

Notice that the macro variable names do *not* include the ampersands. This statement expects macro variable names. If you used an ampersand, then the resolved value would also need to be the name of a macro variable.

**MORE INFORMATION:** Interesting things can be made to happen when %SYMDEL is used in ways other than specified above (see Section 8.2.1).

**SEE ALSO:** Langston (2015b) demonstrates the use of %SYMDEL. First (2001a) uses %SYMDEL to delete a list of macro variables.

## 2.8 Testing Your Knowledge with Chapter Exercises

To test your knowledge, complete the exercises provided below.

### Program 2.8: Exercise Program

```
*****;
**** The class data set, CLINICS, contains 80 ****;
**** observations and 20 variables. The following ****;
**** program will be used to complete the exercises****;
**** in this chapter. ****;
*****;

title1 'Counts of Gender vs. Years of Education';
proc tabulate data=macro3.clinics;
  class sex edu;
  table sex=' ',edu*n=' '/box=sex;
run;
```

```
title1 'Statistics on Heights for Each Gender';
proc univariate data=macro3.clinics;
  class sex;
  var ht;
run;
```

1. Rewrite Program 2.8 by adding at least one macro variable (%LET). Save your program so that you can use it again for your Chapter 3 exercises.
2. (True/False)—Macro variables will not be resolved unless you enclose them in single quotation marks (').
3. Given the following macro variable definitions, how will the macro variable combinations be resolved? Take a guess first, and then use the %PUT statement to check your answer:

```
%let dsn=clinic;
%let I = 3;
%let dsn3 = studydrg;
%let lib = sasuser;
%let b = dsn;
```

&lib&dsn	→	_____	&dsn&I	→	_____
&lib.&dsn	→	_____	&&dsn&I	→	_____
&lib..&dsn	→	_____	&dsn.&I	→	_____
&&bb	→	_____	&&&b	→	_____

4. As extra credit for Item 3, using the above macro variable definitions and only the macro variables &DSN and &I, indicate what combination of &DSN and &I will resolve to CLINIC.STUDYDRG. Use a %PUT statement to show your answers in the SAS Log.
5. What are automatic macro variables?
6. What is the value stored in &SYSDATE9? Does it contain the current date?

# Chapter 3: Defining and Using Macros

<b>3.1 Creating a Macro .....</b>	<b>35</b>
3.1.1 Defining a Macro.....	37
3.1.2 Commenting a Block of Code with Use of %MACRO and %MEND .....	37
3.1.3 Using the /DES Macro Statement Option .....	39
<b>3.2 Invoking a Macro .....</b>	<b>39</b>
<b>3.3 Using System Options with the Macro Facility .....</b>	<b>40</b>
3.3.1 General Macro Options .....	41
3.3.2 Debugging Options .....	41
3.3.3 Use of the Debugging Options .....	41
3.3.4 Autocall Facility Options .....	42
<b>3.4 Testing Your Knowledge with Chapter Exercises .....</b>	<b>44</b>

This chapter introduces several of the simplest methods that you can use to create and use macros.

For the examples in this chapter and indeed for all of the code examples throughout the book, if you want to execute these sample programs, then be sure to follow the setup instructions. Remember that all of the data sets and programs are available for download, so you do not need to retype either the code or the data. For instructions on accessing and setting up the programs and data, see the “Example Code and Data” section within this edition’s “About This Book” front matter.

**SEE ALSO:** Introductions and overviews to the macro language can be found in Cohen (1998), Jaffee (1999), First (2001a, 2001b), Williams (2001), and Whitlock (1999a, 2001a).

---

## 3.1 Creating a Macro

All macros are created with use of the two macro language statements: %MACRO and %MEND. Like the DO and END statements in the DATA step, these two macro statements always come in pairs.

**SYNTAX:** The following syntax is for the %MACRO and %MEND pair.

```
%MACRO macro-name </options>;  
...macro text...  
%MEND <macro-name>;
```

Every macro definition begins with a macro statement (%MACRO), which must contain a name for the macro. The %MEND statement closes the macro definition. Although including the name of the macro on the %MEND statement is optional, it is a very good idea to do so. Not only does it help with documentation, but including the name can help protect you from certain types of programming errors.

Virtually any SAS code can be included between the %MACRO and %MEND statements. The macro definition can be made up of a single character (or even nothing, but that would not tend to be practical) up

to any number of characters. Taken together, these can be used to construct any combination of the following:

- Constant text
- Macro variables
- Macro program statements
- Macro expressions
- Macro functions

## **Constant Text**

The macro language treats constant text in much the same way as character strings are treated in the DATA step. Constant text is not evaluated, resolved, or even examined by the macro processor. It is just passed along. Constant text can include the following:

- SAS data set names
- SAS variable names
- SAS statements
- SAS steps (like DATA and PROC steps)
- Complete and partial SAS programs
- Any combination of the above

## **Macro Variables**

Macro variables are preceded in the macro text by an ampersand (&). Chapter 2 includes a more complete discussion of the definition and use of macro variables.

## **Macro Program Statements**

Macro program statements are evaluated and executed when the macro is called. Many macro program statements must be contained inside a macro and cannot exist in open code. Macro program statements, like DATA step statements, start with a keyword. Unlike other SAS statements, however, macro program statement keywords are preceded by a percent sign (%). Three macro program statements were introduced in Chapter 2. These included the %LET, %PUT, and %SYMDEL statements. Chapter 5 introduces a number of additional macro language statements.

## **Macro Expressions**

Macro expressions perform the same tasks in the macro language as DATA step expressions do in Base SAS. The difference, of course, is that the macro expressions involve the evaluation and assignment of macro variables rather than DATA step variables. They can be used to determine conditional branches in the processing flow of your program, and to create new value assignments. The use of macro expressions in conditional processing is described in Section 5.2. Most conditional processing is accomplished with the %IF statement, which can be used to make decisions based on macro expressions. Two example %IF statements are shown here:

```
%if &nobs = 0 %then %do;
%if &cond = bad %then %badobs;
```

## **Macro Functions**

Macro functions operate on macro variables and constant text. Some of the more useful character functions available in the DATA step have analogous macro functions. There are also a number of macro functions

that are unique to the macro language. Macro functions are described in Chapter 7. The %UPCASE function shown here is an example of a macro function:

```
%let upper = %upcase(&name);
```

Many DATA step functions can double as macro functions through the use of the %SYSFUNC macro function that is described in Section 7.4.2.

### 3.1.1 Defining a Macro

The easiest way to define a macro is to surround existing code with the %MACRO and %MEND statements. Remember that *every* macro definition must begin and end with these two statements; they come in pairs.

Program 3.1.1 creates a macro that looks at a data set. An existing program that contains a PROC CONTENTS and a PROC PRINT has been enclosed by the %MACRO and %MEND statements to form the macro %LOOK.

#### Program 3.1.1: Defining a Simple Macro

```
%macro look;
  title1 "Data Set &dsn";
  proc contents data=&dsn;
    run;

  proc print data=&dsn (obs=10);
    run;
%mend look;
%let dsn = sasclass.clinics;
```

When the %MACRO and %MEND statements are encountered, the macro facility processes the enclosed statements and text. Even though this process is often referred to as *macro compilation*, nonmacro text is not processed and is not compiled at this time. At this time macro variables are not resolved, nor are macro statements executed. Syntax errors in macro statements are detected during this processing phase; however, syntax errors and any other problems with the nonmacro code inside the macro will not be detected until the macro is actually executed (see Section 3.2). After successful compilation the macro definition is stored as an entry in the SASMACR catalog, which by default resides in the WORK library.

If your macro contains syntax errors in the *macro code*, your macro will not compile and will not be usable. You might see a message similar to the following in your SAS Log:

```
ERROR: A dummy macro will be compiled.
```

Remember that even though macros are processed by the macro facility when the %MACRO and %MEND statements are encountered, code that is enclosed in these two statements will not be executed until the macro is called (see Section 3.2). This means that the reference to &DSN in Program 3.1.1 will *not* be resolved during macro compilation.

Although there are no reserved words in Base SAS, there are reserved words in the macro language. You may not use macro language keywords, such as %LET or %PUT, as a macro name. A full list of the reserved words can be found in the documentation, but from a practical standpoint, if you do accidentally pick a name that is reserved, an obvious ERROR is generated in the SAS Log, which will alert you to the problem:

```
56 %macro let;
ERROR: Macro LET has been given a reserved name.
ERROR: A dummy macro will be compiled.
```

### 3.1.2 Commenting a Block of Code with Use of %MACRO and %MEND

When evaluating SAS programs, you might occasionally find blocks of code that have been effectively commented out by means of enclosure between %MACRO and %MEND statements. In effect, a macro is

defined, but it is never called. There is, however, a penalty associated with the use of this technique. This section will not only show you how to use this technique, but it will also help you to recognize the problem and to understand the issues associated with it.

Remember that, although the contents of a macro are compiled when the definition of the macro is encountered, the macro is not executed until the macro is actually called (see Section 3.2). This means that you can effectively comment out blocks of code by creating macros that you purposely do not call. This style of comment can even contain embedded comments that are defined by /\* ..... \*/.

The macro %COMMENT in the following code causes the second DATA step to be ignored, so long as the macro is not called:

```
data invert.specie;
  infile 'species.mas';
  length spcode $ 5 spname $ 40;
  input spcode 21-25 spname;
  run;

%macro comment;
* create the position data set;
data dbmaster.position;
  infile pos;
  length pos $ 2;
  input pos 1-2 /*coord 4-8*/;
  file errs;
  if _error_ then put '-1';
  run;
%mend comment;
```

The following example is a more common application of this type of macro. Here, the macro is used to comment out a debugging step:

```
data new;
  set big;
  ...code not shown...
  run;

%macro debugnew;
  title 'listing for NEW';
  proc print data=new (obs=5);
  run;
%mend debugnew;
```

During the debugging process, the macro %DEBUGNEW could be executed to view the contents of the data set NEW.

As was mentioned earlier, this technique is not without its downside. The macros %COMMENT and %DEBUGNEW will be read, compiled, and stored in the SASMACR catalog for every run of the job that contains them. If %COMMENT appears in multiple places within the program, it will be recompiled each time it is encountered, and additional computer resources will be expended. This can be somewhat inefficient. As a general rule, this is not a good use of macros. That said, the inefficiencies are generally quite minor, and the technique sure can be handy from time to time.

**MORE INFORMATION:** Although asterisk-style comments can generally be used within a macro without problems (as was done in the macro %COMMENT above), this style of comment can produce some surprises. Section 14.3.6 includes an example of a problem introduced by an asterisk-style comment within a macro.

**SEE ALSO:** The use of macros to comment out sections of code was first proposed by Grant (1994, 1998), and later by Stuelpner (1997), Flavin and Carpenter (2000), and Suhr (2001).

### 3.1.3 Using the /DES Macro Statement Option

The /DES option can be used on the %MACRO statement to add a description to the macro. This description, which can be up to 256 characters long, can be seen when viewing the catalog entry.

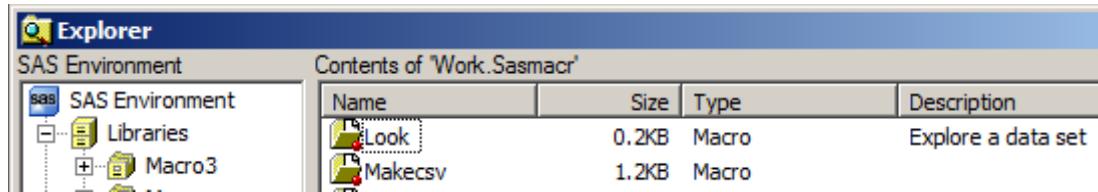
The %MACRO statement in Program 3.1.3 adds a short description to the %LOOK macro.

#### Program 3.1.3 (partial): Using the /DES Option on the %MACRO Statement

```
%macro look /des='Explore a data set';
```

Since macro definitions are stored in catalogs, the description can be surfaced whenever you are working with the catalog. In Figure 3.1.3 the description can be seen in the catalog window in the SAS Explorer.

**Figure 3.1.3: Viewing the Description for the %LOOK Macro**




---

## 3.2 Invoking a Macro

Macros are invoked, or called, by placing a % in front of the macro name. Unlike SAS statements, the macro call does not need to be followed by a semicolon. The macro LOOK in Section 3.1.1 is called by submitting %LOOK. If the macro %LOOK is called within a code stream, the five characters that make up the macro call are effectively replaced with the resolved values of the contents of the macro.

```
.... code not shown ....
%look
.... code not shown ....
```

When the macro call is encountered, it is expanded to the macro definition to become:

```
.... code not shown ....
title1 "Data Set sasclass.clinics ";
proc contents data= sasclass.clinics;
run;

proc print data= sasclass.clinics (obs=10);
run;
.... code not shown ....
```

Notice that the references to &DSN have been resolved, and it is this code that is executed.

When a macro call is encountered during processing, the macro definition is loaded into the macro facility. Here macro statements are processed and any nonmacro text, including text written by the macro, is sent to the input stack, where it is further processed. The contents of the macro are in effect substituted for the macro name in the program. Because this happens before any other program statements in this step are evaluated, you can use the macro to contain statement fragments or complete steps.

In the following example code, the macro %DEBUGNEW defines a PROC PRINT that you might want executed when you debug a program (Section 3.1.2 contains additional discussion of %DEBUGNEW).

**Program 3.2a: Commenting a Macro Call**

```

data new;
  set big;
  ...code not shown...
run;

%macro debugnew;
title 'listing for NEW';
proc print data=new (obs=5);
  run;
%mend debugnew;
① *%debugnew ② * uncomment to use debugnew;

```

- ❶ The macro call has been commented out with an asterisk-style comment. So long as the macro call remains commented out, the macro is not executed.
- ❷ A second asterisk-style comment is used to provide the semicolon for the commented macro.

You can also use a macro variable to control all debugging macros. In the following example, the macro variable &DEBUG takes on the value of \* when the debugging macros are turned off. The macro call in Program 3.2a becomes Program 3.2b.

**Program 3.2b: Using a Macro Variable to Comment a Macro Call**

```

%let debug = *; ③
data new;
  set big;
  ...code not shown...
run;

%macro debugnew;
title 'listing for NEW';
proc print data=new (obs=5);
  run;
%mend debugnew;
&debug ④ %debugnew      * uncomment to use debugnew;

```

- ❸ The call to %DEBUGNEW remains commented so long as &DEBUG contains an asterisk.
- ❹ To execute the macro %DEBUGNEW, the macro variable &DEBUG is cleared or assigned a null text string: %LET DEBUG =;

**MORE INFORMATION:** Additional examples of using asterisk-style comments to comment out macro calls, along with a caveat associated with their use, can be found in Section 5.1.4.

### 3.3 Using System Options with the Macro Facility

A number of SAS system options apply directly to the use of macros. They determine such things as whether the macro facility is available, how it is implemented, what debugging messages are to be printed, and if the macro autocall facility will be available.

**MORE INFORMATION:** Additional system options that interface with the macro language are discussed in Section 8.4.

---

### 3.3.1 General Macro Options

There are a number of system options that can be used to control the overall functionality of the macro facility. Three of the essential ones are shown here.

#### **MACRO**

The MACRO system option, which you must specify at the invocation of SAS (or in a configuration file), determines whether the macro facility is to be available. When you specify NOMACRO, you remove the ability to use the macro facility. As a general rule, this option is already turned on, and you rarely have to use it.

#### **MERROR and SERROR**

When you are using and writing macros, the debugging process is often difficult, even when these two options are turned on. Consequently, they will generally be left on whenever you are working with macros. MERROR enables the macro facility to display a warning in the SAS Log when a macro call is not resolved, and SERROR displays a warning in the SAS Log when a macro variable reference is not resolved.

---

### 3.3.2 Debugging Options

Debugging a macro can be, under the best of conditions, difficult. The SAS Log is often very cryptic when it presents error messages that deal with macros, macro code, or macro variables. You can use several options that are specifically designed for use when you are debugging macros.

#### **MPRINT**

SAS code that is generated by a macro is generally not displayed in the SAS Log. The MPRINT option displays the text or SAS statements that are generated by macro execution, one statement per line, with macro variable references resolved.

#### **MLOGIC**

Macros often are designed with logical branches based on %IF-%THEN/%ELSE statements and %DO loop executions. MLOGIC traces the macro logic and follows the pattern of execution. The resolved result of a %IF statement is displayed in the SAS Log as *true* or *false*. Macro invocation, macro completion, %DO loop evaluations, and macro parameter values are noted. This option is especially useful for nested macros.

#### **SYMBOLGEN**

When you use this option, a message is printed in the SAS Log whenever a macro variable is resolved. This option is very useful when you trace macro variable references with multiple ampersands (for example, &&DAT&I). SGEN can be used as an alias for this option.

#### **MFILE**

Similar to MPRINT, the MFILE option can be used to write out the resolved macro code to a file. Using this option requires a couple of extra steps, but the code that it creates will contain no macro references and can therefore be used to surface errors more easily.

---

### 3.3.3 Use of the Debugging Options

Although you can select which of the debugging options you want to use for any given situation, I tend to just turn on what I consider to be the three primary options (MPRINT, MLOGIC, and SYMBOLGEN), whenever I need to debug my macro code. Because these are on/off options, it is easy to turn them on and off. If you include the following two statements near the start of your programs, you will then be able to

turn these options on or off at any point in the debugging process by simply commenting and uncommenting the appropriate statement:

```
OPTIONS NOMPRINT NOMLOGIC NOSYMBOLGEN;
*OPTIONS MPRINT MLOGIC SYMBOLGEN;
```

When you program in the interactive mode or from the display manager, both statements are needed. This is because any option that you set (or turn on) remains in effect for the duration of that SAS session or until you reset it. Simply deleting the OPTIONS statement will not turn off the options. Of course, this is not an issue if you are debugging in batch mode.

You will probably use the MFILE option less often. However, it can be very useful when the first three debugging options do not highlight the macro error. To take advantage of this option, use a FILENAME statement with a *fileref* of MPRINT, and be sure to turn on both the MPRINT and MFILE options:

```
filename mprint 'c:\mymacros\maccode.sas';
options mprint mfile;
```

As subsequent macro language elements generate text, the resultant code is written to the file MACCODE.SAS. You can turn off the writing of these statements by issuing either a NOMPRINT or a NOMFILE option. Since only the resolved code is written, macro statements, like %LET and %PUT, and macro variables will not be shown.

Very often a simple inspection of this code is sufficient to reveal the problem; however, if this code is resubmitted directly, the SAS Log will be annotated as it always is when no macro elements are included in the program. Generally, the syntax or macro logic error is more easily revealed at this time.

**MORE INFORMATION:** A more flexible version of the OPTIONS statements shown in this section can be found in Program 5.1.1. Section 14.3.4 contains suggestions and things to look for when you are debugging your macros.

**SEE ALSO:** A number of SAS User Group International papers address various aspects of the debugging process. Although some are older papers, they can still provide good insight: Frankel (1991); Gilmore and Helwig (1990); O'Connor (1991); Phillips, Walgamotte, and Drummond (1993).

Chapter 10, “Macro Facility Error Messages and Debugging,” in *SAS 9.4 Macro Language: Reference, Fourth Edition (2015)*, discusses a variety of topics that relate to the debugging and troubleshooting of macros. Appendix 2 discusses the meanings of errors and warnings generated by the macro language.

Litzsinger, Brooks, and Riddle (2002), as well as Edgington and Zhou (2002), have examples that use the MPRINT and MFILE options to generate code.

Heaton (2001) shows some more sophisticated approaches to controlling (turning on and off) these debugging options.

### 3.3.4 Autocall Facility Options

There are three basic forms of macro libraries (for more details about how to create and use these three types of libraries, see Chapter 10). Of these three, the autocall library tends to be the most useful to most macro programmers. Although Chapter 10 has the real details, I feel that every macro programmer should have a basic idea of how and why to use the autocall facility. This short section serves as a brief introduction to the topic.

There is a tendency to reuse useful code. Useful macros might be used in multiple programs or even shared among programmers. This is okay, but through time, as changes are made, the different versions of this same macro often do not function the same way. As a result, code maintenance becomes a nightmare. My basic rule is that a macro definition, the %MACRO to %MEND, can never exist in more than one place or

program. However, the macro can still be used by multiple programs or multiple programmers if the macro definition resides in a macro library.

The autocall facility enables you to store macro definitions so that they can be called by any number of programs. Using this facility enables you to create libraries of macros that you define only once, while also making these macros available to all of your programs or even to other programmers.

The autocall macro facility stores the source code for SAS macros (the %MACRO to %MEND definition) in external files that, taken together, form an autocall library. The library is an aggregate storage location such as a directory that contains files (or members). The only real constraint is that the macro definition and the file containing that definition must have the same name. Generally, an autocall library contains individual files, or members, each of which contains one macro definition. Actually, this is not a constraint so much as a feature. Under Windows, if you ask me for the definition of the %ABC macro, I know that the %MACRO to %MEND will be found in a file named ABC.SAS, and I know which of a limited number of locations it will reside in.

On UNIX platforms the name of the file storing the macro definition must be named with lowercased letters. All caps or mixed case will not work with the autocall facility on UNIX.

## **MAUTOSOURCE**

This option controls the availability of the autocall facility. When in effect, autocall libraries (specified with SASAUTOS=) are included in the search for the macro definition. Usually, this option is turned on by default, and unless you have a *very* specific reason to have it off, you generally want this option to be left on.

## **MRECALL**

When a macro is called and the macro or the macro library is not found, the autocall library might not be searched again on subsequent macro calls. The MRECALL option controls whether the autocall libraries will be searched again when a macro is not found. Typically, this option is turned off (the default) and is really needed only when you are using multiple shared autocall libraries and one or more libraries might be occasionally unavailable. During macro development a library can become unavailable when an attempt to compile a macro is unsuccessful or when the library location is incorrectly specified. The default value NOMRECALL prevents a second attempt. To the user the autocall library will appear to be broken. Restarting SAS clears the way for a second attempt; however, it might be easier to just set this option to MRECALL during macro development.

## **SASAUTOS=**

This option is used to point to locations for collections of macros that will be searched when a macro is called. The following OPTIONS statement defines a location for the autocall library in the Windows operating environment:

```
options sasautos="d:\caltasks\macros";
```

You can specify more than one macro library in the SASAUTOS= option, and you can use *fileref*s (be sure to use FILENAME statements and *not* LIBNAME statements!) instead of the actual specification of the library location. The following example specifies three macro libraries:

```
filename grp5mac 'c:\group5\macros';
filename prj5Amac 'c:\group5\prjA\macros';
options mautosource sasautos=(prj5amac, grp5mac, sasautos);
```

In addition to any autocall libraries that you might create, a number of macros have been included in the autocall library supplied with your SAS software. Some of these macros have been described in Section 10.6. Be sure to include the automatic *fileref* SASAUTOS in your list of autocall libraries. If you do not do so, you will be unable to access any of the autocall macros supplied by SAS.

**MORE INFORMATION:** Section 10.5 goes into detail on various aspects of building and using macro libraries.

**SEE ALSO:** Further discussion of the autocall facility can be found in Tindall and O'Connor (1991), O'Connor (1992), Carey and Carey (1996, p.142), Carpenter and Smith (2001), and Carpenter (2001).

The management of large macro libraries can be problematic, and Bryant (1997) includes a macro that will identify and list the macros in a system. Peszek and Troxell (2000) present a macro that generates an HTML management solution that is built with the use of JAVASCRIPT.

Chapter 9, "Storing and Reusing Macros," in *SAS 9.4 Macro Language Reference, Fourth Edition* provides additional information about autocall macros.

Jennifer Price (1998) gives some ideas on setting up an autocall library.

Peter Crawford (2016) includes a DATA step that will list macros in a macro library.

## 3.4 Testing Your Knowledge with Chapter Exercises

Program 3.4 and the exercises below provide you with an opportunity to test your knowledge of the content in this chapter.

### Program 3.4: Exercise Program

```
*****;
**** The class data set, CLINICS, contains 80      ****;
**** observations and 20 variables. The following ****;
**** program will be used to complete the exercises****;
**** in this chapter.                                ****;
*****;

title1 'Counts of Gender vs. Years of Education';
proc tabulate data=macro3.clinics;
  class sex edu;
  table sex=' ',edu*n=' '/box=sex;
  run;

title1 'Statistics on Heights for Each Gender';
proc univariate data=macro3.clinics;
  class sex;
  var ht;
  run;
```

1. Convert the code in Program 3.4 into a macro and execute it. You might want to use the code from Exercise 1 in Section 2.8 as a starting point. Be sure to examine the SAS Log.
2. In the macro program that you wrote in Exercise 1, use the options MLOGIC, MPRINT, and SYMBOLGEN separately and in combination to see how the SAS Log changes.
3. (True or False) To invoke a macro, you place a percent sign (%) before the macro name.
4. Create and use an autocall library.
  - a. Place the macro that you created in Exercise 1 into an autocall library.
  - b. Make sure that the compiled version of this macro is not already in the WORK.SASMACR catalog. If it is in the catalog, you can delete it using %SYSMACDELETE (see Section 8.2.5) or you can stop and restart SAS.
  - c. Specify the SASAUTOS= system option, and turn on MAUTOSOURCE.
  - d. Execute the macro without executing the %MACRO to %MEND statements.
  - e. Verify that the compiled macro is now in the SASMACR catalog.

# **Chapter 4: Using Macro Parameters**

<b>4.1 Introducing Macro Parameters .....</b>	<b>45</b>
<b>4.2 Using Positional Parameters .....</b>	<b>46</b>
4.2.1 Defining the Macro's Parameters.....	46
4.2.2 Passing Parameter Values into the Macro .....	46
<b>4.3 Using Keyword Parameters .....</b>	<b>48</b>
4.3.1 Defining the Parameters and Their Default Values .....	48
4.3.2 Passing Parameter Values When Calling the Macro .....	48
4.3.3 Documenting Your Macro .....	49
<b>4.4 Choosing between Keyword and Positional Parameters .....</b>	<b>50</b>
4.4.1 Selecting Parameter Types.....	50
4.4.2 Using Keyword and Positional Parameters Together .....	50
4.4.3 Naming Keyword Parameters without the Equal Sign .....	51
<b>4.5 Testing Your Knowledge with Chapter Exercises .....</b>	<b>51</b>

Macros are made more powerful and flexible when information can be transferred to them through the macro call. This chapter expands on the utility of macro variables by detailing their use as parameters that can be passed into and between macros. The chapter introduces two types of macro parameters, and their relative merits are discussed.

For the examples in this chapter and indeed for all of the code examples throughout the book, if you want to execute these sample programs, then be sure to follow the setup instructions. Remember that all of the data sets and programs are available for download, so you do not need to retype either the code or the data. For instructions on accessing and setting up the programs and data, see the “Example Code and Data” section within this edition’s “About This Book” front matter.

**SEE ALSO:** There is a very good overview of beginning macro issues in Whitlock (1999a).

---

## **4.1 Introducing Macro Parameters**

In Chapter 2, “Defining and Using Macro Variables,” macro variables were introduced and defined through the use of the %LET macro statement. Unfortunately, %LET becomes cumbersome and often limiting when it is used to make the values of macro variables available to a macro. If you use %LET in the global space, the macro variables will be available to all macros in the session; however, you will be restricted to using the specified names in all your macros. If you use the %LET within the macro definition, you will need to edit the macro each time you want to change a variable’s value. Macro parameters are used to overcome these limitations.

Macro parameters enable you to define macro variables without using the %LET statement. Macro parameters are used to pass values or text strings into a macro, where they are assigned to a macro variable on that macro’s local symbol table. Because these parameters are given names in the %MACRO statement, the names do not have to correspond to similar values on the global symbol table. The assignment of values to the parameters is made when the macro is called, not when the macro is coded. The following sections describe this process.

There are two types of parameters: *positional* (Section 4.2) and *keyword*, which is also known as *named* (Section 4.3).

## 4.2 Using Positional Parameters

*Positional parameters* derive their name from the fact that they are defined with the use of a specific position on the %MACRO statement. When the macro is called, the value is passed by means of that same corresponding position in the macro call as it has in the macro definition (%MACRO statement).

Positional parameters have only one *slight* advantage over named parameters—a bit less typing. Except for specific uses, most macro programmers agree that named parameters generally should be preferred to positional parameters.

### 4.2.1 Defining the Macro's Parameters

You define positional parameters by listing the macro variable names that are to receive the parameter values in the %MACRO statement. When parameters are present, the macro name is followed by a comma-separated list of macro variables that are enclosed in a pair of parentheses.

The following version of the %LOOK macro uses the %LET statement to establish two global macro variables (&DSN and &OBS):

```
%let dsn = clinics;
%let obs = 10;
%macro look;
  title "data set &dsn";
  proc contents data=&dsn;
    run;
  title2 "first &obs observations";
  proc print data=&dsn (obs=&obs);
    run;
%mend look;
```

We can easily convert this macro so that it uses positional parameters rather than relying on the %LET statement assignments. The resulting version of %LOOK in Program 4.2.1 has two positional parameters, and it is much more flexible than the version that uses global macro variables defined by %LET statements.

#### Program 4.2.1: Defining Positional Parameters

```
%macro look(dsn,obs);
  title "data set &dsn";
  proc contents data=&dsn;
    run;
  title2 "first &obs observations";
  proc print data=&dsn (obs=&obs);
    run;
%mend look;
```

The only difference in how these two versions of %LOOK are coded is in the %MACRO statement. Macro parameters are always written to the local symbol table. In this version of %LOOK, the use of parameters enables you to create &DSN and &OBS as local macro variables. One of the major advantages of using the local symbol table whenever possible is that it minimizes the risk of macro variable collisions (see Program 5.4.2b and Section 7.5.1).

### 4.2.2 Passing Parameter Values into the Macro

When calling a macro with parameters, you follow the macro name with a comma-separated list of parameter values enclosed within parentheses.

Because the parameters are positional, the first value in the macro call is assigned to the macro variable that is listed first in the macro statement's parameter list. When you have multiple parameters, you need to use

commas to separate their values. If you want to pass a value that contains a comma, you will need to use one of the quoting functions that are described in Section 7.1.

The macro call for the macro %LOOK in Program 4.2.1 could be as follows:

```
%look(clinics,10)
```

You do not have to give all parameters a value. Alternative invocations of the %LOOK macro might include the following; however, these macro calls do not necessarily result in usable code:

```
%look()
%look(clinics)
%look(,10)
```

Macro variables that are not assigned a value will resolve to a null string. Thus, the macro call %LOOK(,10) causes the parameter &DSN to resolve to a null value:

```
title "data set ";
proc contents data=;
run;
title2 "first 10 observations";
proc print data= (obs=10);
run;
```

The resolved code contains syntax errors, so it will not run. Be careful to construct code that will resolve to what you expect, and, when possible, anticipate and code around problems like this one.

The following macro %SORTIT, sorts a data set with one to three BY variables. The macro would not be needed or used if there were not at least one BY variable—indeed, syntax errors would be generated if that were the case.

#### **Program 4.2.2a: Defining the Macro %SORTIT**

```
%macro sortit(dsn,by1,by2,by3);
  proc sort data=&dsn;
    by &by1 &by2 &by3;
  run;
%mend sortit;
```

The macro call %SORTIT(CLINICS,LNAME,FNAME) leaves the third BY variable parameter null in the resolved code:

```
proc sort data= CLINICS;
  by LNAME FNAME;
run;
```

Because undefined parameters result in a null string, &BY3 is dropped from the resolved code. This technique enables you to create generalized code that will be syntactically correct at execution time. The macro %SORTIT will work correctly for one, two, or three BY variables. It will, of course, generate errors if no BY variables are passed into it.

The definition of %SORTIT in this code could be made more flexible by creating a single macro variable to hold all of the BY variables. In the Program 4.2.2b version of %SORTIT, the parameter &BYLIST replaces &BY1, &BY2, and &BY3:

#### **Program 4.2.2b: Passing a List of Values**

```
%macro sortit(dsn,bylist);
  proc sort data=&dsn;
    by &bylist;
  run;
%mend sortit;
```

Now, a call to %SORTIT is no longer restricted to a maximum of three macro variables. The macro call of %SORTIT(CLINICS,CLINNO LNAME FNAME SSN) resolves to the following:

```
proc sort data= CLINICS;
  by CLINNO LNAME FNAME SSN;
  run;
```

Notice that there are *no* commas between the names of the BY variables in the macro call. These four variables become a text string that forms the single definition to the macro variable &BYLIST. In this definition of %SORTIT, the user is not limited to three BY variables.

## 4.3 Using Keyword Parameters

Parameters may be designated as *keyword* in the %MACRO statement. Unlike positional parameters, keyword parameters may be used in any order and may be assigned default values. Keyword parameters are especially useful when there are large numbers of parameters or when the names of the parameters themselves will help the user with the correct specification of the macro call.

As far as best practices are concerned, most macro programmers agree that keyword parameters should be preferred over positional parameters. The only advantage of the positional parameters is the need for slightly less typing.

### 4.3.1 Defining the Parameters and Their Default Values

You define keyword parameters by following the parameter name with an equal sign (=) on the %MACRO statement. Default values, when present, follow the equal sign. You can use keyword parameters to redefine the %LOOK macro in Program 4.2.1.

#### Program 4.3.1: Defining Keyword Parameters

```
%macro look(dsn=clinics,obs=);
  title "data set &dsn";
  proc contents data=&dsn;
    run;
  title2 "first &obs observations";
  proc print data=&dsn (obs=&obs);
    run;
%mend look;
```

In this version of %LOOK, the macro variable &DSN will have a default value of CLINICS, while &OBS does not have a default value. If a value is not passed to &OBS, &OBS will take on a null value, in much the same way as a positional parameter will when it is not provided a value.

### 4.3.2 Passing Parameter Values When Calling the Macro

When a macro is called, keyword parameters that are not assigned a value will resolve to their default value, which will be a null value if you do not specify a default. Without additional coding, in a call to a macro using positional parameters, the default parameter value must necessarily be a null value. In Section 4.2.2 we saw that the macro call %LOOK(,10) would result in syntax errors because the data set name (the first parameter) is not provided. When you use the macro %LOOK that is defined with keyword parameters in Program 4.3.1, the macro call %LOOK(OBS=10) will not generate syntax errors as the default name of the data set, CLINICS, will be supplied.

```
title "data set clinics";
proc contents data= clinics;
  run;
title2 "first 10 observations";
proc print data= clinics (obs=10);
  run;
```

Because the macro call %LOOK(OBS=10) did not include a definition for &DSN, the default value of CLINICS was used. The resulting code eliminates the syntax errors that were generated in the %LOOK example in Section 4.2.2. However, because &OBS does not receive a default value, syntax errors will still result if a value is not provided for &OBS.

Unlike this example, as a general rule, it is a good idea to provide appropriate default values for all your parameters so that the macro will work correctly regardless of which parameters you specify.

### 4.3.3 Documenting Your Macro

There are a couple of documentation and style techniques that you can apply that will make your macros more readable and easier to use. These are especially applicable when using keyword parameters with your macro. As you develop your own style of internal documentation, consider not only future programmers, but also the users of your macro.

Things to think about might include:

- When you have more than two or three parameters, consider lining up the parameters vertically (one per line).
- Use descriptive parameter names that will help the user remember what each parameter does.
- Supply default values whenever reasonable defaults are available.
- Copy the list of parameters in the %MACRO statement to build a comment that explains each parameter (include a list of acceptable values when appropriate).

The macro %SCHOEN2 in Program 4.3.3 contains 10 keyword parameters. A full list of the parameters, their default values, and comments for each parameter is included as a part of the macro.

#### Program 4.3.3: Documenting Your Macro

```
%macro schoen2(data      = ,
               event     = status,
               outsch   = schr,
               outbt    = schbt,
               vref     = yes,
               points   = yes,
               df       = 4,
               alpha    = .05,
               rug      = no ,
               no       = no );
*****
* Macro parameters with default values.
data   = ,           name of the analysis data set.
event  = status,     event variable for survival status at exit,
                   1=event,
                   0=censored, currently named STATUS
outsch = schr,      name of output data set containing Schoenfeld
                   residuals.
outbt  = schbt,     name of output data set containing the scaled
                   Schoenfeld residuals(Bt).
vref   = yes,        indicator to control plotting of a vertical
                   reference line at y=0. Values are yes(default)
                   and no.
points = yes,        Indicates whether to plot the actual data points.
Default is yes.
df     = 4,          degrees of freedom for smoothing process.
                   Possible (integral)values are 3 to 7.
Default is 4.
alpha  = .05,        Confidence coefficient for plotting standard
                   error bars. Default is .05.
rug    = no,         indicator to control plotting of rug of x values.
no     = no,         Turn on and off macro debugging
                   no     debugging is off
```

```

blank debugging is on
*****;
OPTIONS &no.symbolgen &no.mprint &no.mlogic;
%put *** Entering SCHOEN2 macro ***;

/*...code not shown...*/

%put *** Leaving SCHOEN2 macro ***;
%mend schoen2;

```

If you need to document your macro for your users, this section of comments is easily copied into a word processing document.

## 4.4 Choosing between Keyword and Positional Parameters

When you are writing macros you will need to decide whether to use positional or keyword parameters. Some macro programmers always use one style or the other, while others make the determination on the basis of the program itself. Although there is no right or wrong way of selecting the type of parameters, there are some things that you might want consider.

### 4.4.1 Selecting Parameter Types

Macros with keyword parameters tend to be easier to document and to use, although there is a bit more typing. In most of my more recent macros, I have tended to use keyword parameters. Only rarely do I mix positional and keyword parameters.

Generally, I decide which type of parameters to use in accordance with some combination of answers to the following questions:

- **Who will be using the macro?** If the macro is going to be used by anyone other than the initial programmer, then keyword parameters have the advantages of ease of use and flexibility. You can build in parameters with default values that are for your use only (perhaps for debugging). This way, the user of the macro needs to be aware of only the parameters that he or she will need to use.
- **How many parameters are being passed?** If the macro only has two or three parameters, then it should be fairly easy for the user to get all of the parameters correctly specified and in the correct order. More than three, and the chances of mistakes on the part of the macro user are magnified. When a macro is only going to be called by another macro, I tend to ease up on this requirement a bit. Still, I tend to prefer keyword parameters.
- **Do any of the parameters need to have defaults?** If default values are required, then it is much easier to supply them with the use of keyword parameters than it is to infer them by using programming statements such as %IF-%THEN / %ELSE.
- **Are you creating a macro function?** User-defined macro functions (see Section 7.5) tend to use positional parameters so that they can mimic the syntax structure of other functions.

**MORE INFORMATION:** Section 14.3.2 discusses other issues related to macro programming style.

### 4.4.2 Using Keyword and Positional Parameters Together

Although you may specify both keyword and positional parameters in the same %MACRO statement, the list of positional parameters must precede the list of keyword parameters. Most macro programmers agree that this is generally a poor programming practice. Decide which you are going to use (as I may have mentioned, my bias is to use keyword parameters), and use only that type of parameter in a given macro.

In Program 4.4.2a the macro %LOOK has one positional and one keyword parameter, and the keyword parameter, &OBS, has the default value of 10.

**Program 4.4.2a: Using Positional and Keyword Parameters Together**

```
%macro look(dsn,obs=10);
  title "data set &dsn";
  proc contents data=&dsn;
    run;
  title2 "first &obs observations";
  proc print data=&dsn (obs=&obs);
    run;
%mend look;
```

The macro call %LOOK(CLINICS) would have the same result as %LOOK(CLINICS,OBS=10) and both would generate the same resolved code:

```
title "data set CLINICS ";
proc contents data= CLINICS;
  run;
title2 "first 10 observations";
proc print data= CLINICS (obs=10);
  run;
```

**4.4.3 Naming Keyword Parameters without the Equal Sign**

In the current versions of SAS, it is now possible to define a macro by using positional parameters and to then call the macro by using keyword parameters. This is not a recommended approach; however, it does allow you to take advantage of *some* of the benefits of keyword parameters.

In Program 4.4.3 the macro %TEST is defined with two positional parameters; however, the macro has been called as if the positional parameters were named parameters.

**Program 4.4.3: Using Keyword Parameters in the Macro Call**

```
%macro test(start,stop);
  %put &=start;
  %put &=stop;
%mend test;
%test(start=aa, stop=bb)
%test(stop=bb, start=aa)
```

The SAS Log shows that the parameter values were correctly assigned even when the parameter order was changed:

```
626 %test(start=aa, stop=bb)
start=aa
stop=bb
627 %test(stop=bb, start=aa)
start=aa
stop=bb
```

**4.5 Testing Your Knowledge with Chapter Exercises**

Test your knowledge of Chapter 4 by answering the following questions.

1. What are the two types of parameters that are available in macros?
2. (True or False) You can specify keyword parameters in any order.
3. Convert Program 4.5.3 into a macro so that it uses at least two parameters (or adapt the macro that you wrote in Chapter 3 Exercises 1 and 2). Do not use a %LET statement. Execute the macro by passing parameters into the macro.
  - a. Use positional parameters.
  - b. Use keyword parameters.

**Program 4.5.3: Program for Exercise 3**

```
*****  
**** The class data set, CLINICS, contains 80      ****;  
**** observations and 20 variables. The following    ****;  
**** program will be used to complete Exercise 3     ****;  
**** in this chapter.                            ****;  
*****  
  
title1 'Counts of Gender vs. Years of Education';  
proc tabulate data=macro3.clinics;  
  class sex edu;  
  table sex=' ',edu*n=' '/box=sex;  
  run;  
  
title1 'Statistics on Heights for Each Gender';  
proc univariate data=macro3.clinics;  
  class sex;  
  var ht;  
  run;
```

4. How many positional parameters are found in the following statement? Is the semicolon needed?

```
%macro tool(dsin,dsout,stop=500);
```

5. Why must keyword parameters follow positional parameters?  
6. What is wrong with the macro code shown in Program 4.5.6?

**Program 4.5.6: Required Correction of Syntax Errors**

```
macro mycopy;  
  proc copy in=work  out=master;  
    select patients;  
  run;  
%mend copy;
```

## **Part 2: Using Macros**

<b>Chapter 5: Controlling Programs with Macros.....</b>	<b>55</b>
<b>Chapter 6: Interfacing with Data Set Values.....</b>	<b>89</b>
<b>Chapter 7: Using Macro Functions .....</b>	<b>131</b>
<b>Chapter 8: Discovering Even More Macro Language Elements .....</b>	<b>195</b>
<b>Chapter 9: Exploring Some Less Common Intermediate Topics .....</b>	<b>231</b>
<b>Chapter 10: Building and Using Macro Libraries .....</b>	<b>253</b>

# Chapter 5: Controlling Programs with Macros

<b>5.1 Macros That Invoke Macros .....</b>	<b>55</b>
5.1.1 Passing Parameters between Macros .....	56
5.1.2 Passing Parameters When Macros Call Macros.....	57
5.1.3 Passing Macro Parameters through Macro Calls—An Illustrated Example .....	58
5.1.4 Controlling Macro Calls.....	62
5.1.5 Nesting Macro Definitions.....	63
<b>5.2 Conditional Execution Using %IF-%THEN/%ELSE Statements .....</b>	<b>64</b>
5.2.1 Executing Macro Statements .....	65
5.2.2 Building SAS Code Dynamically .....	66
5.2.3 Using the IN Comparison Operator.....	69
<b>5.3 Iterative Execution of Macro Statements .....</b>	<b>70</b>
5.3.1 %DO Block .....	70
5.3.2 Iterative %DO Loops .....	73
5.3.3 %DO %UNTIL Loops .....	76
5.3.4 %DO %WHILE Loops .....	77
<b>5.4 Additional Macro Program Statements.....</b>	<b>78</b>
5.4.1 Macro Comments .....	79
5.4.2 %GLOBAL and %LOCAL .....	81
5.4.3 %SYSEXEC .....	84
5.4.4 Termination of Macro Execution with %ABORT .....	84
5.4.5 Normal Termination of Macro Execution with %RETURN .....	85
<b>5.5 Testing Your Knowledge with Chapter Exercises .....</b>	<b>86</b>

The macro language is especially good at controlling the flow and execution order of your SAS programs. It includes a number of statements and related abilities for conditional execution, logical branches, and the building of code sequences by means of macros that call other macros. This is where we begin to explore some of the real power of the macro language.

For the examples in this chapter and indeed for all of the code examples throughout the book, if you want to execute these sample programs, then be sure to follow the setup instructions. Remember that all of the data sets and programs are available for download, so you do not need to retype either the code or the data. For instructions on accessing and setting up the programs and data, see the “Example Code and Data” section within this edition’s “About This Book” front matter.

---

## 5.1 Macros That Invoke Macros

It is not unusual for macros to call other macros and in the process to pass values to the macro that is being called. Often values that are defined in one macro will be used to affect the flow of logic and execution within a different macro. When a macro calls another macro, the call is said to be *nested* within the macro. It is often advantageous to nest macro calls (however, it is only very rarely appropriate to nest macro definitions).

Understanding the process of passing parameter values from one macro to another is difficult for many macro programmers who are new to the concept. Some simply avoid the issue of passing values between macros by using the global symbol table. However, because of my bias against the *per se* use of the global

symbol table (see Section 5.4.2 for the genesis of my bias), I feel that it is far preferable to make use of the local symbol tables and to pass values directly into the macros as parameters.

**MORE INFORMATION:** Sections 5.1.3 and 14.3.3 discuss in more detail the nesting of macro definitions.

### 5.1.1 Passing Parameters between Macros

Because passing parameters between macros is a difficult topic for many macro programmers, this section will offer a number of examples, with increasing complexity. Although it might be initially difficult, it is very important that you understand the concepts presented in this section.

In earlier examples, we have already seen how a value can be passed into a macro as a parameter. In the following macro call, the value of the data set name is passed directly into the %LOOK macro.

```
%look(dsn=clinics, obs=5)
```

If the name of the data set had already been stored in a macro variable, the macro call could have been rewritten to take advantage of that stored value.

```
%let dset = clinics;
%look(dsn=&dset, obs=5)
```

Before the macro %LOOK can be called, any macro references inside of the parentheses must first be resolved. In this case &DSET is resolved to *clinics*, and the macro call again becomes %look(dsn=clinics, obs=5). The important thing to notice is that a macro variable reference can be used inside of the parentheses on a macro call and that the reference will be resolved *before* the macro is actually called.

A similar concept is used when parameter values are passed between macros that call other macros. The difference is often that, instead of the macro variable's having been defined by a %LET, it is very often a parameter value that has been passed into the macro. This is the case in the next example, where the %LOCATE macro is called from within the %PRECOMP macro.

When developing a series of nested macros (macros that call macros that call macros, and so forth), it can be very tiresome to edit individual macros during debugging. One approach is to use comments so that you can turn on (or off) the system options that control the SAS Log (see Section 3.3.2). A more flexible version of the OPTIONS statement is shown in Program 5.1.1.

#### Program 5.1.1: Passing Parameter Values between Macro Calls

```
%macro precomp(subject=,no=no);
options &no.mprint &no.symbolgen &no.mlogic; ①

...Code not shown...

* Create the location data set;
%locate(subject=&subject,no=&no) ②

...Code not shown...

%mend precomp;

%macro locate(subject=, no=no);
options &no.mprint &no.symbolgen &no.mlogic; ③

...Code not shown...

%mend locate;
```

When the macro %PRECOMP is called without specifying a value for the &NO parameter, as in %PRECOMP (subject=1234), the &NO parameter will take on its default value of NO.

- ❶ Before the OPTION statement can be executed, the macro references to &NO must be resolved, in this case to its default value NO. The OPTION statement becomes the following:

```
options nomprint nosymbolgen nomlogic;
```

- ❷ Before the macro %LOCATE can be called, the macro references for both &SUBJECT and &NO must be resolved. The macro call becomes the following:

```
%locate(subject=1234, no=no)
```

- ❸ While the %LOCATE macro is executing, the parameter &NO on the OPTIONS statement is resolved to NO, and the OPTION statement becomes the following:

```
options nomprint nosymbolgen nomlogic;
```

If the %PRECOMP macro had instead been called with the specification of NO=, a null value would have been passed into the macro. Here the macro is specified with a null value for the NO parameter %PRECOMP (subject=1234, no=). The &NO parameter will take on a null value.

- ❶ Before the OPTION statement can be executed, the macro references to &NO must be resolved. The debugging options are turned on as the OPTION statement becomes the following:

```
options mprint symbolgen mlogic;
```

- ❷ Before the macro %LOCATE can be called, the macro references for &SUBJECT and &NO must be resolved. The macro call becomes the following:

```
%locate(subject=1234, no=)
```

- ❸ While the %LOCATE macro is executing, the OPTION statement becomes the following:

```
options mprint symbolgen mlogic;
```

If all the macros within a series of interrelated macros used similar structures for the OPTIONS statement, then you would be able to turn on and off debugging at any given point in your system and be assured that each called macro would have set the same debugging options.

**SEE ALSO:** Heaton (2001) shows a more sophisticated method of turning on and off debugging in a macro.

### 5.1.2 Passing Parameters When Macros Call Macros

It is not at all unusual to have a utility macro that calls a series of other macros. In Sections 4.2 and 4.3 the macros %LOOK and %SORTIT were introduced. In addition to other parameters, these two macros have one parameter in common (the name of the data set). In Program 5.1.2 we build a utility macro (%DOBOTH) to call these other two macros.

**Program 5.1.2: Passing Parameters When Macros Call Macros**

```
%macro doboth;
  %sortit(dset=clinics, bylist=lname fname)
  %look(obs=10)
%mend doboth;

%macro look(dsn=clinics,obs=);
  title1 "data set &dsn";
  proc contents data=&dsn;
    run;
  title2 "first &obs observations";
  proc print data=&dsn (obs=&obs);
    run;
%mend look;

%macro sortit(dset=biomass,bylist=);
  proc sort data=&dset;
    by &bylist;
    run;
%mend sortit;
```

Notice that the macro %DOBOTH is defined before the macros that it calls. The order in which the macros are defined does not matter as long as each macro is defined before it is called. The order of macro definition (and compilation) does not matter, because the calls to %SORTIT and %LOOK in the %DOBOTH macro are not executed until %DOBOTH itself is executed, and at that time all three macros must have been defined.

A call to the %DOBOTH macros will in turn call the macros %SORTIT and %LOOK and these will be resolved to executable code. You will notice that since %SORTIT is called first, the code it generates is also executed first:

```
proc sort data=clinics;
  by lname fname;
  run;
title1 "data set clinics";
proc contents data=clinics;
  run;
title2 "first 10 observations";
proc print data=clinics (obs=10);
  run;
```

In Program 5.1.2 the macro %DOBOTH has no parameters. Therefore, it would be necessary to change or edit the definition of %DOBOTH in order to alter how it calls %SORTIT and %LOOK. %DOBOTH would be much more flexible if the parameters for the macros %SORTIT and %LOOK could be passed directly through the call to %DOBOTH.

---

### **5.1.3 Passing Macro Parameters through Macro Calls—An Illustrated Example**

Program 5.1.3 contains a rewritten macro %DOBOTH (see Program 5.1.2). This version of the macro is written to receive the parameters that are to be used by %SORTIT and %LOOK. Notice that the parameters for the calls to %SORTIT and %LOOK are all %DOBOTH macro variables. The values of these macro variables are specified in the call to %DOBOTH, and their resolved values are passed on to %SORTIT and %LOOK.

**Program 5.1.3: Passing Parameter Values between Macros**

```
%macro doboth(indata=,vlist=,cnt=10);
    %sortit(dset=&indata, bylist=&vlist) ①
    %look(dsn=&indata,obs=&cnt) ②
%mend doboth;

%macro look(dsn=clinics,obs=);
    title1 "data set &dsn";
    proc contents data=&dsn;
        run;
    title2 "first &obs observations";
    proc print data=&dsn (obs=&obs);
        run;
%mend look;

%macro sortit(dset=biomass,bylist=);
    proc sort data=&dset; ③
        by &bylist; ④
        run; ⑤
%mend sortit;
```

Notice that the parameter names in the %DOBOTH call need not be the same names as those that are used in %SORTIT and %LOOK.

```
%doboth(indata=clinics,vlist=lname fname) ⑥
```

- ⑥ The macro call to %DOBOTH generates macro calls for %SORTIT and %LOOK.

```
%sortit(dset=clinics, bylist=lname fname) ①
%look(dsn=clinics,obs=10) ②
```

The call to the %DOBOTH macro generates calls to the %SORTIT and %LOOK macros.

- ① The parameter values are resolved before the %SORTIT macro can be called.
- ② It is important to note that it is the resolved values that are passed into the %LOOK macro, not the unresolved values.

In accordance with the values passed into the macros, the macro calls to %SORTIT, and %LOOK generate the PROC steps that are to be executed.

```
proc sort data=clinics;③
    by lname fname; ④
    run; ⑤
title1 "data set clinics";
proc contents data=clinics;
    run;
title2 "first 10 observations";
proc print data=clinics (obs=10);
    run;
```

The PROC SORT step generated by the %SORTIT contains the resolved values of the macro variables &DSET and &BYLIST.

- ③ The macro variable &DSET has been resolved to CLINICS.
- ④ The list of BY variables comes from the macro variable &BYLIST.

Most important, remember that the macro variables in %DOBOTH (for example, &INDATA and &VLIST), are resolved before the calls to %SORTIT and %LOOK are made. Consequently, it does not

matter that the names of the macro variables in %DOBOTH are different from the ones that are used in the other two macros.

It might help you to visualize the process of macros passing values to other macros if we look at the above calls in more detail.

Table 5.1.3 shows a portion of the sequence of operations that take place when the macro %DOBOTH is called. You can associate the event steps with the location in the code by using the callout numbers.

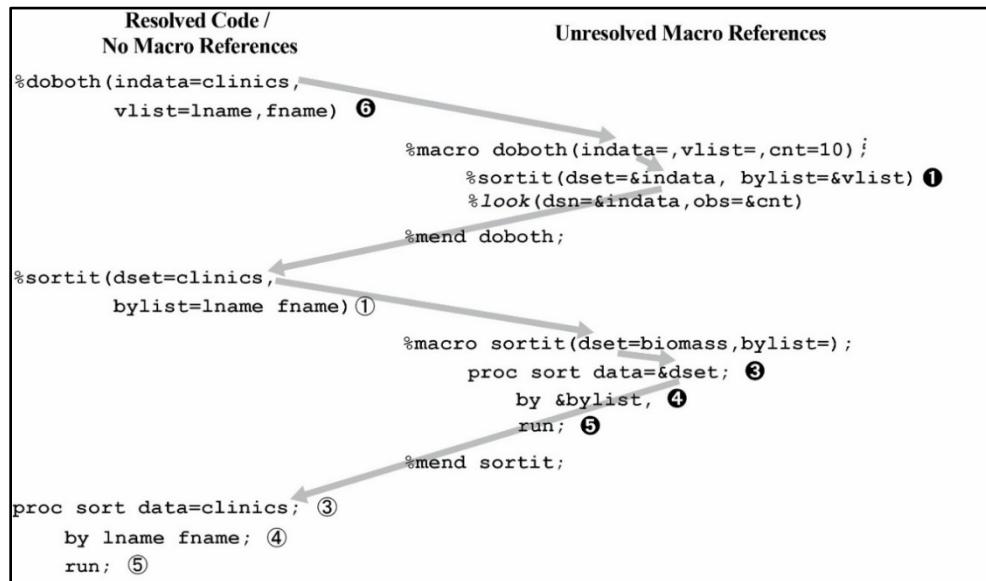
**Table 5.1.3: Step-by-Step Sequence of the Macro Parameter Resolution**

Step	Line	Code	What Happens
1	⑥	%DOBOTH (INDATA=CLINICS, VLIST=LNAME FNAME)	The macro %DOBOTH is called.
2			Macro variables within %DOBOTH receive values; for example, &INDATA receives the value of CLINICS.
3	①	%SORTIT (DSET=&INDATA, BYLIST =&VLIST)	The call for macro %SORTIT contains macro variable references. These references are resolved before the macro is called.
4	①	%SORTIT (DSET=CLINICS, BYLIST =LNAME FNAME)	The macro %SORTIT is called.
5			Macro variables within %SORTIT receive values; for example, &DSET receives the value of CLINICS.
6	③	PROC SORT DATA=&DSET;	The PROC statement contains macro variable references that must be resolved before the statement is submitted for execution.
7	③	PROC SORT DATA=CLINICS;	&DSET is replaced by CLINICS, and the statement is ready to be submitted.
8	④	BY &BYLIST;	The BY statement contains macro variable references that must be resolved before the statement is submitted for execution.
9	④	BY LNAME FNAME ;	&BY1, &BY2, and &BY3 are replaced with their respective values,

Step	Line	Code	What Happens
			and the statement is ready to submit (&BY3 is null).
10	⑤	RUN;	This statement contains no macro references and is submitted directly.
11			%SORTIT is complete and the next line in %DOBOTH is executed.
12	②	%look(dsn=&indata,obs=&cnt)	The call for macro %LOOK contains macro variable references. These must be resolved before the macro is called.
		[Remaining steps not shown]	The process of resolution and execution for %LOOK is similar to that for %SORTIT.

For the PROC SORT step generated by %SORTIT, the process of the resolution of the macro parameters can also be viewed through the steps in Figure 5.1.3.

**Figure 5.1.3: Resolution of the Data Set Name**



### 5.1.4 Controlling Macro Calls

Often it is useful to create a SAS program that consists simply of calls to macros. Doing so enables you to easily control, through macro calls, the execution of selected portions of the program.

The macro calls shown in Program 5.1.4a execute a series of macros that are either defined earlier in a portion of the code that is not shown here, or stored in a macro. Although you do not have the definitions of the macros available to look at here, it is instructive to look at how the macro calls are set up.

#### Program 5.1.4a: Program Control through a Series of Macro Calls

```
...Code not shown...
***** CREATE DATA AS REQUIRED*****;
%MACRO DOIT;
① %SETUP(NEW1,15,32)      * GATHER DATA FOR WORK.NEW1;
   %SETUP(NEW2,33,40)      * GATHER DATA FOR WORK.NEW2;
② %*SETUP(NEW3,41,72)      * GATHER DATA FOR WORK.NEW3;
③ %ALL(FINAL.COMBINE)    * COMBINE DATA FOR ANALYSIS;
%MEND DOIT;
%DOIT                      * CREATE THE ANALYSIS DATA SET;

***** ANALYZE COMBINED DATA ****;;
④ %CORREL                  * CORRELATION OF P/C VARS;
④ %ANOVA                   * ANALYSIS OF VARIANCE (P/C);
```

- ① The %SETUP macro creates a data set (for example, NEW1) based on some criteria in the second and third parameters.
- ② Because each macro call is followed by an asterisk-style comment, a given call to %SETUP can be commented out simply by commenting the macro call with a macro comment. In this example, an asterisk-style comment could also have been used; however, best practices recommend that asterisk-style comments not be used to comment out macro code within a macro. Section 14.3.6 shows how asterisk-style comments can cause problems within a macro definition.
- ③ The macro %ALL then combines all of the NEWi data sets (for example, NEW1, NEW2) that were created by the series of calls to %SETUP into the permanent data set FINAL.COMBINE. The data generation macros are wrapped in the %DOIT macro so that all the data generation macros can be easily commented at one time.
- ④ The two analysis macros (%CORREL and %ANOVA) then operate on FINAL.COMBINE to do the selected analyses. Either of these macro calls can also be commented out.

Using this arrangement of macro calls gives you a lot of flexibility. You can use comments to eliminate calls to macros that are not required (as is the third call to %SETUP ②). In addition, all control is located within ten lines of code rather than spread throughout an entire program. Section 3.2 contains more information on commenting out macro calls. This type of arrangement of macro calls can also serve as documentation of what was executed, how the parameters were specified when executed, and in what order the macros were called. Program 5.1.4b shows how Program 5.1.4a could be documented in this way.

#### Program 5.1.4b: Documenting Past Program Flow

```
***** Analysis performed November 23, 2014 ****;;
%MACRO DOIT;
  %SETUP(NEW1,15,32)      * GATHER DATA FOR WORK.NEW1;
  %SETUP(NEW2,33,40)      * GATHER DATA FOR WORK.NEW2;
  %ALL(FINAL.COMBINE)    * COMBINE DATA FOR ANALYSIS;

  ***** ANALYZE COMBINED DATA ****;;
  %ANOVA(type=A)          * ANALYSIS OF VARIANCE (P/C);

%MEND DOIT;
⑤ %*DOIT                  * CREATE THE ANALYSIS DATA SET;
```

- 5 The macro %DOIT definition documents what was done for the November 23, 2014 analysis; however, because the macro call has now been commented, the analysis will not inadvertently be re-executed. Notice that this macro call is in open code and either the asterisk or macro comments could be used.

A slight disadvantage of using the asterisk-style comment is that the comment itself is stored as part of the compiled macro. Comments defined with /\* ... \*/ and macro comments (see Section 5.4.1) are excluded from the compiled macro and can result in a small incremental savings in compilation time and storage size. Program 5.1.4c shows that the macro calls shown in Program 5.1.4a can be rewritten using /\* ... \*/ style comments.

#### Program 5.1.4c: Using Slash-Asterisk-Style Comments

```
%MACRO DOIT;
  %SETUP(NEW1,15,32)    /* GATHER DATA FOR WORK.NEW1*/
  %SETUP(NEW2,33,40)    /* GATHER DATA FOR WORK.NEW2*/
  /*%SETUP(NEW3,41,72)    /* GATHER DATA FOR WORK.NEW3*/
  %ALL(FINAL,COMBINE)   /* COMBINE DATA FOR ANALYSIS*/
%MEND DOIT;
%DOIT                  /* CREATE THE ANALYSIS DATA SET*/

***** ANALYZE COMBINED DATA ****;
%CORREL                /* CORRELATION OF P/C VARS */
%ANOVA                 /* ANALYSIS OF VARIANCE (P/C) */
```

## 5.1.5 Nesting Macro Definitions

Already in this chapter we have seen a series of examples that have macros calling other macros. Although it is very common to have macro calls within a macro, it is not at all common to be in a situation in which the macro definitions need to be nested—actually, I have never had this need, but I have unfortunately seen a number of nested macro definitions in my client’s code.

A nested macro definition occurs when the %MACRO through %MEND statements are contained within the definition of another macro. Almost always when you see this in a program, it is because the programmer does not truly understand how macro definitions are stored. It is only *very rarely* necessary or even appropriate to nest macro definitions.

In the pseudo code shown in Program 5.1.5a the definition of the macro %DEF is nested within the definition of the macro %ABC.

#### Program 5.1.5a: Nesting Macro Definitions

```
%macro abc;
  ...Code not shown...
  %macro def;
    ...Code not shown...
  %mend def;
  %def
  ...Code not shown...
%mend abc      ;
%abc
```

Every time the macro %ABC is executed, the macro %DEF will be recompiled. There is no need for this. By moving the definition of the embedded macro (%DEF) so that it is not contained within the %MACRO and %MEND of the macro %ABC, you will ensure that it is compiled only once. It does not matter which macro is defined first, so long as both are defined before %ABC is called.

**Program 5.1.5b: Nesting Macro Calls but Not Definitions**

```
%macro abc;
  ...Code not shown...
  %def
%mend abc      ;
%macro def;
  ...Code not shown...
%mend def;
%abc
```

Now that I have said to never nest macro definitions, one legitimate use might be if the macro %ABC defined some conditions that were necessary in order to create a customized version of %DEF. Here again, though, I would tend to pass these conditions into %DEF as parameters, if at all possible, and use macro logic to perform the customization.

**MORE INFORMATION:** Another coding technique, in which nested macro definitions *might* have some utility, is when you are using a macro to “comment” out a block of code. Section 3.1.2 discusses this technique in more detail. Additional discussion on nesting macro definitions can be found in Section 14.3.3.

**SEE ALSO:** The SAS supplied autocall macro %LOG4SAS (`(\Program Files\SASHome2\SASFoundation\9.4\core\sasmacro)`) uses this technique to compile a series of macros with a single macro call.

**5.2 Conditional Execution Using %IF-%THEN/%ELSE Statements**

IF-THEN/ELSE processing in the DATA step enables you to write programs that will act conditionally in accordance with values contained in variables on the Program Data Vector (PDV). The macro %IF-%THEN and %ELSE statements are analogous to the DATA step IF-THEN/ELSE statements. However, they are not constrained to the DATA step and do *not* operate on DATA step variables.

Macro %IF-%THEN and %ELSE statements may appear anywhere inside of a macro (they cannot be used in open code) and are used to conditionally

- Assign values to macro variables,
- Execute other macros, and
- Create SAS code and SAS statements.

**SYNTAX:**

```
%IF expression %THEN result-text;
%ELSE result-text;
```

Most of the same rules that apply to the DATA step IF-THEN/ELSE statements apply here as well. There are three primary differences that you must keep in mind when using macro language %IF-%THEN/%ELSE statements:

- They do *not* operate on the values of DATA step variables. Macro statements and macro references are resolved before *any* data are read. Macro expressions will *never* directly evaluate the value of a variable on the PDV.
- They control program flow and code construction, *not* the flow of the execution of a DATA step. Again, it is very important to remember that, whatever the macro statements do, they will be resolved and executed long before the SAS program is compiled and executed (see Figure 1.4d).
- Comparisons are made against literal text strings, and quotes are generally neither used nor desirable. For example, “X”= ‘X’ would be false.

Macro logical expressions can contain many of the same comparison operators that are used in other expressions. This includes Boolean operators such as AND and OR, as well as the logical negation operator NOT. As in the DATA step and elsewhere within SAS, a logical expression resolves to *true* (1) or *false* (0); however, in macro expressions what is true and false is not exactly the same as elsewhere. In macro expressions zero (0) is false, and one (1) is true; however, not all nonzero evaluations are true—some cause errors. There are a number of other differences in how logical expressions are evaluated:

- The IN operator has different syntax (see Section 5.2.3).
- Comparisons between two integers are evaluated as numeric (for example,  $20 > 9$  is true).
- Comparisons between one or more non-integer values are evaluated alphabetically (for example,  $20 > 9.1$  is false because 2 sorts before 9 alphabetically).
- To be evaluated as true or false, an expression must evaluate to an integer (for example, expressions that evaluate to either 1 or 5 would both be true, but 5.5 would cause an error).
- Missing values are not recognized in the macro language and therefore cannot be evaluated to false. An expression that evaluates to period (.) or to a null value will cause an error.
- The colon (:) comparison modifier cannot be used, because it becomes part of the comparison itself (for example, `Smith =: S` is false, although `:S =:S` would be true).

**MORE INFORMATION:** Section 1.4 discusses the phases of macro execution and highlights the relationship between macro language and DATA step execution. There is often initial confusion on the roles of the IF and %IF statements. Section 14.3.5 also addresses the differences.

**SEE ALSO:** Virgile (1997) uses the CALL EXECUTE routine to set up conditional macro logic outside of macros.

## 5.2.1 Executing Macro Statements

Like the IF statement's *action* in the DATA step, the *result-text* in the macro %IF takes place when the *expression* is true. The construction of the *expression* is similar to what it would be for the IF statement, and the same comparison and Boolean operators are available. However, the %IF's expression will *not*, as was hinted at earlier, be testing values of DATA step variables. This point has been repeated multiple times because it is a major point of confusion for many first-time users.

In this example, the %IF tests the value of the macro variable &CITY. Notice that the value *Albany* is not enclosed in quotes as it would be if CITY were a character variable on the PDV:

```
%if &city=Albany %then %let state>New York;
%else %let state>California;
```

In the DATA step, the quotes are parsing characters that help the parser distinguish between variable names and constant strings. Because macro variable names are always preceded by an ampersand (&), macro variables are easily distinguished from text (anything that does not have a macro trigger); therefore, quotes are not required. In this example, when the value *Albany* is stored in &CITY, the macro variable &STATE will be set to *New York*. For all other values of &CITY, the %ELSE causes the second %LET statement to be executed, and &STATE is set to *California*.

Unlike in the DATA step, where the programmer must be concerned about the length of a variable when assigning it values, in the macro language the storage is allocated dynamically as it is needed, so truncation is not an issue. In the previous example, although *New York* is coded first and is shorter than *California*, there will be no truncation issues.

In Program 5.1.1 the %SORTIT macro will fail with a syntax error if it is called without passing any BY variables into the macro. It would be nice if the %SORTIT macro were executed only if at least one BY variable was provided. In the revised %DOBOTH macro in Program 5.2.1, the macro %SORTIT is not executed unless it is determined that the macro parameter &VLIST is not null.

**Program 5.2.1: Conditional Execution of a Macro**

```
%macro doboth(indata=,vlist=,cnt=10);
  %if &vlist ne %then %sortit(dset=&indata, bylist=&vlist);
  %else %put SORTIT was not called;
  %look(dsn=&indata,obs=&cnt)
%mend doboth;
```

Notice that, since there is nothing on the right side of the equal sign, the expression (`&vlist ne`) used in this macro %IF statement would not work in a DATA step IF statement. Because macro variables can contain a null value (remember that this is different from a blank), and because quotes are unnecessary, this syntax is correctly interpreted. But it does look strange. To make it look less strange, some programmers might rewrite the %IF using double quotes as placeholders:

```
%if "&vlist" ne "" %then %sortit(dset=&indata, bylist=&vlist);
```

Using double quotes on both sides of the comparison operator is not recommended. Using them generally shows poor understanding of how the %IF comparison actually works. Also, there is a tendency to place a blank between the two double quotes on the right, but doing so will include the blank as part of the characters being compared, which would probably be undesirable. When quotes are included they become a part of the text that is to be compared; for this reason, you cannot use single quotes on the right side and double quotes on the left.

In each of the previous examples, the result-text is a macro statement or a macro call that is then immediately executed. When the result-text does not contain any macro variables or calls, the result-text is “left behind” and becomes part of the SAS code that is generated (“typed”) by the macro.

**MORE INFORMATION:** Special considerations should be taken when you are comparing numeric values. See Sections 5.2.3, 7.3.1 (%EVAL), and 7.3.3 (%SYSEVALF) for more details.

Refinements of the above code that compare a macro variable to a null value can be found in Section 7.1.2, Section 9.5, and in the answer to Exercise 4 in the Chapter 7, which is found in Appendix 1 (Section A1.7.4).

---

**5.2.2 Building SAS Code Dynamically**

Dynamic coding is the third level of learning of the macro language (see Section 1.2.3 and Chapter 11), and one of its hallmarks is the use of conditional macro statements to build Base SAS statements from the inside out. The example in this section introduces this concept in a very simplified way.

You can use the %IF to conditionally insert SAS code into a program. It enables you to create programs that are not fully defined until they are actually executed. In Program 5.2.2a, the data set that is to be named in the SET statement is not known until the parameter (&STATE), which is passed into %DASTEP, is evaluated.

**Program 5.2.2a: Building a SET Statement Dynamically**

```
%macro dastep(state=AZ);
data subhosp;
  set ①
  %if &state=CA %then cahosp; ②
  %else azhosp;
  ; ③
  where date>'19jun2014'd;
  run;
%mend dastep;
```

- ① The SET statement will start here.
- ② The shaded macro code will be executed before the SET statement is compiled. Depending on the value of &STATE, either CAHOSP or AZHOSP will be “left behind” as the data set name.

- ③ This semicolon will be used to close the SET statement.

If the macro call %dastep(state=CA) is submitted, the resolved SET statement will have no macro language elements and will read the data set CAHOSP.

```
data subhosp;
  set
    cahosp
  ;
  where date>'19jun2014'd;
run;
```

Note the use of semicolons in the statements in Program 5.2.2a. Initially it might seem that there are just too many semicolons for this code to possibly work. Certainly, there are three semicolons after the keyword SET. How can you as the programmer, let alone the SAS parser, keep track of which semicolon is used with which statement?

Remembering that macro language elements are executed first will help. Also remember that the macro parser will isolate macro language elements to form macro tokens and pass these tokens to the macro facility. Since each macro statement ends with a semicolon, these semicolons will be used to form the macro tokens (see the shaded code ②). So, although at first glance it seems that these semicolons will end the SET statement or at least confuse the DATA step compiler, instead they are used to form the macro token (%IF-%THEN/%ELSE). Because all macro references and macro-level statements will be resolved or executed before the DATA step is compiled, any semicolons that are associated with macro statements will no longer be present by the time the SET statement is compiled.

In Program 5.2.2a, the semicolon that is used with the SET statement is on its own line ③ (just after the %ELSE statement). It could also have been on the same line as the %ELSE. It was placed on its own line to make it easier for the programmer to read; SAS does not prefer one location over the other.

The section of code in Program 5.2.2b is one of several DATA steps in the macro %SENRATE. Macro %IF-%THEN/%ELSE logic is used to determine how the variable GENDER is to be defined relative to the incoming variable SEX. The DATA step will create one of a series of tables that are to be written to the library SENRATE. For table SENRATE.RTSEN5 (&NUM=5), the proper value for the variable GENDER already exists in the variable SEX (only a RENAME is required). For all other values of &NUM, the PUT function uses the value of the variable SEX to create the variable GENDER.

#### Program 5.2.2b: Semicolon Specification

```
%macro senrate(num=);
  ...Code not shown...
  data senrate.rtsen&num;
    set ratedata&num;
    * Create the variable GENDER
    * based on the variable SEX.
    *   Table 5 Rename SEX to GENDER;
    *   Table 3 & 7 use a format to define GENDER;
    %if &num=5 %then rename sex = gender; ④
    %else gender = put(sex, $sexdef.); ④
  ; ⑤
  ...Code not shown...
%mend senrate;
```

- ④ The semicolon in the %IF statement ends the %IF statement and not the assignment statement. The semicolon in the %ELSE statement ends the %ELSE and not the assignment statement. At this point neither of the assignment statements has its semicolon. Technically, until the %IF-%THEN/%ELSE executes, neither of the assignment statements has even been created. After the %IF-%THEN/%ELSE

executes, only one of the assignment statements will have become part of the DATA step, and it is this assignment statement that receives the semicolon ⑤.

- ⑤ This semicolon will be used to close the assignment statement that results from the execution of the %IF-%THEN/%ELSE ④.

For &NUM=3, the DATA step in Program 5.2.2b would generate GENDER, using the PUT function:

```
...Code not shown...

data senrate.rtsen3;
set ratedata3;
* Create the variable GENDER
* based on the variable SEX.
* Table 5 Rename SEX to GENDER;
* Table 3 & 7 use a format to define GENDER;
gender = put(sex, $sexdef.)
; ⑤

...Code not shown...
```

Using a “hanging” semicolon, as was done in the previous two examples, is not unusual, but there are alternate coding solutions. One is to use the %DO block, which is described in Section 5.3.1.

Misplacement of the semicolon can have some unintended consequences. In Program 5.2.2c the attempt has been made to close each of the potential base language statements with a semicolon rather than following the %ELSE statement as was done in Program 5.2.2b.

#### Program 5.2.2c: Semicolon Misplacement

```
%macro senrate(num=);
...Code not shown...

data senrate.rtsen&num;
set ratedata&num;
* Create the variable GENDER
* based on the variable SEX.
* Table 5 Rename SEX to GENDER;
* Table 3 & 7 use a format to define GENDER;
%if &num=5 %then rename sex = gender; ; ⑥
%else gender = put(sex, $sexdef.); ;

...Code not shown...

%mend senrate;
```

- ⑥ The use of two semicolons causes a macro language error, because although we want the first semicolon to close the assignment statement and the second semicolon to close the %IF (or %ELSE) this is not what happens. The scanner detects the first semicolon and uses it to close the %IF. Although the second semicolon is not part of a macro language statement, the scanner knows that when a %ELSE follows a %IF, the %ELSE will be the next statement, and the existence of the second semicolon precludes that possibility. Now when the scanner finds the %ELSE, it has no corresponding %IF (they have become disassociated by the second semicolon in the %IF) and the %ELSE causes an error.

**MORE INFORMATION:** The %DO block is often used for semicolon management (see Section 5.3.1 for another version of the above example). The discussion of macro %DOBOTH in Section 5.3.1 also includes comments on the management of the semicolon.

**SEE ALSO:** Leighton (1997) contains several examples of dynamic code building. The topics of semicolon management and, specifically, the use of the double semicolon are addressed by Yindra (1998).

### 5.2.3 Using the IN Comparison Operator

The IN comparison operator enables you to easily check to see if one value also appears within a list of alternate values. Effectively, this enables you to more easily check what would otherwise be coded as a series of OR conditions (Program 5.2.3a).

#### Program 5.2.3a: Using a List of OR Operators

```
%if &state=&v1 or &state=&v2 or &state=&v3 %then  
  %put &=state is one of the list |&v1 &v2 &v3|;
```

Although the IN operator in the macro language is similar to the DATA step's comparison operator of the same name, there are some differences that need to be noted. As with the DATA step, you can use it to search for an item within a list of items; however, the ability to use this operator is turned on by the MINOPERATOR system option, and by default this operator is not available (NOMINOPERATOR):

```
options minoperator;
```

If you attempt to use the IN operator without turning on this option, then it will not be seen as an acceptable comparison operator, and the SAS Log will display an error:

```
ERROR: A character operand was found in the %EVAL function or %IF condition  
where a numeric operand is required.
```

When using this operator, you can use either IN or the pound sign (#).

#### SYNTAX:

```
%IF item in target_list %THEN action;  
%IF item# target_list %THEN action;
```

During evaluation, the expression, which uses an implied %EVAL function (see Section 7.3.2), will return a 1 if *item* is found as one of the values in *target\_list*. Otherwise, it returns a 0. In the following %IF statement, the expression is true because the value *aa* is in the list:

```
%if aa in aa bb cc dd %then %do;
```

Notice that the *target list* is separated by spaces and that the IN operator immediately follows the *item* being searched for. By default, the *target\_list* must be space delimited. For this reason, the following comparison will return a 0 (*false*) because the comma-separated list will be seen as a single item (there are no spaces):

```
%if aa in aa,bb,cc,dd %then %do;
```

Having a space as the only possible list delimiter could be inconvenient; consequently, the system option MINDELIMITER can be used to change the default delimiter from a space to some other character. In the next example, the delimiter is set to a comma, so the expression in the %IF would now be evaluated to *true*:

```
options mindelimiter=',';  
%if aa in aa,bb,cc,dd %then %do;
```

In actual use the individual values in the list might be macro variables, or the list itself might be a macro variable that contains the list. When you create the list, you will need to be careful that you correctly specify the delimiter. In Program 5.2.3b a space-separated list is searched using the pound sign as the IN operator.

#### Program 5.2.3b: Testing the IN Operator

```
%macro test(state=, list=);
%if %upcase(&state) # %upcase(&list) %then ①
    %put TEST &=state is in &=list;
%mend test;

%test(state= ak, list= CA WA AK)②
```

- ① Because the comparison is case sensitive, the %UPCASE function has been used.
- ② The list is space-separated.

The IN operator requires an explicit %EVAL (see Section 7.3.1) when it is used outside of the %IF statement. In the %LET statement that follows, the macro variable &INLIST will take on the value of either 0 or 1.

```
%let inlist = %eval(&state in &list);
```

**MORE INFORMATION:** The construction of macro variables that contain lists is discussed in a number of sections within this book. In Programs 5.2.3a and 5.2.3b the postal codes for some states (Oregon, Indiana, and Nebraska) will cause these and related comparisons to fail. Solutions are discussed in Section 7.1.1.

## 5.3 Iterative Execution of Macro Statements

The %DO block, iterative %DO, %DO %UNTIL, and %DO %WHILE statements in the macro language are very similar to the corresponding statements that are used in the DATA step. Like the %IF statement these statements are not confined to the DATA step; however, you must use them inside of a macro.

All forms of the %DO statement must be matched with a %END statement.

### 5.3.1 %DO Block

The simplest form of the macro %DO statement is the %DO block, and it is analogous to the DATA step's DO block. Remember that the %DO statement is always paired with a %END.

#### SYNTAX:

```
%DO;
  . . . text . . .
%END;
```

The %DO block is most commonly needed when you want the result-text of a %IF-%THEN statement to contain multiple macro calls, multiple macro statements, or even multiple SAS statements.

You can also use the %DO block to help with semicolon management when you dynamically create code. In Section 5.2.2 the “hanging” semicolon problem, and the confusion that it causes, was discussed. In Program 5.3.1a the %DO block is used to enclose the semicolon that terminates the SET statement, thereby eliminating the “hanging” semicolon.

**Program 5.3.1a: Using %DO to Enclose the Semicolon**

```
%macro dastep(state=AZ);
data subhosp;
  set
    %if &state=CA %then %do;
      cahosp; ①
    %end;
    %else %do;
      azhosp; ①
    %end;
    where date>'19jun2014'd;
    run;
%mend dastep;
```

- ① This semicolon will terminate the SET statement, and the %DO-%END masks it from being seen as the terminator of the %IF statement.

Of course, if you are going to use this type of solution, which is no longer dynamically creating the SET statement, you might as well enclose the whole SET statement inside of the %DO block as is done in Program 5.3.1b.

**Program 5.3.1b: Enclosing the Entire SET Statement in the %DO Block**

```
%macro dastep(state=AZ);
data subhosp;
  %if &state=CA %then %do;
    set cahosp;
  %end;
  %else %do;
    set azhosp;
  %end;
  where date>'19jun2014'd;
  run;
%mend dastep;
```

In a similar manner, Program 5.3.1c shows that the %DO block can be used to enclose the assignment statements in the %SENRATE macro (Program 5.2.2b).

**Program 5.3.1c: Enclosing Complete Statements in the %DO Block**

```
%macro senrate(num=);
...Code not shown...
data senrate.rtsen&num;
set ratedata&num;
* Create the variable GENDER
* based on the variable SEX.
*   Table 5 Rename SEX to GENDER;
*   Table 3 & 7 use a format to define GENDER;
%if &num=5 %then %do;
  rename sex = gender;
%end;
%else %do;
  gender = put(sex, $sexdef.);
%end;
...Code not shown...
%mend senrate;
```

It is easier to follow the logic in this code than it was to follow that in Program 5.2.2a, because the %DO blocks contain complete SAS statements here.

Along the same lines, you can greatly simplify macro %DOBOTH in Section 5.2.1 by eliminating the calls to %SORTIT and %LOOK. The new macro becomes Program 5.3.1d:

#### Program 5.3.1d: Using %DO to Enclose Semicolons

```
%macro doboth(indata=,vlist=,cnt=10);
  %if &vlist ne %then %do; ②
    proc sort data=&indata;
      by &vlist;
      run;
  %end;
  %else %put PROC SORT was not called;

  title1 "data set &indata";
  proc contents data=&indata;
    run;
  proc print data=&indata
  %if &cnt>1 %then %do; ③
    (obs=&cnt);
    title2 "First &cnt Observations";
  %end;
  %else %do; ④
    (obs=max);
    title2;
  %end;
  run;
%mend doboth;
```

- ② The PROC SORT step will not be included in the final code when the &VLIST macro variable is null.
- ③ In the PROC PRINT, &CNT is checked before adding the OBS= option and a customized title.
- ④ The OBS=MAX is specified when the number of observations to be printed is not an integer greater than 1.

You can also use the %DO block to conditionally execute entire blocks of code. In Program 5.3.1e, blocks of SAS code are conditionally executed. The macro %TDATAPREP is used to create the temporary data set TDATA. The difficulty lies in the fact that there are two styles of input data that must be handled differently. The data type is specified in &STYLE, and this macro variable is used to determine the appropriate code that is to be inserted into the program:

#### Program 5.3.1e: Conditional Insertion of Blocks of Code

```
%macro Tdataprep(style=new,
                 dsn=,
                 temp=,
                 maxage=,
                 regvar=);
***** ⑤
* style  = new      Current data format
*        OLD      historical data form
* dsn    =          data set containing old style data
*          data type for new data
* temp   =          testing temperature used to subset old data
* maxage =          maximum product age for old data
* regvar =          Test regression variable, used with new data
*****;

%* test for type of data and set up accordingly.; ⑤
%if &style=OLD %then %do;
  * read the data for a specific test;
```

```

proc sort data=old.&dsn out=tdata;
  by varno age;
  where temp=&temp & age le &maxage;
  run;
%end;
%else %do;
  data tdata (keep=varno canno temp age depvar test package);
    set cps.dest (keep=base no &regvar.1-&regvar.6
                  temp test package);
    array can &regvar.1-&regvar.6;
    length age varno 8;
    where base="&dsn";
    ...Code not shown...
  run;

  proc sort data=tdata;
    by varno age temp;
    run;
%end;
%mend tdataprep;

```

- 5 Notice the use of macro comments rather than asterisk-style comments (see Section 5.4.1 for more on macro comments). Within a macro, macro comments and /\* . . . \*/ style comments are safer than asterisk comments. This is especially true when you are commenting out macro code. In this example, any of the three styles of comments would have worked.

### 5.3.2 Iterative %DO Loops

Unlike the %DO block, which is executed once, the iterative %DO forms a loop that allows a series of passes. Although the form of the iterative %DO resembles the DO statement used in the DATA step, there are differences between them:

- The %WHILE and %UNTIL specifications cannot be added to the increments of the iterative %DO.
- Only integers can be used for the *start*, *stop*, and *increment* values.
- Only one specification is allowed; you cannot form composite loop specifications.
- Text or character loop specifications are not allowed.
- The %TO is required.

#### SYNTAX:

```
%DO macro-variable = start %TO stop <%BY increment>;
  . . . text . . .
%END;
```

The iterative %DO defines and increments a macro variable. If the index variable does not already exist, it is created. In Program 5.3.2a, the macro variable &YEAR is initialized to &START and incremented by one ending with &STOP.

#### Program 5.3.2a: Using the Iterative %DO Loop

```
%macro allyr(start=04,stop=05);
  %do year = &start %to &stop; ①
    data temp;
      set yr&year;
      year = 2000 + &year; ②
    run;
```

```

proc datasets lib=work nolist;
    append base=allyear data=temp; ③
    quit;
%end;
%mend allyr;
%allyr(start=12, stop=14)

```

- ① The macro variable &YEAR is initialized as &START and incremented by 1 as the loop progresses to &STOP.
- ② Although the DATA step also has a variable YEAR, it will not be confused with &YEAR. The incoming data sets are named YR12, YR13, and so on.
- ③ The incoming data sets are read one at a time, and they are appended to the all-inclusive data set ALLYEAR.

The macro call %allyr(start=12, stop=14) causes the following code to be generated:

```

data temp;
    set yr12;
    year = 2000 + 12; ②
    run;
proc datasets lib=work nolist; ③
    append base=allyear data=temp;
    quit;

data temp;
    set yr13;
    year = 2000 + 13; ②
    run;
proc datasets lib=work nolist; ③
    append base=allyear data=temp;
    quit;

data temp;
    set yr14;
    year = 2000 + 14; ②
    run;
proc datasets lib=work nolist; ③
    append base=allyear data=temp;
    quit;

```

The macro language is great at rapidly generating code. In Program 5.3.2a the macro %ALLYR almost instantly generated a DATA step and a PROC step for each of the three years. Although the creation of these steps was very fast, the execution of these steps might not be so very fast. In the resulting program, each observation from each of the data sets has to be handled four times: twice in the DATA step and twice in the PROC step. Moreover, there is the additional overhead of the six steps themselves. As macro programmers, we need to be very aware of the code that we generate. The macro language can generate code very quickly, but that code may not necessarily be efficient code. It is our responsibility to make sure that the generated code is also good code. As a macro programmer, when you think of the product produced by your macro, think about the code generated by your macro. Your program may ultimately generate a report or a data set, but the final product of your macro is the code that it generates.

Recognizing the efficiency issues of generated code is important. And if efficiency is taken into consideration, then we can greatly simplify the generated code in Program 5.3.2a and improve the overall efficiency of the program by using dynamic programming techniques and thereby taking better advantage

of the %DO loop. Rather than having the macro create separate DATA and PROC steps for each year, we can build the code dynamically, and handle each observation only twice in a single step. Program 5.3.2b uses %DO loops to dynamically build the SET statement and the assignment statement that specifies the YEAR variable:

#### Program 5.3.2b: Dynamically Building Code with the Iterative %DO

```
%macro allyr2(start=10,stop=14);
  data allyear;
    set
      %do year = &start %to &stop;
        yr&year(in=in&year)
      %end;;
    year = 2000
    %do year = &start %to &stop;
      + (in&year*&year)
    %end;;
    run;
  %mend allyr2;
  %allyr2(start=11, stop=13)
```

The call to %allyr2 (start=11, stop=13) generates only one DATA step:

```
data allyear;
set yr11(in=in11)
  yr12(in=in12)
  yr13(in=in13);
year = 2000 +
  (in11*11) +
  (in12*12) +
  (in13*13);
run;
```

In this DATA step, the value of YEAR is assigned by taking advantage of the IN= data set option. The values of IN11, IN12, and IN13 will be either *true* or *false* (1 or 0), and only one of the three will be *true* at any given time.

In both Programs 5.2.2a and 5.3.2b the SET statement is built from the inside through the use of %DO blocks. In the case of Program 5.3.2b a %DO loop is used to build the names of the data sets that are to appear within the SET statement.

The %READNEW macro, which is shown in Program 5.3.2c, also reads a series of data sets and builds the SET statement dynamically using a %DO loop. However, in this example, the %DO loop is to be passed into the macro and is placed in the macro call rather than within the macro itself. In the call to %READNEW, the %DO loop is written as the value of the &DSIN parameter. Although this simplifies the code in the macro, it makes the macro call more complex. More important, it does not do what its authors of the macro intended, at least not in the way that they intended. Let's examine their code more closely.

#### Program 5.3.2c: Passing the Iterative %DO into a SET Statement

```
%macro readnew(dsin=,dsout=);
  data &dsout;
    set &dsin;

  ...Code not shown...

  %mend readnew;

*****;
%macro doit;
* use selected surveys*;
%readnew(dsout=new1,dsin=%do i=16 %to 18; dbbio.sur&i %end;)
%readnew(dsout=new2,dsin=%do i=42 %to 54; dbbio.sur&i %end;)
```

```
%readnew(dsout=new3,dsin=%do i=55 %to 72; dbbio.sur&i %end; )
%all
%mend doit;

%doit
```

The macro's author intended to pass the %DO loop into the macro %READNEW, where it would be used to build the SET statement. This is not, however, what happens. Because in the macro call the parameter &DSIN contains the macro %DO statement, the loop will be executed and resolved before the resulting parameter value is passed to %READNEW. In the first call to %READNEW, it is the resolved value of the %DO loop that is passed. The resultant macro call will be the same as if the call had been coded without a %DO loop.

```
%readnew(dsout=new1,dsin=dbbio.sur16 dbbio.sur17 dbbio.sur18)
```

Notice that no macro %DO statements were actually passed. Only their resolved values were passed. If you really do need to pass values that contain the symbols % and &, you may need to use quoting functions (see Section 7.1).

You must be careful when using statements like this in a macro call, because the resolved text must be an appropriate parameter value. In this case everything worked out okay, and the correct result was obtained, if not in the way the programmer expected.

It is also interesting to note that the three calls to %READNEW are contained within the macro %DOIT. If this had not been the case (that is, if the calls to %READNEW had been in open code), then the %DO statements would have caused an error because they can be used only within a macro. It is likely that the author of this macro expected the %DO to be inserted into %READNEW and then only surrounded the calls to %READNEW with the %DOIT definition when the first attempt did not work as anticipated.

**SEE ALSO:** Beverly (1999) introduces the %DO, and Leighton (1997) includes examples of %DO loops that are used to build code dynamically.

### 5.3.3 %DO %UNTIL Loops

Continuous loops can also be executed without the use of incremental variables. The %DO %UNTIL statement executes a block repeatedly until the specified condition is met. Be careful when using %DO %UNTIL, because it is possible to construct an infinite loop. Because the %DO %UNTIL statement contains a logical macro expression and the loop continues to execute until the expression is true, be sure that the condition can be met.

#### SYNTAX:

```
%DO %UNTIL(expression);
  . . . text . . .
%END;
```

When the expression in the parentheses is evaluated and the expression is false, the %DO statement is executed again. The loop continues to execute until the expression is true. Like the DATA step DO UNTIL statement, the expression is evaluated at the end of the loop. (This is sometimes expressed as “evaluation at the bottom of the loop.”) This means that the loop always executes at least once and will not execute again if the expression becomes true during the execution of the loop.

Program 5.3.3 rewrites the iterative %DO statement found in Program 5.3.2a using a %DO %UNTIL loop. Because the automatic loop counter of the iterative %DO loop is not available, a loop counter (&YEAR) must be created and incremented manually.

**Program 5.3.3: Using %DO %UNTIL**

```
%macro allyr3(start=,stop=);
  %let year=&start; ①
  %do %until(&year > &stop); ②
    data temp;
      set yr&year;
      year = 2000 + &year;
      run;
    proc datasets lib=work nolist;
      append base=allyear data=temp;
      quit;
    %let year = %eval(&year + 1); ③
  %end;
%mend allyr3;
%allyr3(start=12, stop=14)
```

- ① The loop indicator is initialized to the value of &START.
- ② The values of the two macro variables &YEAR and &STOP are compared to determine whether the loop should execute again. Because the comparison is evaluated at the bottom of the loop, the loop will execute at least once.
- ③ The %EVAL function is used to perform arithmetic operations within the macro language. A special function is required because the macro language does not distinguish between numeric and nonnumeric values. The %EVAL macro function is discussed in more detail in Section 7.3.1.

**MORE INFORMATION:** Program 7.2.3e uses a %DO %UNTIL loop.

**SEE ALSO:** Tassoni, Chen, and Chu (1997) contains an example of a %DO %UNTIL statement. Several usages of the %DO %UNTIL can be found in Vandenbroucke (2016).

**5.3.4 %DO %WHILE Loops**

Like the %DO %UNTIL statement, the %DO %WHILE statement executes a block of code repeatedly. Unlike %DO %UNTIL, however, the %DO %WHILE loops are executed as long as or while the specified condition is met.

**SYNTAX:**

```
%DO %WHILE (expression);
  . . . text . . .
%END;
```

Like the DO WHILE statement in the DATA step, the expression in the %DO %WHILE statement is evaluated (at the top of the loop) before the loop is executed. This means that the loop will not automatically execute at least once as the %DO %UNTIL does. The %DO %WHILE loop executes as long as the *expression* is evaluated as true.

Program 5.3.3 has been rewritten using the %DO %WHILE in Program 5.3.4.

**Program 5.3.4: Using the %DO %WHILE**

```
%macro allyr4(start,stop);
  %let year=&start;
  %do %while(&year <= &stop); ①
    data temp;
      set yr&year;
      year = 2000 + &year;
      run;
    proc datasets lib=work nolist;
      append base=allyear data=temp;
      quit;
```

```
%let year = %eval(&year + 1); ②
%end;
%mend allyr4;
%allyr4(start=12, stop=14)
```

- ❶ The loop will continue to execute as long as this expression is true. When &YEAR exceeds &STOP the loop will terminate. Unlike the %DO %UNTIL, if the initial value of &YEAR is greater than &STOP, the loop will not execute even once.
- ❷ The %EVAL function is used to increment the value of &YEAR.

**MORE INFORMATION:** The %DO %While is used in Programs 8.3.3d, 14.3.5, 14.3.6f, and others.

**SEE ALSO:** Roberts (1997) uses the %DO %WHILE statement to break down a macro variable list.

## 5.4 Additional Macro Program Statements

The macro language is rich in program statements. Several of these have already been introduced and used in previous examples. These statements are highlighted in Table 5.4a. See Appendix 4 for a complete listing of macro language elements covered in this book.

**Table 5.4a: Macro Program Statements Discussed by Section**

Statement	Section
%LET	2.1
%PUT	2.4
%SYMDEL	2.8
%MACRO	3.1
%MEND	3.1
%IF-%THEN/%ELSE	5.2
%DO	5.3.1 and 5.3.2
%DO %UNTIL	5.3.3
%DO %WHILE	5.3.4
%END	5.3.1 through 5.3.4

Table 5.4b highlights a number of additional macro program statements that are introduced in the following subsections of this chapter.

**Table 5.4b: Additional Macro Program Statements to Be Discussed, by Section**

<b>Statement</b>	<b>Section</b>
Macro comments	5.4.1
%GLOBAL	5.4.2
%LOCAL	5.4.2
%SYSEXEC	5.4.3
%ABORT	5.4.4
%RETURN	5.4.5

### 5.4.1 Macro Comments

Macro comments behave in much the same way as the SAS asterisk-style comment. As a general rule, unless you are commenting out macro statements within a macro, writing macro functions (see Section 7.5.2) or dynamic macros (see Chapter 11), there is little advantage to using macro comments over non-macro comments. However, you might notice the following differences:

- The SAS Log will contain asterisk-style comments for the text that is generated by the macro when you are using the MPRINT system option (see Section 3.3.2). Macro comments and comments denoted by /\* .... \*/ will not be shown in the SAS Log.
- Asterisk-style comments are stored as part of the compiled macro and, therefore, take up additional, albeit minimal, resources.
- Macro comments can be used in places and ways that asterisk-style comments cannot be used.
- Inside of a macro definition, asterisk-style comments will generally not comment out macro language elements (see Section 14.3.6) and are not recommended for that purpose.
- SAS recommends that asterisk-style comments not be used inside of a macro definition.

SAS recommends that the /\* .... \*/ style comment be your first choice for commenting your program. These comments are stripped out of your code before any of the other types of comments. Macro comments are removed by the macro facility, and asterisk-style comments are removed last (after macro language elements have been processed).

For my work, I tend to save the use of the /\* .... \*/ style comments to comment out blocks of code. Macro comments are then used to comment on all macro language elements, and asterisk-style comments are used to comment only on non-macro language code.

#### SYNTAX:

```
%* comment text ;
```

Macro comments and asterisk-style comments are complete statements, and as such they are *tokenized*. This means that they cannot contain embedded semicolons or unmatched quotation marks. Since the /\* .... \*/ style comments are processed as individual characters (they are not tokenized), they can offer a bit more flexibility.

While any of the three styles of comments can be used in a program, they are not all displayed in the SAS Log in the same way. When used within a macro definition, only the asterisk-style macro comments appear

in the SAS Log. The %DOIT macro in Program 5.4.1a uses each of the three comment styles within the macro:

#### Program 5.4.1a: Using Macro Comments

```
%macro doit;
/* This macro does nothing of interest;
 * This macro variable &VAR remains unresolved;
 /* Only the previous comment shows with MPRINT */
%mend doit;
```

When %DOIT is executed with the MPRINT system option turned on, the SAS Log shows only the asterisk-style comment following the macro call. This is the only one of the three comments that are stored as a part of the compiled macro %DOIT.

```
399  %doit
MPRINT(DOIT):      * This macro variable &VAR remains unresolved
```

When used in open code, however, all three types of comments will appear in the SAS Log.

Macro comments can be especially useful when you dynamically build SAS code. If for no other reason than the fact that most programmers, even experienced macro programmers, often have difficulties understanding dynamic macros, these types of programs should be well documented. Comment your programs liberally. In Program 5.4.1b, a %DO loop is used to build a SET statement, and a comment would provide helpful documentation. An asterisk-style comment could be used before the SET statement.

#### Program 5.4.1b: Documenting with an Asterisk-Style Comment

```
%macro allyr(start=10,stop=14);
  data allyear;
    * Include the data from the years of interest;
    * Use years from &start to &stop;
    set
    %do year = &start %to &stop;
      yr&year(in=in&year)
    %end;;
  ...Code not shown...
```

For the years 11–13, the DATA step becomes as follows:

```
data allyear;
  * Include the data from the years of interest;
  * Use years from &start to &stop;
  set yr11(in=in11)
    yr12(in=in12)
    yr13(in=in13);
  ...Code not shown...
```

Asterisk-style comments could not be placed just before the %DO statement, because they would not have been stripped out and would have become embedded within the SET statement (Program 5.4.1c).

#### Program 5.4.1c: Asterisk-Style Comments Causing Problems

```
%macro allyr(start=10,stop=14);
  data allyear;
    set
    * The macro DO will generate the names;
    * of the incoming data sets;
    %do year = &start %to &stop;
      yr&year(in=in&year)
    %end;;
  ...Code not shown...
```

When this version of %ALLYR is executed, the asterisk-style comments will cause syntax errors because they become a part of the SET statement:

```
data allyear
  set
  * The macro DO will generate the names;
  * of the incoming data sets;
    yr11(in=in11)
    yr12(in=in12)
    yr13(in=in13);
...Code not shown...
```

Since, of course, an asterisk-style comment cannot be embedded within other statements, it causes syntax errors when the DATA step is compiled.

You could, however, use macro-style comments to document the %DO process within what will become the SET statement (Program 5.4.1d). These comments will be removed during macro compilation and will never be seen by the DATA step compiler.

#### **Program 5.4.1d: Using Macro Comments to Comment on Macro Code**

```
%macro allyr(start=10,stop=14);
  data allyear;
    set
    /* The %DO will generate the names;
    /* of the incoming data sets;
    %do year = &start %to &stop;
      yr&year(in=in&year)
    %end;;
...Code not shown...
```

The use of macro comments also enables you to specify %DO in the comment, whereas in the asterisk-style comment this would have caused an error. You could also use a comment that is defined by /\* ... \*/, as these comments can be embedded within other SAS statements.

**SEE ALSO:** SAS Usage Note 32684 discusses the topic of using comments inside of a macro in more detail. See <http://support.sas.com/kb/32684>.

---

## **5.4.2 %GLOBAL and %LOCAL**

When a macro variable is created, it must be assigned to a symbol table. Very often there is more than one symbol table, and the default rules for determining which symbol table is to receive the macro variable can be quite arcane. The %GLOBAL and %LOCAL statements can be used to both create macro variables and control to which table they are to be assigned.

The %GLOBAL statement names macro variables in the GLOBAL symbol table, and these macro variables will be available in all scopes or referencing environments. The %LOCAL statement is used within a macro definition, which is a limited referencing environment that can have a LOCAL symbol table.

#### **SYNTAX:**

```
%GLOBAL macro-variable-list;
%LOCAL macro-variable-list;
```

Unless first declared otherwise by these statements, macro variables are automatically assigned to a specific symbol table in accordance with a complex set of arcane rules that are applied when the macro variables

are first defined. The default assignment of macro variables to a scope and the issues associated with this assignment are described in more detail in Section 14.5.

When the %GLOBAL and %LOCAL statements are not used, macro variables are *generally*

- Global when they are defined outside of a macro, and
- Local to a macro when they are defined inside of a macro.

In the previous sentence the emphasis is on the word *generally* because there are a number of important, and often subtle, exceptions to these two rules. The examples in this section do not address these exceptions. The detailed rules for determining when a macro variable is to be global or local can be found in Section 14.5.

Referencing environments will not be a problem when you either define macro variables outside of all macros (put them into the global symbol table), or pass macro variable values into and out of each macro as parameters (macro parameters are always local to that macro).

Unfortunately, these approaches often are not practical. For more complicated applications, it may be useful to create a macro variable that you want to have available to other macros without having to physically pass it as a parameter. Understanding the scope of the macro variable becomes important for you at this point. Failure to understand and heed the warnings associated with these concepts can result in macro variable collisions (see Section 14.3.6).

Program 5.4.2a demonstrates macro variable scope and availability of macro variable values through the use of nested macro calls.

#### Program 5.4.2a: Demonstrating Macro Variable Scope

```
%let outside = AAA; ①

%macro one;
  %global inone; ②
  %let inone = BBB;
%mend one;

%macro two;
  %let intwo = CCC; ③
%mend two;

%macro last;
  %one
  %two
  %put &outside &inone &intwo; ④
%mend last;

%last
```

- ① The %GLOBAL statement is not needed to place this macro variable into the GLOBAL symbol table as &OUTSIDE is created outside of any macro. It is, therefore, a GLOBAL macro variable and is automatically available within any macro.
- ② The macro variable &INONE is globalized when it is first defined in the %GLOBAL statement. So, although it is defined inside of a macro (%ONE), it too is now available within other macros.
- ③ The macro variable &INTWO is created inside of %TWO and is not globalized. Its definition, its scope, or even knowledge of its existence is limited to the macro %TWO.
- ④ The %PUT statement in the macro %LAST demonstrates these availabilities. When %LAST is called, the SAS Log shows that the macro variable &INTWO is no longer defined when the %PUT is executed.

```
521 %last
WARNING: Apparent symbolic reference INTWO not resolved.
AAA BBB &intwo
```

The %LOCAL statement seems to be used less frequently than %GLOBAL and is almost certainly used less often than it should be. Most commonly, it is used as a means to protect the values of global macro variables. In Program 5.4.2b, the macro variable &YEAR is created in the macro %ALLYR (see the code associated with Program 5.4.2b for the full definition of %ALLYR3) and will most likely be written to the local symbol table associated with %ALLYR3. However, because of those pesky arcane assignment rules, if %ALLYR3 is called within an application that has a different globalized macro variable of the same name (&YEAR), the assignment of &YEAR in %ALLYR3 will change the global value of &YEAR, not a local value.

#### Program 5.4.2b: Showing a Macro Variable Collision

```
%macro allyr3(start=,stop=);
  %let year=&start;
  %do %until(&year > &stop);
...Code not shown...
%mend allyr3;
%let year = FRED;
%put The global value of year is &year;
%allyr3(start=12, stop=14)
%put The global value of year is &year;
```

The SAS Log shows that the value of &YEAR in the GLOBAL symbol table has been changed by the macro %ALLYR3 from FRED to 15.

```
539  %mend allyr3;
540  %put The global value of year is &year;
The global value of year is FRED
541  %allyr3(start=12, stop=14)
...Text not shown...
542  %put The global value of year is &year;
The global value of year is 15
```

By adding the %LOCAL statement as has been done in Program 5.4.2c, you can make sure that the global value of &YEAR and the local value of &YEAR can both coexist in different symbol tables at the same time. In my opinion every macro definition should have a %LOCAL statement that names every macro variable that is intended to be local.

#### Program 5.4.2c: Using %LOCAL to Prevent Macro Variable Collisions

```
%macro allyr3(start=,stop=);
  %local year; ⑤
  %let year=&start; ⑥
...Code not shown...
%mend allyr3;
%let year = FRED;
%put The global value of year is &year;
%allyr3(start=12, stop=14)
%put The global value of year is &year;
```

- ⑤** The %LOCAL statement adds &YEAR to the local symbol table with a null value.
- ⑥** Because &YEAR is on the local table, all references to &YEAR within %ALLYR3 will be resolved with values from the local table.

Program 5.4.2c shows that it is possible for a macro variable to seem to have more than one definition at a given time, depending on whether it is local or global. This dual (or should we say dueling?) definition can be confusing, and you should avoid it when possible. Although it may at first seem that one macro variable has more than one value, in fact these are different macro variables, in different tables, that happen to have the same name. Since we do not have a way to name or point to the symbol table of interest, we have to

keep track of the symbol tables ourselves. Usually, this is not a major issue, but we do need to pay attention.

**MORE INFORMATION:** Starting in SAS 9.4, you can further protect macro variables by making them READONLY; see Section 8.2.6 for more details.

### 5.4.3 %SYSEXEC

You can issue operating environment commands from within a macro or in open code if you use the %SYSEXEC macro statement. This statement is analogous to the X statement, with the primary difference being in the timing of the execution of the statement. As a macro language element, the %SYSEXEC statement is executed by the macro facility:

- Specified operating system commands are executed immediately.
- Any return codes are assigned to the automatic macro variable &SYSRC.

#### SYNTAX:

```
%SYSEXEC operating system command;
```

Enter the operating system command just as you would enter it at your operating system prompt. The command is *not* enclosed in quotes. Everything between the %SYSEXEC and its semicolon is passed directly to the operating system for immediate execution.

In the following mainframe command, the partitioned data set member CLEANUP will be executed. The quotes are needed here because they are expected by the operating system:

```
%SYSEXEC ex 'userid.mytools.sascode(cleanup)';
```

The following %SYSEXEC statement is followed by a query to its return code, which is stored in the automatic macro variable &SYSRC:

```
%sysexec dir *.sas;
%if &sysrc = 0 %then %do;
```

If you have not used the X statement before on your operating system, you should consult the appropriate *SAS Companion* so that you will know what to expect in terms of behavior for %SYSEXEC and what values can be taken on by &SYSRC.

**MORE INFORMATION:** The example in Section 8.2.4 uses the %SYSEXEC statement to create a system directory under Windows.

**SEE ALSO:** The %SYSEXEC statement is compared to the X statement by Wang (2003).

### 5.4.4 Termination of Macro Execution with %ABORT

Like the ABORT statement in the DATA step, the macro %ABORT statement can be used to control a program's continued execution in accordance with factors that you can specify. When this statement is executed, the execution of the current macro is stopped with an abnormal termination. Depending on the statement's arguments, and how SAS is executing (batch / interactive / AF program), it can potentially stop the SAS job or SAS session. In each case an error message is written to the SAS Log.

#### SYNTAX

```
%ABORT <ABEND|RETURN<n>>;
```

You might need to experiment with this statement and its arguments for your application and operating system. In each case the current macro's execution is terminated. %ABORT without any arguments seems to be the most benign, and with the ABEND argument it is the most severe.

In the following %ABORT statement, the return code of 27 will be passed back to the operating environment:

```
%abort return 27;
```

As is the case with the DATA step ABORT statement, the severity of the termination will depend a lot on how and when the %ABORT is executed and under what operating environment (batch or interactive) the macro is executing. Since this is considered an abnormal termination, how your operating system treats abnormal terminations may also determine whether the whole SAS session is terminated.

**MORE INFORMATION:** The %ABORT statement is used in Section 8.3.2.

**SEE ALSO:** Henry (2015) uses the %ABORT statement to exit a macro after an unexpected return code. The %ABORT statement might cause server termination issues when using SAS Enterprise Guide (Schwarz, 2015).

#### 5.4.5 Normal Termination of Macro Execution with %RETURN

Unlike the %ABORT statement, which results in an abnormal termination of a macro's execution, the %RETURN statement can be used to cause normal termination of a macro. The consequences for using %RETURN as opposed to %ABORT are therefore much less severe.

##### SYNTAX:

```
%RETURN;
```

When the %RETURN statement is executed, which is usually done conditionally with a %IF, the current macro ceases executing, and control is returned to the program immediately after the macro call. The %MEANS macro shown here operates only on data sets with more than 24 observations.

```
%macro means (dsn=);
%if %obscnt(&dsn) < 25 %then %return;
proc means data=&dsn noprint;
...Code not shown...
%mend means;
%means (dsn=sashelp.class)
```

**MORE INFORMATION:** The %OBSCNT macro returns the number of observations in a data set and is described in Program 11.2.6a.

**SEE ALSO:** Hughes (2016b) uses the %RETURN to exit a macro after detecting an error.

## 5.5 Testing Your Knowledge with Chapter Exercises

Complete the following exercises to test your knowledge of the contents of this chapter.

1. Using the macro %PRINTT in Program 5.5.1, construct the macro call that will invoke the PRINT procedure, using the SAS data set SASCLASS.CLINICS.

### Program 5.5.1: Write a Macro Call for the PRINTT Macro

```
%macro printt(dsin,proc=);
  proc &proc data=&dsin;
  run;
%mend printt;
```

2. Use the %DO, %DO %UNTIL, or %DO %WHILE statement to construct a macro that will print the message "This is Test number" in the SAS Log ten times, each time displaying the test number: 1, 2, 3, 4, and so on.
  - a. Extra Credit: Solve this exercise three times, once for each type of %DO.
  - b. The SAS Log should show something like the following:

```
This is Test 1
This is Test 2
This is Test 3
This is Test 4
This is Test 5
This is Test 6
This is Test 7
This is Test 8
This is Test 9
This is Test 10
```

3. (True or False) Every macro definition must end with a %MEND statement.
4. (True or False) The %LET statement can be used inside as well as outside of a macro.
5. Which of these statements are syntactically incorrect? Why?
  - a. %RETURN ABEND;
  - b. %GLOBLE SAVE;
  - c. %IF &SEX = M THEN %PUT MALE;
6. Write a macro that will be generic enough to handle the MEANS and UNIVARIATE procedures. Include the following minimum information as macro variables:
  - a. Name of the procedure
  - b. Title Line 1
  - c. VAR statement for selection of numeric variables
7. Convert the PROC MEANS step in Program 5.5.7 to a macro that will either print a standard set of statistics, or generate the means as shown, depending on whether the user specifies an output data set.

### Program 5.5.7: Generalize this PROC MEANS step

```
proc means data=macro3.clinics noprint;
var ht wt;
output out=stats mean= max= / autoname;
run;
```

- Extra Credit: Enable the user to specify the variable list and the statistics that will be printed or written to the new data set.
8. The rather silly Program 5.5.8 uses the macro variable names &INDAT and &OUTDAT in both of the macros %REGTEST and %DOTESETS. When %REGTEST is called from within %DOTESETS are the values of these variables in %DOTESETS changed? This question can be restated as follows: "Is a %LOCAL statement required?"

#### **Program 5.5.8: Is a %LOCAL Statement Required?**

```
%macro regtest(indat,outdat);
  proc reg data=&indat;
    model count=distance;
    output out=&outdat r=resid;
    run;
%mend regtest;

%macro dotests(indat,outdat);
  data r1;
    set &indat;
    if station=1;
    run;
    %regtest(r1,r1out);

  data r2;
    set &indat;
    if station=2;
    run;
    %regtest(r2,r2out);

  data &outdat;
    set r1out r2out;
    run;
%mend dotests;

%dotests(biodat,bioreg)
```

9. In the macro %TRY shown here (Program 5.5.9), which %PUT statements will be executed?

#### **Program 5.5.9: Which Expressions Evaluate to True?**

```
%let a = AAA;
%macro try;
  %put &=a;
  %if &a = AAA %then %put no quotes;
  %if '&a' = 'AAA' %then %put single quotes;
  %if 'AAA' = 'AAA' %then %put exact strings;
  %if "&a" = "AAA" %then %put double quotes;
  %if "&a" = 'AAA' %then %put mixed quotes;
  %if "&a" = AAA %then %put quotes on one side only;
%mend try;
%try
```

10. The macro %ALLYR (Section 5.3.2) does not meet Y2K programming standards as it operates against two digit years. Fix this macro so that it will work correctly for any years between 1990 and the present. Assume that the incoming data sets will have four-digit years in the names.

# **Chapter 6: Interfacing with Data Set Values**

<b>6.1 Using the SYMPUTX Routine to Create Macro Variables .....</b>	<b>90</b>
6.1.1 Introducing SYMPUTX Syntax .....	91
6.1.2 Comparing SYMPUTX with SYMPUT .....	93
6.1.3 Using a Macro Variable in the Same Step That Created It.....	95
6.1.4 Building a List of Macro Variables.....	96
<b>6.2 Defining Macro Variables in a PROC SQL Step .....</b>	<b>98</b>
6.2.1 Placing a Single Value into a Single Macro Variable .....	98
6.2.2 Building a List of Values .....	99
6.2.3 Placing a List of Values into a Series of Macro Variables .....	102
6.2.4 Understanding Automatic SQL-Generated Macro Variables .....	105
<b>6.3 Moving Text from Macro Variables into Code .....</b>	<b>106</b>
6.3.1 Assignment and RETAIN Statements.....	106
6.3.2 SYMGET and SYMGETN Functions.....	107
6.3.3 The RESOLVE Function.....	109
6.3.4 Comparison of the SYMGET and RESOLVE Functions .....	111
6.3.5 Less-Than-Optimal Uses of SYMGET and RESOLVE .....	115
<b>6.4 Using Data to Control Program Flow .....</b>	<b>116</b>
6.4.1 Assigning Macro Variable Values .....	117
6.4.2 Assigning Macro Variable Names as well as Values .....	119
<b>6.5 Executing Macro Code Using CALL EXECUTE .....</b>	<b>121</b>
6.5.1 Executing Non-Macro Code .....	122
6.5.2 Executing Macro Code.....	123
6.5.3 Addressing Timing Issues .....	125
<b>6.6 Testing Your Knowledge with Chapter Exercises .....</b>	<b>129</b>

Very often in macro language programs and applications you will want the data itself to drive the definitions of the macro variables and parameters. The data used to create the macro variable definitions might be the same data that is to be analyzed, or you might have some kind of control file that can be used to provide directions to your program. In either case, you need the ability to take information that is stored in a SAS data set or in some other format and convert it into values for macro variables. This chapter introduces various aspects of this process.

For the examples in this chapter and indeed for all of the code examples throughout the book, if you want to execute these sample programs, then be sure to follow the setup instructions. Remember that all of the data sets and programs are available for download, so you do not need to retype either the code or the data. For instructions on accessing and setting up the programs and data, see the “Example Code and Data” section within this edition’s “About This Book” front matter.

## 6.1 Using the SYMPUTX Routine to Create Macro Variables

In Section 1.4 it was shown that macro language references are resolved and executed long before the DATA step is compiled and executed. Although the diagram in Figure 1.4d is fairly straightforward, its implications are not so easily seen in some of the code that we write.

The DATA step in Program 6.1a represents one of the most commonly asked questions of SAS Technical Support relative to the macro language. Here the programmer wants to assign Jane's age, which is held in the data set variable AGE, to the macro variable &JANE\_AGE.

### Program 6.1a: Conditional Execution of the %LET

```
data age;
  set sashelp.class (where=(name='Jane')) ;
  %let jane_age = age;
  run;
%put &=jane_age;
```

The SAS Log shows that the value actually stored in &JANE\_AGE is in fact the letters a-g-e:

```
5   %put &=jane_age;
JANE_AGE=age
```

Although from a DATA step programmer's point of view, it appears that the %LET statement is a part of the DATA step, Figure 1.4d shows us otherwise. The %LET will be executed by the macro facility as soon as it is encountered. This will be *before* the DATA step has even been compiled, *before* there is a Program Data Vector (PDV), *before* there is even an AGE variable, and *long before* any data has been read. Placing the %LET inside of the DATA step like this is an attempt to conditionally execute (using the implied loop of the DATA step) the %LET statement. And this just does not work. The timing shown in Figure 1.4d is against us. Although you cannot conditionally execute macro language elements directly using a DATA step language element, you can do so indirectly using the CALL EXECUTE routine (see Section 6.5) and the DOSUBL function (see Section 8.5.1).

When we want to move a value from the PDV to a symbol table, we need to use a non-macro language element (one without either an & or a %). Fortunately, there are tools that we can use to transfer information from the PDV to the symbol tables. In the DATA step, we have the SYMPUTX routine and the older SYMPUT routine. Like the %LET statement, the SYMPUTX call routine can be used to assign a value to a macro variable. However, SYMPUTX is not a macro-level statement; instead, it is an executable DATA step routine. As such, it is used as part of a DATA step and enables you to directly assign values of data set variables to macro variables during DATA step execution.

The routine has three arguments, and like any other DATA step function these can be any combination of a variable name or a constant value (such as a character string). The first argument identifies the name of the macro variable, the second argument identifies the value to be assigned to the macro variable, and the optional third argument allows specification of a local or global symbol table.

#### SYNTAX:

```
CALL SYMPUTX (macro_varname,value<,symbol_table_code>);
```

The SYMPUTX routine replaces the older, less flexible SYMPUT routine, which had a couple of disadvantages. SYMPUT expects the second argument to be character, and it has no ability to control which symbol table is to receive the macro variable. The only disadvantage of SYMPUTX is that its name contains an X, which is a bit harder to type.

In Program 6.1b SYMPUTX is used to assign the value of the DATA step variable AGE to the macro variable &JANE\_AGE.

**Program 6.1b: Using SYMPUTX to Create a Macro Variable**

```
data age;
  set sashelp.class (where=(name='Jane'));
  call symputx('jane_age',age);
  run;
%put &=jane_age;
```

Using this assignment, you see in the SAS Log that the macro variable &JANE\_AGE now correctly contains Jane's age:

```
13  %put &=jane_age;
JANE_AGE=12
```

**MORE INFORMATION:** By default both the SYMPUTX and SYMPUT routines will create a global macro variable when used in open code. However, when used inside of a macro the macro variable will usually be local to that macro. The detailed default rules for determining whether the macro variable will be global or local are described in Section 14.5.4.

**SEE ALSO:** An overview of using the SYMPUT routine can be found in Palmer (1998 and 2001).

## 6.1.1 Introducing SYMPUTX Syntax

### First Argument of SYMPUTX

The first argument of the SYMPUTX routine is used to name the macro variable that is to either be created or assigned a value. The argument might be either a character string, DATA step variable, or a combination of the two. Each of the following three examples assigns a value to the macro variable &DSN1.

**Character string.** A *character string* enclosed in quotes creates a macro variable with the name that is enclosed in the quotes:

```
call symputx('dsn1',argument2);
```

**Character variable.** When the first argument is a DATA step *character variable*, its value becomes the macro variable name:

```
macvar = 'dsn1';
call symputx(macvar,argument2);
```

**Character variable and string used together.** A character expression, which can be made up of any combination of character variables and strings, may be used to determine the macro variable name:

```
n=1;
call symputx('dsn'||left(put(n,12.)),argument2);
```

In this case 'DSN' will have a '1' appended onto it forming 'DSN1'. Notice that the PUT function converts the numeric variable N to a character string that is then left-justified and then appended using the concatenation operator (||). The LEFT and PUT functions can be replaced with the CATT function to simplify the first argument.

```
call symputx(catt('dsn',n),argument2);
```

### Second Argument of SYMPUTX

The second argument contains the value that is to be assigned to the macro variable named in the first argument. As is the case with the first argument, the second argument can be a character string, variable name, or a combination of the two. In each of the following examples, the macro variable &DSN1 is assigned the value of CLINICS.

**Character string.** The character string becomes the value assigned to the macro variable:

```
call symputx('dsn1','clinics');
```

**Character variable.** The value of the DATA step variable DATASET is assigned to the macro variable:

```
dataset='clinics';
call symput('dsn1',dataset);
```

**Character variable and string used together.** The resolved value of the character expression is assigned to the macro variable:

```
partname='clin';
call symput('dsn1',catt(partname,'ics'));
```

The CATT function removes trailing blanks and is needed if PARTNAME has a length that is greater than the number of nonblank characters. For example, if the length of PARTNAME was \$6, and you did not perform a trim prior to concatenation, the character expression would become CLIN ICS, and this string, including the embedded blanks, would be stored in &DSN1.

## SYMPUTX Example

In Program 6.1.1 a clinic number is supplied to the macro %PRINCLIN. The clinic number is used to retrieve the clinic name, which is used in the TITLE1 statement.

### Program 6.1.1: Using the SYMPUTX Routine

```
%macro princlin(clnum=123456);
data clin&clnum;
  set macro3.clinics(where=(clnnum="&clnum")) ①
    end=eof; ②
  if eof ③ then call symputx('clname',clinnname,'l'); ④
  run;
title1 "Data for &clname"; ⑤
proc print data=clin&clnum;
  var lname fname dob symp;
  run;
%mend princlin;
%princlin(clnum=043320)
```

- ① The incoming value of the clinic number is used to subset the data.
- ② The END= option on the SET statement creates a temporary variable (EOF) that will be true only for the last observation that meets the WHERE criteria to be read from the incoming data set.
- ③ EOF will be true ( 1 ) for only one observation.
- ④ When processing the last observation for this clinic number, create the macro variable &CLNAME using the DATA step variable CLINNAME. Because the SYMPUTX routine has more overhead than the IF statement, resources are saved by conditionally executing the SYMPUTX routine only once. This call to the SYMPUTX routine places the macro variable &CLNAME in the local symbol table by using the optional third argument (see Section 6.1.2 for more on the use of this argument).
- ⑤ The TITLE1 statement cannot be placed either before or within the DATA step because the macro variable &CLNAME must first be defined by the SYMPUTX ④.

**MORE INFORMATION:** Additional examples of the use of SYMPUTX and SYMPUT can be found throughout the remainder of this book.

**SEE ALSO:** Landers and Bryher (1997) has several examples that highlight various behaviors of the SYMPUT routine under different conditions.

### 6.1.2 Comparing SYMPUTX with SYMPUT

The SYMPUTX routine is a more recent addition (SAS 9) to the DATA step than the SYMPUT routine, and although SYMPUTX has now been in the language for several releases, SYMPUT is still commonly used both in new and legacy code. The SYMPUT routine is still completely functional, however it just does not offer the flexibility of the SYMPUTX routine. For new code, most macro programmers now prefer to use SYMPUTX. When modifying existing or legacy code, I will generally convert any uses of SYMPUT to SYMPUTX. However, for code that is executing successfully, and does not otherwise need to be modified, I will usually not go out of my way to convert the usages of SYMPUT to SYMPUTX.

SYMPUTX has two major advantages over SYMPUT:

- The way it handles numeric values in the second argument
- The optional third argument that controls symbol table placement of the generated macro variable

A secondary advantage of SYMPUTX is that values are automatically both left-justified and trimmed before being assigned to the macro variable.

#### Numeric Second Argument

When the second argument is numeric, SYMPUT first converts it to a character string before storing the converted value in the macro variable. This conversion process generates a conversion note in the SAS Log and produces a right-justified string. SYMPUTX not only converts the value without generating a note, but left-justifies and trims it as well. For most users, the convenience of this automatic conversion is sufficient to justify the use of SYMPUTX over SYMPUT. Program 6.1.2a demonstrates the differences in the conversion of a numeric second argument.

#### Program 6.1.2a: Handling a Numeric Second Argument

```
data _null_;
x=5;
call symput('x1',x); ①
call symput('x2',trim(left(put(x,6.)))); ②
call symputx('x3',x); ③
run;
%put The vertical bars show leading and trailing blanks.;
%put Using SYMPUT X1 is |&x1|;
%put Using SYMPUT with TRIM LEFT PUT X2 is |&x2|;
%put Using SYMPUTX X3 is |&x3|;
```

The SAS Log shows the difference between how SYMPUT and SYMPUTX convert the value of the numeric variable X before storing it in the macro variable:

```
NOTE: Numeric values have been converted to character values at the places
given by: ④
(Line):(Column).
156:18
NOTE: DATA statement used (Total process time):
      real time          0.01 seconds
      cpu time           0.01 seconds

160 %put The vertical bars show leading and trailing blanks.;

The vertical bars show leading and trailing blanks.
161 %put Using SYMPUT X1 is |&x1|;
Using SYMPUT X1 is |      5| ①
162 %put Using SYMPUT with TRIM LEFT PUT X2 is |&x2|;
Using SYMPUT with TRIM LEFT PUT X2 is |5| ②
163 %put Using SYMPUTX X3 is |&x3|;
Using SYMPUTX X3 is |5| ③
```

- ❶ The converted value is right-justified. The default conversion format is BEST12.
- ❷ The use of the TRIM/LEFT/PUT functions result in a trimmed value that does not generate the conversion note.
- ❸ SYMPUTX also generates the trimmed value without the explicit use of the TRIM/LEFT/PUT functions.
- ❹ SYMPUT issues a numeric to character conversion NOTE.

### Optional Third Argument

The optional third argument for SYMPUTX is a code that can be used to alter or control the destination symbol table. Although both SYMPUT and SYMPUTX by default generally write to the most local symbol table, this is not always the case (Section 14.5.4 discusses the rules for these assignments). When the third argument is specified as one of the following case insensitive codes, the macro variable will be written using one of three possible schemes.

**Table 6.1.2: Values for the Optional Third Argument**

Code	Location of the New Macro Variable
G	The global symbol table
L	The most local symbol table that currently exists
F	The most local symbol table that already contains this macro variable, or, if the variable does not yet exist, the most local table that currently exists

For many programmers who are just starting to use the macro language, the use of this third argument will probably seem fairly unimportant. However, as the macros become more sophisticated the placement of the macro variables in appropriate symbol tables becomes crucial. Program 6.1.2b demonstrates a few of the various possible combinations of situations that are effected by the rules that are described in Section 14.5.4.

### Program 6.1.2b: Using the Third Argument of SYMPUTX

```
%macro placeit(val=); ❶
  data _null_;
  string=&val";
  call symput('c1',string); ❷
  call symputx('c2',string); ❸
  call symputx('c3',string,'f'); ❹
  call symputx('c4',string,'g'); ❺
  call symputx('c5',string,'l'); ❻
  run;
  %put _user_;
  %mend placeit;
  %let c3=inglobal; ❻
  %placeit(val=clinics)
```

The SAS Log shows the various symbol table assignments of the macro variables generated within the %PLACEIT macro:

```
PLACEIT VAL clinics ❶
PLACEIT C1 clinics ❷
PLACEIT C2 clinics ❸
GLOBAL C3 clinics ❹ ❺
GLOBAL C4 clinics ❺
PLACEIT C5 clinics ❻
```

- ❸ Assigning &C3 in open code places it in the global symbol table.
- ❹ As a parameter, not only is &VAL in the local table, but also it establishes the local table.
- ❺ The table assignments for each of these macro variables use the same rules; however, unlike &C1 and &C2, because &C3 already exists in the global table (❸), this value replaces the existing value of &C3 in the global table. Changing the ‘f’ to ‘l’ in the SYMPUTX would force &C3 to be written to the local table, and there would be two macro variables named &C3, each in a different symbol table and each having a different value.
- ❻ &C4 is forced into the global table.
- ❼ &C5 is forced into the local table.

### 6.1.3 Using a Macro Variable in the Same Step That Created It

It is unfortunately all too common for new users to try to access a macro variable in the same DATA step in which it is created. While this is generally a poor idea (usually it shows a poor understanding of the operation of the DATA step and the use of the PDV), it can be done successfully using DATA step functions and routines. However, you cannot define a macro variable using SYMPUTX and then attempt to access that macro variable using an ampersand (&) in the same DATA step. This makes sense (see Figure 1.4d) when you remember that values are assigned to the macro variable through SYMPUTX during DATA step execution. However, any macro variable references involving the use of an & are resolved even *before* the compilation phase of the DATA step. And since the DATA step execution phase comes after the compilation of the entire DATA step is complete, the macro facility would be attempting to resolve a macro variable that would not be defined until the DATA step's execution phase. It is generally considered unfair to ask any language to resolve or use variables that have not yet been defined. Program 6.1.3a demonstrates that the macro variable's value cannot be obtained by referencing it in the same step that it is created with an &.

#### Program 6.1.3a: Retrieving in the Same Step Using an &

```
data age;
  set sashelp.class(where=(name='Jane'));
  call symputx('jane_age',age);
  j_age = &jane_age;
run;
```

The SAS Log shows that the macro variable &JANE\_AGE has not yet been defined when the macro facility attempts to resolve the macro variable reference. This macro variable will not be created until DATA step execution. It is not about the order of the statements within the DATA step; it is all about the order of events as outlined in Figure 1.4d.

```
24  data age;
25    set sashelp.class(where=(name='Jane'));
26    call symputx('jane_age',age); ❷
27    j_age = &jane_age; ❸ ❹
      -
      22
WARNING: Apparent symbolic reference JANE_AGE not resolved.
ERROR 22-322: Syntax error, expecting one of the following: a name,
a quoted string, a numeric constant, a datetime constant, a missing
value, INPUT, PUT.

28      run;
```

- ❶ Before this DATA step statement can even be compiled, the macro variable reference &JANE\_AGE must be resolved. Because at this point this variable has not yet been defined by the execution of the CALL SYMPUTX, a warning message stating that &JANE\_AGE is not resolved will be written to the SAS Log.
- ❷ During DATA step compilation, this statement is prepared for the execution phase. If the compilation of the DATA step is successful, then during the execution phase the macro variable &JANE\_AGE will receive the value contained in the numeric DATA step variable AGE.

- ❸ Because the &JANE\_AGE was unresolved during the macro resolution phase, the expression on the right of the equal sign will still contain the characters exactly as they are written, including the &. This causes the syntax error, which prevents the DATA step from compiling successfully.

Program 6.1.3b shows that, had the macro variable in Program 6.1.3a ❸ been quoted as "&JANE\_AGE", there would have been no syntax error, the DATA step would have compiled, and the variable J\_AGE would have received the value &JANE\_AGE (including the &).

#### Program 6.1.3b: Retrieving in the Same Step Using an &

```
data age;
  set sashelp.class (where=(name='Jane')) ;
  call symputx('jane_age',age);
  j_age = "&jane_age"; ❹
run;
```

The SAS Log shows that the macro variable &JANE\_AGE fails to resolve as before; however, because we are creating a character variable this time (as compared to ❸ above), there is no illegal character to cause a syntax error:

```
56   data age;
57     set sashelp.class (where=(name='Jane')) ;
58     call symputx('jane_age',age);
59     j_age = "&jane_age";
WARNING: Apparent symbolic reference JANE_AGE not resolved.
60   put j_age=;
61   run;

j_age=&jane_age
```

- ❹ When a macro variable fails to resolve, the macro facility marks the ampersand so that it will no longer be seen as a macro trigger. As a result, the constant text including the ampersand is stored in the character variable J\_AGE.

**MORE INFORMATION:** It is very rare that you truly need to build a macro variable and then use it again in the same DATA step. After all, you have the value in a DATA step variable already, so why not use it? When you do need to access the macro variable symbol tables from within a DATA step, you will need to use either the SYMGET or RESOLVE functions. These are described in more detail in Sections 6.3.2 and 6.3.3. Additional discussion of the use of a macro variable in the same step as it is created can be found in Section 6.3.4. A legitimate use of macro variables in the same step as they were created can be found in the section on macro hash tables (see Section 9.3).

---

## 6.1.4 Building a List of Macro Variables

It is often helpful to build a series of macro variables from within a single DATA step. In each of the Programs 6.1b and 6.1.1 a single macro variable is created. It is very common that we want to build more than one macro variable and we can do this in the DATA step (see Section 6.2.3 for a description of how to do this in a PROC SQL step).

The most common technique, and probably the one with the most utility, uses a numbered list of macro variables that are named with a common prefix. In Program 6.1.4 the DATA step is used to build a list of macro variables with the prefix of NAME. The resulting variables are in the form of &NAME1, &NAME2, and so on.

**Program 6.1.4: Creating a Numbered List**

```
%macro lstnames;
%local i;
proc sort data=macro3.clinics(where=(lname=: 'S'))
    out=clinics; ①
by lname fname;
run;

data _null_;
set clinics end=eof;
by lname; ②
if first.lname then do; ③
    cnt+1; ④
    call symputx(catt('name',cnt),lname,'l'); ⑤
end;
if eof then call symputx('nameN',cnt,'l'); ⑥
run;

%do i = 1 %to &namen; ⑦
    %put &&name&i; ⑧
%end;
%mend lstnames;
%lstnames
```

- ① Sort the incoming data set so that BY-group processing can be used in the DATA step.
- ② Use the BY statement to create the FIRST.LNAME temporary variable.
- ③ Select each unique last name once.
- ④ Each last name is counted.
- ⑤ Each last name (LNAME) is saved to a macro variable in the local symbol table. Each macro variable name starts with the prefix 'NAME', and the CATT function is used to concatenate the number of each successive name (CNT) onto the prefix. The CATT function not only does the concatenation, but it converts CNT to character and performs a LEFT-TRIM as part of the concatenation process.
- ⑥ The final count of names is saved in the macro variable &NAMEN, which is written to the local symbol table.
- ⑦ A %DO loop ranging from 1 to the number of names (&NAMEN) can be used to cycle through the list of macro variables.
- ⑧ The list of macro variables is addressed by using what is known as the &&VAR&I construct (this is also known as *macro indirect referencing*). The double ampersand resolves to a single ampersand and delays resolution until the index variable (&I) is resolved. When &I contains a 3, &&NAME&I will, on the first scanning pass, resolve to &NAME3, which in turn, on the second scanning pass resolves to Smith.

The names that meet the criteria are written to the SAS Log:

Saunders
Simpson
Smith
Stubs

**MORE INFORMATION:** The resolution process of macro variables with multiple ampersands is discussed in more detail in Section 2.5.3.

In what is often simpler coding, PROC SQL can also be used to create lists of macro variables (see Section 6.2.3).

Numerous examples of the generation of vertical macro variable lists can be found throughout the book. See especially Chapter 11 on dynamic programming methods.

## 6.2 Defining Macro Variables in a PROC SQL Step

The SQL step can be especially useful when defining and populating macro variables. The %LET statement cannot be used within an SQL step to store data set values for the same reasons that it cannot be used within a DATA step (see Figure 1.4d). Teaching the SQL step is outside the scope of this book; however, since it can be used to create and populate macro variables, the discussion of certain aspects of SQL are appropriate.

When defining a macro variable within an SQL step, the INTO clause is used with the colon (:) to designate the name of the macro variable(s) to be created. You can use macros and macro variables within a PROC SQL step in much the same way as in other PROC steps. The exception is the syntax related to creating macro variables within SQL.

The SYMPUTX routine is not available in an SQL step, and there is no straightforward way to force a macro variable onto a particular symbol table when building macro variables dynamically in SQL.

Unlike most other procedures, statements within an SQL step are processed sequentially and are executed immediately. Furthermore, SQL statements are handled differently than the statements of other procedures and are expected to conform to specific syntax standards.

**MORE INFORMATION:** Additional examples of the use of PROC SQL to define and populate macro variables can be found throughout the book (see Sections 11.2.2, 11.4.1, and 12.1.1).

**SEE ALSO:** An introduction and additional detail on the topic of using SQL to populate macro variables can be found in Zdeb (1999b), Shi and Roberge (2003), and Satchi (2000 and 2001). Tassoni (1997) uses a macro to write SQL statements and Palmer (1997) uses macro variables in a PROC SQL step. Examples of SQL steps that create a series of macro variables can be found in Eddlestone (1997), Satchi (2000), and Casas (2002). First (2001b), as well as Rajecki and Kahle (2000), both create a list of macro variables with a single INTO clause.

### 6.2.1 Placing a Single Value into a Single Macro Variable

A PROC SQL step is used in Program 6.2.1 to count the number of observations that contain a specified string (&CLN) in the table column CLINNAME. The SQL COUNT function is used to count the observations that match the WHERE clause criteria and the result is written to the macro variable &NOBS.

#### Program 6.2.1: Storing a Count in a Macro Variable

```
%let cln = Beth;
proc sql noprint;
  select count(*)
    into :nobs ①
    from macro3.clinics(where=(clinnname=:&cln")) ; ③
  quit;
%put number of clinics for &cln is &nobs; ②
```

- ① The INTO clause is used to create the new macro variable NOBS. The colon informs the SELECT statement that the result of the COUNT function is to be written into a macro variable.
- ② Once created, the new macro variable is used in the same way as any other macro variable. Both macro variables are preceded by an ampersand in the %PUT statement.

- ③ Notice that the colon in the FROM clause is used as a character comparison operator in the data set WHERE= option just as it is in the DATA step. SAS will not be confused by these two very different uses of the colon.

The SAS Log shows that the macro variable &NOBS created by SQL is right-justified with a series of leading blanks:

```
78  %put number of clinics for &cln is &nobs;
      number of clinics for Beth is        4
```

There are a number of ways to eliminate those leading blanks if necessary. One of the easiest is to use the %LET statement, because it automatically removes leading and trailing blanks:

```
%let nobs=&nobs;
```

Within SQL, macro variables are also known as environmental variables. When being assigned values, the name of the macro variable will follow an INTO clause and will be preceded by a colon. As is shown in Program 6.2.1, when macro variables are to be used within an SQL step the macro variable is preceded by the ampersand.

**MORE INFORMATION:** More complex examples of SQL steps can be found throughout Sections 6.2 and 11.2.

The %LEFT function can also be used to left-justify text. It is first described in Section 7.2.6.

## 6.2.2 Building a List of Values

It is often useful to be able to build a macro variable that contains a list of values or words (see Section 11.4, which concentrates on the creation and use of lists). You might also want to build a list that can be used with the IN operator, perhaps to be used in a WHERE statement. This list form is often referred to as a horizontal list (see Section 6.2.3 for a description of a vertical list). Remember that a macro variable can contain no more than 64K characters, and while this number is large, it can easily be exceeded when building the list dynamically. Also be sure that none of the words in this list contain whatever character is used as the word delimiter.

In Program 6.2.2a we want to create a list of student names in the SASHELP.CLASS data set. The SEPARATED BY phrase follows the name of the macro variable that is to be created. This phrase identifies the list delimiter. Without this phrase only one value would be stored in &NAMELIST.

### Program 6.2.2a: Storing a List in a Macro Variable

```
proc sql noprint;
  select name
    into :namelist separated by' ' ❶
    from sashelp.class;
  quit;
%put Names are: &namelist;
```

The SAS Log shows that the macro variable &NAMELIST contains a space separated list of the names in SASHELP.CLASS:

```
79  %put Names are: &namelist;
Names are: Alfred Alice Barbara Carol Henry James Jane
Janet Jeffrey John Joyce Judy Louise
Mary Philip Robert Ronald Thomas William
```

- ❶ The SEPARATED BY phrase is used to allow more than one value to be stored in the macro variable &NAMELIST.

It is possible to create more than one macro variable in the SELECT statement. In the following example two macro variables are created (&LASTNAMES and &DOBIRTHS), and each will contain a comma-separated list of values.

#### Program 6.2.2b: Storing Multiple Comma-Separated Lists

```
proc sql noprint;
  select lname, dob ②
    into :lastnames separated by ',', ③
        :dobirths separated by ','
      from macro3.clinics(where=(lname='S'));
  %let numobs=&sqllobs; ④
  quit;
  %put lastnames are: &lastnames; ⑤
  %put dobirths are: &dobirths; ⑥
  %put number of names is: &numobs;
```

- ② The values of the variables LNAME and DOB will be written to macro variables.
- ③ The list of values of LNAME will be written into &LASTNAMES with the values separated by a comma. Without the SEPARATED BY clause, the individual values will replace previous values rather than be appended onto the list. The last comma is needed to separate the two macro variables (:LASTNAMES and :DOBIRTHS).
- ④ The SQL step automatically creates the macro variable &SQLOBS. In this case, it contains the number of rows in the incoming data set that meet the WHERE criteria, which will also be the number of values in each macro variable. Usually, it is a good idea to save this value in a different macro variable, because &SQLOBS will probably be changed with the next SQL statement.

The %PUT statements show the values of these macro variables in the SAS Log:

```
110 %put lastnames are: &lastnames; ⑤
lastnames are: Smith,Simpson,Simpson,Stubs,Saunders
111 %put dobirths are: &dobirths; ⑥
dobirths are: 18MAR52,18APR33,18APR33,11JUN47,01MAR49
112 %put number of names is: &numobs;
number of names is: 5
```

- ⑤ The data set MACRO3.CLINICS contains five observations with a last name, LNAME, which starts with an 'S'.
- ⑥ In the data set the variable DOB has already been assigned the format DATE7., and the format is automatically applied before the values are written to the macro variable &DOBIRTHS.

Except when it is to be displayed, the unformatted date is usually easier to work with, and although the variable DOB in the data set MACRO3.CLINICS has been assigned the DATE7. format, you might want to actually store the unformatted SAS dates in the macro variable &DOBIRTHS. The FORMAT= option is used in Program 6.2.2c to unformat the date before it is stored in the macro variable.

#### Program 6.2.2c: Storing Unformatted Values

```
proc sql noprint;
  select dob format=6. ⑦
    into :dobirths separated by ','
      from macro3.clinics(where=(lname='S'));
  quit;
  %put &=dobirths;
  title1 '6.2.2c Selected patients';
  proc print data=macro3.clinics;
    where dob in(&dobirths); ⑧
    var lname fname dob;
  run;
```

- 7 The FORMAT= option is used to give the variable DOB a new (neutral) format. This allows the actual SAS date value to be stored in the macro variable &DOBIRTHS.

```
DOBIRTHS=-2845,-9754,-9754,-4587,-3958
```

- 8 The comma-separated list of dates is used in a WHERE statement. The formatted values would have caused an error, however now the unformatted values are actually written into the WHERE statement. The resolved WHERE statement shows the actual date values.

```
WHERE dob in (-9754, -4587, -3958, -2845);
```

You may notice a couple of differences in this list of dates compared to the ones generated by the %PUT 7. Here there are only four dates (not five—there is a duplicate birthdate), and they are ascending order. This change is part of the process of the optimization of the WHERE clause.

When storing numeric values in a macro variable, the SEPARATED BY phrase can be used to effectively perform a TRIM/LEFT on the value. Here the values of HEIGHT and WEIGHT are stored respectively in &HT and &WT.

```
proc sql noprint;
select height, weight
  into :ht, :wt
    from sashelp.class
      where name='Alfred';
quit;
%put |&ht|;
%put |&wt|;
```

The SAS Log shows that the values are right-justified, indicating that, although no note was generated, there was an automatic numeric-to-character conversion (much like the one performed by CALL SYMPUTX).

```
461  %put |&ht|;
|       69|
462  %put |&wt|;
|   112.5|
```

There are a number of ways to left-justify a macro variable. One is to simply reassign it using a %LET statement.

```
%let ht=&ht;
```

We can also perform the left-justification through the use of the SEPARATED BY phrase in the SQL step. In Program 6.2.2d this phrase is used even though there is only one value to be stored in the list (&HT is a list of one).

#### Program 6.2.2d: Storing Left-Justified Numeric Values

```
proc sql noprint;
select height, weight
  into :ht separated by ' ',
        :wt separated by ' '
    from sashelp.class
      where name='Alfred';
quit;
%put |&ht|;
%put |&wt|;
```

The SAS Log shows that the values have been both left-justified and trimmed:

```
470 %put |&ht|;
|69|
471 %put |&wt|;
|112.5|
```

**MORE INFORMATION:** &SQLOBS is one of several automatic macro variables that are generated by each SQL step. Section 6.2.4 describes these macro variables in more detail.

**SEE ALSO:** Widawski (1997a) and White (2001) both create a list of filenames using SQL and each places the list in a macro variable.

### 6.2.3 Placing a List of Values into a Series of Macro Variables

Rather than placing a list of values into one macro variable (horizontal list), as was done in Section 6.2.2, you can create a series of macro variables, each with its own distinct value. Often referred to as a *vertical list* of macro variables, this form of a macro variable list overcomes the two major disadvantages of the horizontal list:

- Each item in the list can contain up to 64K characters
- We do not need to worry whether any given item contains a list delimiter

This type of list can be created in a DATA step (see Program 6.1.4), however unless a DATA step is already required, it is often easier to build the list using a PROC SQL step.

In Program 6.2.3a PROC CONTENTS is used to create a table that will contain one row for each variable in the original data set (&DSN). PROC SQL is then used to read this list and to write the variable names into a series of macro variables of the form &VARNAME1, &VARNAME2, and so on.

#### Program 6.2.3a: Creating a Numbered List of Macro Variables

```
%macro varlist(dsn=);
%local i;
* Determine the list of variables in this
* base data set;
proc contents data= &dsn ❶
      out= cont noplay;
run;

* Collect the variable names;
proc sql noplay;
  select distinct name ❷
    into :varnamel->:varname999 ❸
    from cont;
quit;

%do i = 1 %to &sqlobs; ❹
  %put &i &&varname&i;
%end;
%mend varlist;

%varlist(dsn=macro3.clinics)
```

- ❶ The OUT= data set (WORK.CONT) has one row per variable in &DSN. The names of these variables are stored in the variable NAME.
- ❷ Select each unique variable name from NAME (they should already be unique).
- ❸ The variable names are stored in a series of macro variables &VARNAME1, &VARNAME2, and so on. As it is currently coded, no more than 999 values can be stored. This is way more than is needed in

this example, but SAS will only create the number of values that are actually required (not the full 999).

**CAVEAT:** When specifying the upper bound for the number of macro variables (999 in this case), you should be very sure that the number is large enough. If it is too small, values that do not fit will be lost and no error or warning will be issued.

- ④ The number of distinct values of NAME is stored in &SQLOBS. This value can be used to cycle through the list of macro variables. The %DO loop is included here only as an illustration, and would not normally be used in a macro such as %VARLIST.

The data set MACRO3.CLINICS has 20 variables, and the SAS Log shows the following:

```

1 ADMIT
2 CLINNAME
3 CLINNUM
4 DEATH
5 DIAG
6 DISCH
7 DOB
8 DT_DIAG
9 EDU
10 EXAM
11 FNAME
12 HT
13 LNAME
14 PROCED
15 RACE
16 REGION
17 SEX
18 SSN
19 SYMP
20 WT

```

Because of the large penalty for specifying an upper bound that is too small, there are a couple of ways to make sure that the value is large enough. Since it does not matter if the upper bound is way over what you need, you can use the automatic macro variable &SYSMAXLONG, which contains the largest integer supported by your operating system.

#### Program 6.2.3b: Using &SYSMAXLONG as the Upper Bound

```

select distinct name
  into :varname1-:varname&sysmaxlong
     from cont;

```

Starting in SAS 9.3 the upper bound does not actually even need to be specified. Program 6.2.3c shows that you can follow the first specification with a dash and then just not write an upper bound. Notice that the dash following the first macro variable name must be included. Without it only one macro variable would be created.

#### Program 6.2.3c: Specifying Only the Lower Bound

```

select distinct name
  into :varname1-
     from cont;

```

You can also specify the list of macro variables so that the numbers have leading 0s. The resulting list of macro variables could be named: &VARNAME001, &VARNAME002, and &VARNAME003. Although specifying your list this way would make it easier to sort the macro variable names, it would make it more difficult to step through the variables when you are using an iterative %DO loop.

**Program 6.2.3d: Specification with Leading Zeros**

```
select distinct name
  into :varname001-:varname999
    from cont;
```

One of the major advantages of using the SQL INTO clause is that it enables you to create multiple lists, which are coordinated with the same index value. In Program 6.2.3e the SQL step builds two series of macro variables. In this example, we save the unique values of CLINNAME and count the number of observations within each unique clinic. The clinic names and the corresponding counts are each stored in macro variables.

**Program 6.2.3e: Creating Coordinated Numbered Lists**

```
proc sql noprint;
  select distinct clinname, count(*) ⑤
    into :name1-, ⑥
      :cnt1-
    from macro3.clinics ⑦
      group by clinname; ⑧
quit;
```

- ⑤ Select the unique values of CLINNAME using the DISTINCT keyword and use the COUNT function to count the observations within each clinic.
- ⑥ The names will be placed in the macro variables &NAME1... and the counts in &CNT1.... Since there are 27 clinics, &NAME28 will not be created.
- ⑦ The variable CLINNAME is drawn from MACRO3.CLINICS.
- ⑧ The GROUP BY clause causes the COUNT function to count within groups as opposed to across all groups.

Although the coding to create coordinated lists is usually easier in a PROC SQL, the same set of macro variables could also be created by using a combination of PROC SUMMARY and the DATA step.

**Program 6.2.3f: Creating Coordinated Lists by Using the DATA Step**

```
proc summary data=macro3.clinics nway;
  class clinname;
  output out=cnt; ⑨
  run;
data _null_;
  set cnt;
  i+1;
  call symputx(catt('name',i),clinnname);
  call symputx(catt('cnt',i),_freq_); ⑩
  call symputx('namecnt',i);
run;
```

- ⑨ The data set WORK.CNT will contain one row per clinic name and the only statistic will be the observation count (\_FREQ\_).
- ⑩ The CATT function is used to concatenate the row counter with the macro variable name. By using SYMPUTX in preference to SYMPUT, the numeric-to-character conversions do not generate a note in the SAS Log.

**MORE INFORMATION:** An example of building a list of data set variables using metadata and DATA step functions can be found in Program 12.4.2d.

**SEE ALSO:** Examples of SQL steps that create a series of macro variables can be found in Eddlestone (1997), Satchi (2000), and Casas (2002). First (2001b), as well as Rajcecki and Kahle (2000), both create a list of macro variables with a single SQL INTO clause.

## 6.2.4 Understanding Automatic SQL-Generated Macro Variables

Whenever PROC SQL is executed, a series of macro variables is generated and placed in the most local symbol table. The names of these macro variables all start with the letters SQL and include the following:

### **SQLOBS**

Number of rows processed by the Select statement

### **SQLLOOPS**

Number of iterations of the inner loop

### **SQLRC**

Step Return Code; 0 when the step is successful

### **SQLEXITCODE**

Contains the highest return code that occurred from some types of SQL insert failures

### **SQLXRC**

Return code from a DBMS when an SQL pass-through is used

### **SQLXMSG**

DBMS message generated by an SQL pass-through

### **SQLXOBS**

Undocumented, but seems to be associated with the number of observations associated with the pass-through (it is created even if there is no pass-through)

### **SYS\_SQL\_IP\_STMT**

This macro variable contains the SELECT statement, which is to be passed to the database in an implicit pass-through (this macro variable is created even if there is no pass-through)

#### Program 6.2.4: Showing SQL Automatic Macro Variables

```
proc sql noprint;
  select lname into :lastnames separated by ','
    from macro3.clinics(where=(lname='S'));
  %let numobs=&sqlobs;
quit;
```

Because this program was executed in open code, the macro variables were all written to the global symbol table. The SAS Log shows that there were five names that start with 'S'. Here is the result of the %PUT:

```
26  %put _user_;
GLOBAL SQLOBS 5
GLOBAL NUMOBS 5
GLOBAL SQLOOPS 26
GLOBAL SYS_SQL_IP_ALL -1
GLOBAL SYS_SQL_IP_STMT
GLOBAL SQLXOBS 0
GLOBAL SQLRC 0
GLOBAL SQLEXITCODE 0
GLOBAL LASTNAMES Smith,Simpson,Simpson,Stubs,Saunders
```

As you have probably noticed in a number of examples in Section 6.2, I tend to use &SQLOBS a lot in my programming work. Capturing the return codes of SQL steps that go wrong (&SQLRC and &SQLEXITCODE) has also been helpful.

## 6.3 Moving Text from Macro Variables into Code

Because the macro language is at its core a code generator, using the text stored in macro variables to write code is very straightforward. The timing issues discussed in Section 6.1 that prevent us from directly using the macro language to move information from data set variables into macro variables do not apply when going the other way. Since we are having the macro language write code (code that is written before it is parsed), there are no timing problems.

You can use any of several methods to convert information that is stored in macro variables to values that are to be stored in DATA step variables. Since macro references are resolved before the DATA step is either compiled or executed, moving a value that has already been established on a macro symbol table to a data set can successfully involve the use of macro references, and, consequently, this process is more direct than when writing a DATA step value to a macro variable. As was shown in Section 6.1, when writing DATA step values to the symbol table, you cannot use macro references such as &VAR or %LET statements.

Because you will assign a value to a variable on the Program Data Vector, the easiest and most common methods are through the use of either the assignment or RETAIN statements. In other situations, you might need to use either the SYMGET or RESOLVE functions.

---

### 6.3.1 Assignment and RETAIN Statements

You can use the DATA step assignment statement to create a DATA step variable or to assign a value to an existing variable. In Program 6.3.1a the macro variable &DSN is created and assigned the value clinics. Later in the program, &DSN is used to define the character variable DSET in the data set SUBWT.

#### Program 6.3.1a: Inserting Macro Variable Values into Code

```
%let dsn = clinics;
...code not shown...
data subwt;
  set clinics (keep=fname wt);
  dset = "&dsn";
  where wt>80;
run;
```

After the macro variable has been resolved, the resultant DATA step code shows that the constant of "clinics" will be assigned to the variable DSET.

```
data subwt;
  set clinics (keep=fname wt);
  dset = "clinics";
  where wt>80;
run;
```

Because in this example the variable DSET will contain a constant value, it is incrementally more efficient to assign its value through the RETAIN statement, which is a compilation time statement as opposed to the assignment statement, which is an executable statement.

#### Program 6.3.1b: Using RETAIN with a Macro Variable

```
data subwt;
  set clinics (keep=fname wt);
  retain dset "&dset";
  where wt>80;
run;
```

The assignment and RETAIN statements are the simplest methods and usually will be your primary methods for creating and assigning DATA step variables using symbol table values. The next most likely method, the SYMGET function, is covered in the next section.

**MORE INFORMATION:** A date constant is established using an assignment statement and the automatic macro variables &SYSDATE and &SYSTIME in Section 2.6.1.

### 6.3.2 SYMGET and SYMGETN Functions

The SYMGET function is not a macro function. Like the SYMPUTX routine, it is used in the DATA step and converts the current value of a macro variable to a character string so that it can be placed into a DATA step variable. Effectively SYMGET is the functional opposite of the SYMPUT routine. Almost always the techniques in Section 6.3.1 will be sufficient to move information from the symbol tables into your code. However, there are circumstances where something more is needed, and usually SYMGET will solve those types of situations.

The SYMGETN function was written to work with SCL programs and, although it seems to work in the DATA step, it was not designed to do so. Use caution if you use SYMGETN outside of SCL, because its use in the DATA step is unsupported.

The single argument of the SYMGET function is either a macro variable name, a DATA step variable that takes on a value that is a macro variable name, or a character expression that constructs a macro variable name.

#### SYNTAX:

```
variable = SYMGET(argument);
```

When using the SYMGET function remember that

- it is used within a DATA step
- a character string with a default length of 200 is returned
- the argument is enclosed in quotes to directly specify the name of a macro variable
- when the argument is written without quotes, it is assumed to be either a DATA step variable whose value is a macro variable name or an expression that constructs a macro variable name.

Using an assignment statement as opposed to the SYMGET function will have similar, but not necessarily exactly the same results. In Program 6.3.2a, the variables DATASET and DSET will have the same value, but not the same length.

#### Program 6.3.2a: Retrieving a Macro Variable by Using SYMGET

```
%let dsn = clinics;

data subwt;
  set &dsn(keep=fname wt);
  where wt>80;
  dataset=symget('dsn'); ①
  dset = "&dsn"; ②
run;
```

- ① Because its length is not otherwise specified, the variable DATASET will have a length of 200 (the default returned by SYMGET).
- ② Since &DSN is resolved before the statement is parsed, the variable DSET will have a length of 7.

In most cases you will not need to use the SYMGET function because it is simply easier to assign a value using either a RETAIN or assignment statement, as was done for DSET in the previous example.

In the following example, the data set CLINICS has a variable CODE that identifies an adjustment factor that is to be applied to the variable WT. The variable CODE can take on the values of 1, 2, or 3, and the three adjustment factors have already been loaded into three corresponding macro variables: &ADJ1, &ADJ2, and &ADJ3. One way to apply the adjustment factors would be to use a series of IF statements.

#### Program 6.3.2b: Resolving Macro Variables before Compilation of the DATA Step

```
data adjwt;
  set clinics;
  if code=1 then wtadj = wt*&adj1;
  else if code=2 then wtadj = wt*&adj2;
  else if code=3 then wtadj = wt*&adj3;
run;
```

You can simplify the coding structure of the DATA step by eliminating the IF-THEN/ELSE processing and, instead, do a table lookup. Here the SYMGET function is used to replace the IF-THEN/ELSE in the DATA step.

```
data adjwt;
  set clinics;
  wtadj = wt*symget(catt('adj',code));
run;
```

The SYMGET function always returns a character string, and this character string is converted to numeric (the conversion is done automatically by SAS) before the arithmetic operation of multiplication can be performed. This results in a note being written in the SAS Log.

```
NOTE: Character values have been converted to numeric values at the places
given by:
(Line):(Column).
10:9
```

This warning can be eliminated by using the INPUT function to convert the SYMGET results to numeric prior to performing the multiplication.

```
wtadj = wt*input(symget(catt('adj',code)),best12.);
```

Unlike SYMGET, which always returns a character string, the SYMGETN function returns a numeric value. By using the SYMGETN function, we can avoid using the INPUT function to force the conversion to numeric.

```
wtadj = wt*symgetn(catt('adj',code));
```

The SYMGETN function will return an error if the macro variable that it calls cannot be converted to a numeric value. Also, *please remember* that, although it generally seems to work, the use of SYMGETN is unsupported in the DATA step.

SYMGET is most often used when the macro variable is to be resolved at the time of DATA step execution. If you are resolving macro variables in compiled language elements such as a compiled DATA step, a VIEW, a SAS Component Language (SCL) program, or in a user-defined function (PROC FCMP), then SYMGET and SYMGETN become much more helpful, if not essential (see Appendix 2 Section A2.2.3 for more on the use of the macro language with these topics).

**MORE INFORMATION:** A series of SYMGET examples appear as part of a comparison with the RESOLVE function in Section 6.3.3. The SYMGET function is especially useful in SAS Component Language programs because those programs are compiled, and in compiled programs macro variables are not usually referred to using an ampersand. Appendix 2 discusses various issues associated with

macros and macro references in SAS Component Language and other compiled programs. Finally, a table lookup using formats is described in Section 9.4.

**SEE ALSO:** John Piet (2000) shows a good example of an application that uses SYMGET and DATA values to determine the macro variable of interest.

### 6.3.3 The RESOLVE Function

Your third choice for assigning symbol table values to the PDV is the DATA step's RESOLVE function. If you only rarely need to use SYMGET, then RESOLVE is needed even more rarely. Until you really need to use it, you probably only need to know that it exists, and basically what it does.

This function is very similar to the SYMGET function. Although the RESOLVE function tends to use more resources than SYMGET, it will also accept a wider variety of arguments. RESOLVE returns a character string, and will attempt to resolve some arguments that SYMGET will not attempt to resolve.

#### SYNTAX:

```
variable = RESOLVE(argument);
```

The argument to the RESOLVE function is passed to the macro processor for resolution and potentially even execution. When the argument is enclosed in single quotes, the string is not resolved when the DATA step is compiled, but instead is passed intact to the macro facility for resolution during DATA step execution. This means that the argument can contain direct macro references that use the & and % signs.

The arguments for RESOLVE are not interchangeable with those of SYMGET. The three types of arguments that RESOLVE will act on are as follows:

- DATA step variable names
- Text enclosed in single quotes
- Character expressions

Unlike with the SYMGET function, if the argument is an unquoted DATA step variable, the variable's value from the Program Data Vector (PDV) will be returned. If the resolved value from the PDV contains macro references (% or &), then macro facility resolution will take place during DATA step execution.

Text enclosed in single quotes is passed directly to the macro processor for resolution during DATA step execution. If the text does not contain macro references the value is passed back. Macro references are resolved.

A character expression is also passed to the macro processor. This expression can contain macro variable references and macro calls that include & and % signs.

Generally, when using RESOLVE with macro variables, the argument will have an & or a % either directly or indirectly, since otherwise the SYMGET function could be used. Because the argument in the RESOLVE function is passed directly to the macro facility for processing, any macro references are resolved or executed before the statement that contains the RESOLVE is executed. In Program 6.3.3a a macro variable is to be used in a DATA step VIEW. Without using the RESOLVE function the macro variable &WT would be resolved during the compilation of the VIEW. The RESOLVE function prevents resolution until the VIEW is executed.

#### Program 6.3.3a: Using RESOLVE in a VIEW

```
data partial/view=partial;
  set macro3.clinics;
  where wt lt input(resolve('&wt'), 6.);
  run;

%let wt=100;
proc print data=partial;
  var lname fname ht wt;
run;
```

The use of the INPUT function converts the character string returned by the RESOLVE function into a numeric value.

In this example, because we know the name of the macro variable and it is a constant, we could have also used the SYMGET function.

```
where wt lt input(symget('wt'), 6.);
```

In Program 6.3.3a either the SYMGET or RESOLVE function can be used because the argument is the name of a constant macro variable. When the macro variable is embedded in a DATA step variable, SYMGET cannot be used. In Program 6.3.3b a series of macro variables are created using SYMPUTX. Each of these macro variables are reused in the same DATA step and each forms a part of the name of a file. The filename along with the appropriate macro variable name is stored in the DATA step variable PREFIX. The RESOLVE function is used to force the resolution of the macro variable.

#### **Program 6.3.3b: Finalizing a Variable's Value by Using RESOLVE**

```
data list;
drop Yesterday;
if _n_ = 1 then do; /* create the macro vars */ ①
  Yesterday = today()-1; /* assumes this step run after midnight */
  call symputx('ddmmyy',put(Yesterday,ddmmyy6.));
  call symputx('yymmd',put(Yesterday,yymmd6.));
  call symputx('monyy' ,put(Yesterday,monyy5.));
  call symputx('monyyyy',put(Yesterday,monyyyy7.));
end; /* create the macro vars */
format prefix dsname $44.;
input prefix $; ②
dsname = resolve(prefix); ③
datalines;
RESOLVES.TO.YESTERDAY.D&ddmmyy ④
ALSO.RESOLVES.TO.YESTERDAY.A&yymmd
RESOLVES.TO.MONTH.AND.YEAR.&monyy
RESOLVES.TO.MONTH.AND.FOUR.DIGIT.YEAR.&monyyyy
run;
```

Source: Don Henderson, Henderson Consulting Services.

- ① Before any of the incoming data are read into the DATA step, a series of macro variables are created using yesterday's date and a specific format.
- ② The raw data are read into the variable PREFIX. Each value of PREFIX contains an unresolved macro variable reference.
- ③ The RESOLVE function causes the embedded macro variable references in PREFIX to be resolved.
- ④ Macro variable references in the DATALINES section of the DATA step are not resolved during the compilation phase. Nor are they resolved when the data are read using the INPUT statement ②.

**Figure 6.3.3b: Values Resolved during DATA Step Execution**

#### **Sample Substitution for File Names**

prefix	dsname
RESOLVES.TO.YESTERDAY.D&ddmmyy	RESOLVES.TO.YESTERDAY.D211214
ALSO.RESOLVES.TO.YESTERDAY.A&yymmd	ALSO.RESOLVES.TO.YESTERDAY.A141221
RESOLVES.TO.MONTH.AND.YEAR.&monyy	RESOLVES.TO.MONTH.AND.YEAR.DEC14
RESOLVES.TO.MONTH.AND.FOUR.DIGIT.YEAR.&monyy	RESOLVES.TO.MONTH.AND.FOUR.DIGIT.YEAR.DEC14

### 6.3.4 Comparison of the SYMGET and RESOLVE Functions

As has been mentioned in Sections 6.3.2 and 6.3.3, you will rarely need to use either the SYMGET function or even more rarely the RESOLVE function. Very often you will find examples of their usage where the application is less than optimal.

It is important to note that when returning a number, the values returned are not always the same. In Program 6.3.4a the text value 5 is stored in &TEST. This value is then returned four different ways.

**Program 6.3.4a: Retrieving a Numeric Value from a Macro Variable**

```
%let test=5;
data b;
  w = &test;
  x = symget('test');
  y = resolve('&test');
  run;
proc contents data=b;
  run;
proc print data=b;
  run;
```

A portion of the output from the CONTENTS procedure shows that the variables created do not have the same attributes.

**Figure 6.3.4a: Variable Attributes**

Alphabetic List of Variables and Attributes			
#	Variable	Type	Len
1	w	Num	8
2	x	Char	200
3	y	Char	200

Program 6.3.4b contains a single DATA step with several calls to the SYMGET and RESOLVE functions. The following excerpt from the SAS Log illustrates some of the differences between these two functions.

**Program 6.3.4b: SAS Log**

```
1
2           %let dsn = clinics;
3           %let clinics = Bethesda;
4
5           data demo;
6             dname = 'clinics';
7             amp   = '&dsn';
8             * unquoted arguments;
9             a1 = symget(dname);
10            a2 = resolve(dname);
11            a3 = symget(dsn);
12            a4 = resolve(dsn);
13            a5 = symget(amp);
14            a6 = resolve(amp);
15            put / a1= a2= a3= a4= a5= a6=;
16
17           * Using a single quote;
18           b1 = symget('dname');
19           b2 = resolve('dname');
20           b3 = symget('dsn');
21           b4 = resolve('dsn');
```

```

22          b5 = symget('amp');
23          b6 = resolve('amp');
24          put / b1= b2= b3= b4= b5= b6=;
25
26          * Single quote with &;
27          c1 = symget('&dsn');
28          c2 = resolve('&dsn');
29          c3 = symget('&amp');
30          c4 = resolve('&amp');
31          put / c1= c2= c3= c4=;
32
33          * Double quote with &;
34          d1 = symget("&dsn");
35          d2 = resolve("&dsn");
36          d3 = symget("&amp");
WARNING: Apparent symbolic reference AMP not resolved.
37          d4 = resolve("&amp");
WARNING: Apparent symbolic reference AMP not resolved.
38          put / d1= d2= d3= d4=;
39
40          * Quoted triple ampersand;
41          e1 = resolve('&&&dsn');
42          put / e1= ;
43          run;

NOTE: Numeric values have been converted to character values at the places
given by:
      (Line):(Column).
      11:24   12:25
NOTE: Variable dsn is uninitialized.
NOTE: Invalid argument to function SYMGET at line 11 column 17.
NOTE: Invalid argument to function SYMGET at line 13 column 17.

a1=Bethesda a2=clinics a3= a4=. a5= a6=clinics
NOTE: Invalid argument to function SYMGET at line 18 column 17.
NOTE: Invalid argument to function SYMGET at line 22 column 17.

b1= b2=dname b3=clinics b4=dsn b5= b6=amp
NOTE: Invalid argument to function SYMGET at line 27 column 17.
NOTE: Invalid argument to function SYMGET at line 29 column 17.
WARNING: Apparent symbolic reference AMP not resolved.

c1= c2=clinics c3= c4=&amp
NOTE: Invalid argument to function SYMGET at line 36 column 17.
WARNING: Apparent symbolic reference AMP not resolved.

d1=Bethesda d2=clinics d3= d4=&amp

e1=Bethesda
dname=clinics amp=&dsn a1=Bethesda a2=clinics a3= dsn=. a4=. a5=
a6=clinics b1= b2=dname
b3=clinics b4=dsn b5= b6=amp c1= c2=clinics c3= c4=&amp d1=Bethesda
d2=clinics d3= d4=&amp
e1=Bethesda _ERROR_=1 _N_=1

```

**Table 6.3.4b: Comparison of SYMGET and RESOLVE—DATA Step Variable as the Argument**

SAS Log Line No.	Function Call	Resultant Value
9	symget(dname)	A1='Bethesda'
		DNAME has the value 'clinics', which points to the macro variable of the same name.
10	resolve(dname)	A2='clinics'
		The value of DNAME is retrieved from the PDV. Its value contains no macro references.
11	symget(dsn)	A3=''
		Variable DSN is uninitialized. DSN is set to missing.
12	resolve(dsn)	A4=''
		The current value of DSN on the PDV is used, and as we just saw, this is missing.
13	symget(amp)	A5=''
		AMP has the value of '&DSN'. The & is not expected and causes an INVALID argument message.
14	resolve(amp)	A6='clinics'
		The value of amp '&DSN' is retrieved from the PDV, and is passed to the macro facility where it is resolved.

**Table 6.3.4b (Continued): Comparison of SYMGET and RESOLVE—Quoted Value as the Argument**

SAS Log Line No.	Function Call	Resultant Value
18	symget('dname')	B1=''
		The macro variable &DNAME does not exist.
19	resolve('dname')	B2='dname'
		The string contains no macro references and is passed back to the assignment statement.
20	symget('dsn')	B3='clinics'
		&DSN is a macro variable that has the value of 'clinics'.
21	resolve('dsn')	B4='dsn'
		'dsn' has no macro or data set references; it is only a string and is passed through.
22	symget('amp')	B5=''
		There is no macro variable named AMP and this causes an INVALID argument message.
23	resolve('amp')	B6='amp'
		'amp' has no macro or data set references; it is only a string and is passed through.

**Table 6.3.4b (Continued): Comparison of SYMGET and RESOLVE—Quoting an Argument Containing an &**

SAS Log Line No.	Function Call	Resultant Value
27	symget('&dsn')	C1=' '
	&dsn is not a valid macro variable name (the & is not anticipated) and results in a run-time INVALID argument message.	
28	resolve('&dsn')	C2='clinics'
	'&dsn' is passed to the macro facility where it is correctly resolved.	
29	symget('&amp')	C3=' '
	&AMP is not a valid macro variable name (the & is not anticipated) and results in a run-time INVALID argument message.	
30	resolve('&amp')	C4='&amp'
	'&amp' is passed to the macro facility, but there is no macro variable with this name and the warning 'Apparent symbolic reference AMP is not resolved' is issued at run time. The unresolved value is returned.	

**Table 6.3.4b (Continued): Comparison of SYMGET and RESOLVE—Double Quotes with an & in the Argument**

SAS Log Line No.	Function Call	Resolved Call	Resultant Value
34–37	In each statement the macro reference (&DSN and &AMP) is resolved before the DATA step is compiled. It is the resolved call that is executed by the DATA step.		
34	symget("&dsn")	symget("clinics")	D1='Bethesda'
	The value of the macro variable clinics is retrieved.		
35	resolve("&dsn")	resolve("clinics")	D2='clinics'
	'Clinics' has no macro or data set references; it is only a string and is passed through.		
36	symget("&amp")	symget("&amp")	D3=' '
	&amp is not a valid macro variable name and results in a compile-time INVALID argument message. At execution this results in a missing value.		
37	resolve("&amp")	resolve("&amp")	D4='&amp'
	'&amp' is passed to the macro facility, but there is no macro variable with this name and the warning 'Apparent symbolic reference AMP is not resolved' is issued at run time. The unresolved value is returned.		

**Table 6.3.4b (Continued): Comparison of SYMGET and RESOLVE—Argument with Triple Ampersands in Single Quotes**

SAS Log	Line No.	Function call	Resultant Value
	41	resolve('&&&dsn')	E1='Bethesda'
&&&DSN is passed to the macro facility where it is resolved, first to &CLINICS and then to Bethesda.			

**SEE ALSO:** *SAS Technical Report P-222, Changes and Enhancements to Base SAS Software*, Release 6.07 provides some of the earliest written documentation for the RESOLVE function (pp. 313–315). *SAS 9.4 Macro Language Reference, Fourth Edition* documents the RESOLVE function (Chapter 16). Whitlock (1998) gives a short introduction to the use of the RESOLVE function. The SYMGET and RESOLVE functions are discussed by Jaffee (1999).

### 6.3.5 Less-Than-Optimal Uses of SYMGET and RESOLVE

It has often been said that one cannot both create and use a macro variable in the same DATA step. As was shown in Program 6.3.3b, this is not actually true; by using some combinations of SYMPUTX, SYMGET, and RESOLVE, you actually can both create and use a macro variable in the same DATA step. The real question is why would you want to? While there are some legitimate reasons for doing so (see Program 6.3.3b and Section 9.3 on building macro hash tables), most attempts are made by programmers who do not have a clear understanding of the use of the PDV (see Program 6.3.5c at the end of this section).

Since both the SYMGET and the RESOLVE functions access the macro symbol table at DATA step execution, both can retrieve values that were just added with the SYMPUTX routine. In Program 6.2.2a an SQL step was used to build a macro variable containing a list of values. A similar step is used in Program 6.3.5a, which creates the macro variable &LASTNAMES.

#### Program 6.3.5a: Using SQL to Create a Concatenated List of Last Names

```
proc sql noprint;
  select lname
    into :lastnames separated by '|'
      from macro3.clinics;
quit;
```

Two alternative approaches to building this macro variable are shown in Programs 6.3.5b and 6.3.5c. Both require more coding and both have major disadvantages, some of which might not be immediately obvious. This approach will be the fastest of the three, and will not have some of the limitations of the other two. As a general rule, of the three approaches, this technique should be considered the ‘best practice’ approach.

Although a bit more coding is involved, it is also possible to create this same macro variable in a DATA step. In Program 6.3.5b the DATA step is used to build the same list of last names.

#### Program 6.3.5b: Concatenating a List of Values in the DATA Step

```
data _null_;
  set macro3.clinics(keep=lname) end=eof;
  length allnames $32767; ①
  retain allnames ' '; ②
  allnames = catx('|',allnames,lname); ③
  if eof then call symputx('lastnames',allnames); ④
run;
```

- ① A length is specified for the DATA step variable that will be used to collect the list of last names. The maximum size of a DATA step variable is 32K characters.

- ❷ This list is initialized to missing and is retained from one observation to the next.
- ❸ The CATX function is used to append each of the last names (LNAME) onto the growing list of names held by ALLNAMES.
- ❹ Once all the names have been accumulated the list is written to the macro variable &LASTNAMES.

This approach has a major disadvantage. Although a macro variable can hold 64K characters, a DATA step variable is limited to a maximum of 32K characters ❶. In the SQL step (Program 6.3.5a) the macro variable is built directly and would be capable of being assigned the full 64K.

In an attempt to overcome the limitation of the 32K size barrier, we might try to circumvent the use of the PDV altogether by accumulating the values directly into the macro variable. In Program 6.3.5c the SYMPUTX and SYMGET functions are used together.

#### **Program 6.3.5c: Using SYMGET and SYMPUTX to Concatenate a List of Values in the DATA Step**

```
%let lastnames=;
data _null_;
  set macro3.clinics(keep=lname);
  call symputx('lastnames', ❸
               catx('|', ❶
                     symget('lastnames') , ❷
                     lname); ❹
run;
```

- ❺ The SYMPUTX function is used to replace the existing value of &LASTNAMES with the new value being built by the CATX function (❻, ❼, and ❽).
- ❻ The ‘|’ character is being used as the name separator.
- ❼ SYMGET retrieves the current value of &LASTNAMES.
- ❽ The current value of LNAME is appended onto the growing list of last names that are held in the macro variable &LASTNAMES.

The DATA step shown in Program 6.3.5c has even more severe limitations over the DATA step in Program 6.3.5b. Although we are not creating the intermediate variable on the PDV, we are still subject to the 32K character limitation. The concatenation buffer used by the various CAT functions (including the concatenation operator ('||')) also has a 32K limitation. Also, the SYMPUTX, SYMGET, and RESOLVE functions have a fair amount of execution overhead. It is because of this overhead that in Program 6.3.5b the SYMPUTX ❻ is executed conditionally. In Program 6.3.5c both the SYMPUTX and the SYMGET will be executed for every observation.

**MORE INFORMATION:** A concatenated list of values can also be loaded into a macro variable through the use of a PROC SQL step; see Sections 6.2.2 and 11.4.1.

**SEE ALSO:** Ian Whitlock gives a nice comparison of the SYMGET and RESOLVE functions in Whitlock (1998).

---

## **6.4 Using Data to Control Program Flow**

Often a number of macro variables need to be defined in order to control a process or series of programs. Rather than pass the values into macros through the use of parameters, the control information can be stored in data such as SAS data sets or Microsoft Excel files. These data values can then be used to control a macro by creating macro variables using the tools described in Sections 6.1 and 6.2.

Control data sets, also known as metadata, are most often used when the number of macro variables is large or when there are a great number of macro calls. The techniques associated with the use of metadata files are collectively known as dynamic programming techniques. Because of their flexibility, these techniques

can offer the macro programmer a great many opportunities to write highly adaptable programs and macros.

Control files can be specially created for a particular task or sometimes the information needed can often be found in existing data sets, VIEWS, or many other locations. There are a number of sources of metadata in addition to those that are constructed by the user. Part III of this book (Chapters 11-13) is dedicated to dynamic programming techniques.

**MORE INFORMATION:** A number of sections in this book demonstrate the use of metadata to build macro variables. In addition to the following sections (6.4.1 and 6.4.2), look in Part III (Chapters 11 and 12).

**SEE ALSO:** Zhang, Chen, and Wong (2003) provide an overview of metadata within the context of a data warehouse.

## 6.4.1 Assigning Macro Variable Values

Before we can take advantage of stored information, we need to identify the information source, extract the appropriate information, and then write it to macro variables for later use.

Program 6.4.1a is a rather simplistic example that illustrates the steps that are needed to control a macro using values that are contained in a data set. Two macro variables, &DSN and &OBS, are to be assigned values using the DATA step variables DSNAME and NOBS that are stored in the data set CONTROL.

### Program 6.4.1a: Pulling Macro Variables from Data

```
%macro look;
  data _null_;
    set control;
    call symputx('dsn',dsname,'1');
    call symputx('obs',nobs,'1');
    run;
    title1 "Contents of &dsn";
    proc contents data=&dsn;
      run;
    title1 "First &obs Observations of &dsn";
    proc print data=&dsn (obs=&obs);
      run;
  %mend look;

data control;
  dsname='macro3.clinics';
  nobs=5;
  run;

%look
```

The DATA step creates a single observation data set containing the two variables of interest. These are then read by the DATA \_NULL\_ step in the macro %LOOK where their values are converted to the macro variables used in the PROC CONTENTS and PROC PRINT steps.

```
title1 "Contents of macro3.clinics";
proc contents data= macro3.clinics;
  run;
title1 "First 5 Observations of macro3.clinics";
proc print data= macro3.clinics (obs=5);
  run;
```

Of course, if there were only two parameters each taking on a single value, it would be much easier and certainly more straightforward to use macro parameters. These techniques become valuable as the number of macro variables or calls to the macro become large. They also allow us to present the analysis or

reporting based on what the data actually contains. Which in turn allows the reporting to automatically adjust with changes in the parameters.

Often, control parameters can be determined directly from the analysis data, thus reducing the need for a separate control data set. The macro %REGIONRPT in Program 6.4.1b produces a separate series of analyses for each value of the classification variable REGION. Each of the procedure steps could have used a BY statement, however, the overall report would have been collated by REGION within procedure rather than across procedures.

#### **Program 6.4.1b: Using the Analysis Data as Its Own Control File**

```
%macro RegionRpt;
  * Build a macro variable for each level of REGION;
  proc sql noprint;
    select distinct region
      into :reg1 - ❶
      from macro3.clinics;
  %let total = &sqlobs; ❷
  quit;

  * Separate analyses for each level of REGION;
  %do i=1 %to &total; ❸
    title1 "Region: &&reg&i"; ❹
    proc print data=macro3.clinics(where=(region="&&reg&i")) ❹
      obs=10;
      . . . . code not shown . . . .
    proc report data=macro3.clinics(where=(region="&&reg&i"))
      nowd;
      . . . . code not shown . . . .
    proc tabulate data=macro3.clinics(where=(region="&&reg&i"));
      . . . . code not shown . . . .
    %end;
  %mend regionrpt;
%regionrpt
```

The macro %REGIONRPT will execute the procedures within the %DO loop once for each unique value of REGION. The macro variable &TOTAL holds the total number of REGIONS, and is used to set up a macro %DO loop (see Section 5.3.2). &&REG&I acts as a macro vector that contains the various values of the variable REGION (Section 2.5.3 discusses the resolution of double ampersand macro variables).

- ❶ A series of macro variables of the form &REG1, &REG2, &REG3, and so on hold the unique values taken on by the variable REGION.
- ❷ The number of unique regions will be held in &TOTAL.
- ❸ A %DO loop is used to step through the &TOTAL regions one region at a time.
- ❹ The values of the individual regions are identified by using the indirect reference &&REG&I.

This is one form of what is known as list processing, with this form of macro variables being known as a vertical list. List processing is a key component of the dynamic programming techniques described in Part III of this book.

**MORE INFORMATION:** The solutions to Chapter 6 exercises 2 and 3, which revisit Program 6.4.1b, correct some efficiency deficiencies of the approach shown here.

**SEE ALSO:** Carpenter and Callahan (1988) contains two related macros that you can use to split data into operational subsets for further processing. Similar approaches are taken in more sophisticated examples by Blair (1998), and Talbott and Westerlund (1998). Wright (2001) uses a PROC FREQ to generate the control file. Pass (1998, 2000) also controls the program flow for BY groups by breaking up the data into a series of steps. His approach, however, does not require the use of the &&VAR&I construct.

### 6.4.2 Assigning Macro Variable Names as well as Values

Rather than creating a list with a common name and index value as was done in Program 6.4.1b, you will sometimes find it useful to assign the name of the macro variable, as well as its value.

When using data to create an indexed list of macro variables, you will typically have one column per list of macro variables with one indexed macro variable per observation. In the alternative technique shown in this section our control data has one observation per macro variable with two data set variables being needed to fully define each macro variable. The WORK.CONTROL data set will have one observation for each macro variable that is to be built.

**Figure 6.4.2a: Control Data Set**

VIEWTABLE: Work.Control		
	mvarname	value
1	dsn	macro3.clinics
2	obs	5

In Program 6.4.2a the DATA step variable MVARNAME contains the name of the macro variable, and VALUE has its intended value. Each observation in the CONTROL data set (there are two) will be used to define one macro variable.

Notice that only one SYMPUTX is used, there is no hard coding of macro variable names in the DATA \_NULL\_ step, and there are no quotes used in the first SYMPUTX argument.

#### Program 6.4.2a: Using Data to Both Name the Macro Variable and Assign a Value

```
%macro look;
  data _null_;
    set control;
    call symputx(mvarname,value,'l'); ①
  run;
  title "Contents of &dsn";
  proc contents data=&dsn;
  run;
  title2 "Showing the first &obs Observations";
  proc print data=&dsn (obs=&obs);
  run;
%mend look;

data control;
  mvarname='dsn'; ②
  value='macro3.clinics'; ③
  output;
  mvarname='obs'; ②
  value=5; ③
  output;
run;
%look
```

- ① The macro variable name as well as the value are both defined using data set variables.
- ② The macro variable name is stored in MVARNAME.
- ③ The macro variable value is stored in VALUE.

Effectively, we have issued two %LET statements except we have done it through the use of a data set.

```
%let dsn = macro3.clinics;
%let obs = 5;
```

Of course, the author of this version of %LOOK had to know the name of the macro variables that were to be created (&DSN and &OBS). The following revised %LOOK macro, which uses the same control data set, overcomes this limitation by passing in the name of the macro variables as macro parameters. Here the parameters &DAT and &CNT do not hold the final values, but instead hold the name of the macro variable that holds the final value. A triple ampersand reference is used so that the macro variable name can be resolved before attempting to resolve the value that it holds.

#### **Program 6.4.2b: Passing the Names of Macro Variables Rather Than the Values**

```
%macro look(dat=,cnt=);
  title "Contents of &&&dat";
  proc contents data=&&&dat;
  run;
  title2 "Showing the first &&&cnt Observations";
  proc print data=&&&dat (obs=&&&cnt);
  run;
%mend look;
%look(dat=dsn,cnt=obs)
```

When %LOOK is executed the macro reference &&&DAT is resolved through a series of steps.

**Table 6.4.2b: Showing the Resolution of a Triple Ampersand Macro Variable**

Action	Macro Variable Reference	Explanation
Unresolved macro variable reference	&&&DAT	The triple ampersand delays resolution.
First scanning pass	$\begin{matrix} \&\& \\ \downarrow & \downarrow \end{matrix}$	Adjacent ampersands are taken together and resolved into one.
Interpreted as	&	DSN
Second scanning pass	&DSN	This is the macro variable that holds the true value of interest.
Resolves to	MACRO3.CLINICS	The resolved value becomes the data set name.

Remember that when you see a macro variable with three ampersands, it is a macro variable that holds the name of a macro variable that resolves to the value of interest. This is true whether it is in the form of &&&VAR or &&&VAR&I.

An indirect table lookup takes place when two steps are required to retrieve a given piece of information. In these cases, you are often given a key or index that will enable you to look up information that you can in turn use to retrieve the information of interest. When the index is numeric, vertical lists of the form &&VAR&I are possible. However, when the index value is character, such as a name, numeric lists become problematic. This is the case in the next example, Program 6.4.2c, which uses the data to both name the macro variable and provide its value. We can use this type of macro variable list as a form of a hash table to perform a lookup operation.

**Program 6.4.2c: Using Data Set Variables to Name Macro Variables**

```

data _null_;
  infile cards truncover;
  input ticker $ @16 company $55.; ⑤
  call symputx(ticker,company,'1'); ⑥
  datalines;
    IBM      International Business Machines Corporation
    INTEL    Intel Corporation
    Microsoft Microsoft Corporation
    run;

data stocks;
  set sashelp.stocks(keep=stock date volume);
  length CorpName $55;
  CorpName = symget(stock); ⑦
  run;

```

- ⑤ A company nickname (TICKER) and company name (COMPANY) are read into a DATA step.
- ⑥ The company name is stored in a macro variable that has the name of its nickname. In this example, the macro variable &INTEL will hold the name of the company (Intel Corporation).
- ⑦ If we ever want to look up the company name using the nickname, all we need to do is retrieve the appropriate macro variable's value. In SASHELP STOCKS the variable STOCK holds the company nickname.

**MORE INFORMATION:** A list of named macro variables is used to form a simple hash table in Section 9.3.

**SEE ALSO:** Pahmer (2015) uses triple ampersand macro variables in a macro.

**6.5 Executing Macro Code Using CALL EXECUTE**

Ian Whitlock writes, “Like many powerful features in SAS, CALL EXECUTE is too good to ignore and yet may be dangerous for the unwary” (1997).

As was carefully pointed out in the diagram in Section 1.3, macro statements execute before the DATA step is compiled and certainly before it is executed. This is why you cannot use a %LET statement to assign a value of a DATA step variable to a macro variable. However, there are times, while we are executing a DATA step, when we do need to be able to control macro statement execution. The CALL EXECUTE routine allows us to pass code (with or without macro references) directly to the macro facility during the execution of the DATA step.

**SYNTAX:**

```
CALL EXECUTE(argument);
```

The argument to the CALL EXECUTE routine can contain constant text or macro instructions. The value of the argument is always passed to the macro facility. If it contains only text, without macro references, the code is placed in an input stack for execution immediately after the current step completes (a step

boundary is encountered). If the argument does include macro references, they are executed or resolved immediately (the execution of the DATA step is temporarily suspended).

**Table 6.5: Types of CALL EXECUTE Arguments**

What the Argument Contains	Action Taken
Text without any macro references: <pre>call execute('proc print data=temp;');</pre>	Text is added to the program stack for execution after the DATA step.
Unquoted macro references or macro references quoted with double quotes: <pre>call execute(%abc);</pre>	The macro references are executed during the DATA step's compilation phase. The resultant code becomes the argument for the CALL EXECUTE.
Macro references quoted with single quotes: <pre>call execute('%abc');</pre>	Text is passed to the macro processor, where it is immediately executed (the execution of the DATA step is suspended). Point of clarification: If the macro %ABC contains macro statements or macro references, these macro statements or macro references will also be executed immediately. Any remaining text will be added to the stack for later execution.
Macro references quoted and masked from the macro processor: <pre>call execute ('%nrstr(%abc)');</pre>	The macro reference is added to the stack where it will be resolved and executed after the DATA step terminates.

**MORE INFORMATION:** The CALL EXECUTE routine is discussed further in Sections 11.5 and A1.6.5

**SEE ALSO:** An introduction to the CALL EXECUTE routine can be found in Whitlock (1997). Another explanation of the CALL EXECUTE routine can be found in Heaton-Wright (2003). Jaffee (1999) has an example and explanation of the CALL EXECUTE routine, and Hamilton (2000) uses the CALL EXECUTE routine to display error conditions in the SAS Log. A number of examples that use CALL EXECUTE are shown in Rook and Yeh (2001). These include the delayed execution of operating system commands. Croonan and Theuwissen (2002) use a series of CALL EXECUTES to perform a merge, while Mounib and Satchi (2000) use CALL EXECUTE in an example on matched sampling. Chow (1999); Jin, Jin, and Wang (2003); and Jiang (2003) each include a CALL EXECUTE example. Xia and Birkenmaier (2001) and Tomb and Carter (2001) both build a CALL EXECUTE to call a macro. Finally, several approaches using CALL EXECUTE to solve the problem of renaming variables are presented by Viergever (2003).

### 6.5.1 Executing Non-Macro Code

During the execution of the DATA step, the value of the CALL EXECUTE argument is passed to the macro facility. When the argument contains no macro references, the code is placed in the program stack for execution at the end of the DATA step that called the CALL EXECUTE.

The data set MACRO3.DBDIR contains the names of a series of data sets that you would like to print. In Program 6.5.1 the CALL EXECUTE will cause a PROC PRINT step to be executed for each data set named by the variable DSN.

**Program 6.5.1: Executing Non-Macro Code Using CALL EXECUTE**

```
data _null_;
  set macro3.dbmdir;
  prc = 'proc print data='||dsn||';run;';
  call execute(prc);
run;
```

The SAS Log shows that a series of PROC PRINT steps have been generated and are executed immediately after the DATA \_NULL\_ step executes:

```
NOTE: CALL EXECUTE generated line.
1  + proc print data=DEMOG ;run;
2  + proc print data=MEDHIS ;run;
3  + proc print data=PHYSEXAM;run;
4  + proc print data=VITALS ;run;
```

Rather than construct a DATA set variable (PRC), as was done in Program 6.5.1, the CALL EXECUTE argument can be constructed directly:

```
call execute('proc print data='||dsn||';run;');
```

**SEE ALSO:** Hughes (2016b) uses CALL EXECUTE to execute a LIBNAME statement.

**6.5.2 Executing Macro Code**

As has been shown in numerous examples in this book, we already know that macro references are executed as soon as they are received by the macro facility. Sometimes we want to delay this execution, and often we do this by masking the macro references. One of the most common ways of masking or hiding macro references is by enclosing them in single quotes.

Rather than building the PROC PRINT step within the CALL EXECUTE argument as was done in Program 6.5.1, the PROC step could be placed within a macro with the DATA step name passed as a macro parameter. In Program 6.5.2a the macro %PRTDSN contains a simple PROC PRINT, and the data set name is provided by the DATA step variable DSN.

**Program 6.5.2a: Using CALL EXECUTE with a Macro Call**

```
%macro prtdsn(dsn=);
  proc print data=&dsn;
  run;
%mend prtdsn;

data _null_;
  set macro3.dbmdir;
  call execute('"%prtdsn(dsn='||dsn||')'");
run;
```

Because the %PRTDSN is enclosed in single quotes, the text is not seen as a macro reference when the DATA step is compiled. During execution of the DATA step, the argument of the CALL EXECUTE will take on a value such as, %prtdsn(DEMOG). When the value is passed to the macro facility, it will be correctly interpreted as a macro call and the macro will be immediately executed. However, since the macro generates non-macro text, this text will be added to the program stack for execution after the current DATA step terminates. The macro call %prtdsn(DEMOG) will add the following code to the program stack:

```
proc print data=DEMOG ;
run;
```

As in the previous example, when the argument contains macro references, they are executed or resolved immediately. However, if the macro does not generate any non-macro text, nothing is added to the stack. This is demonstrated by Program 6.5.2b, which uses CALL EXECUTE to call the macro %TIMING, which only has macro language elements and does not generate any non-macro code.

#### Program 6.5.2b: Immediate Execution of Macro Language Elements

```
%macro timing(dset=);
  %put in timing &=dset; ❸
%mend timing;
data _null_;
  set macro3.dbdir; ❶
  call execute('%timing(dset='||dsn||')'); ❷
  put dsn=; ❹
run;
```

The SAS Log shows that the CALL EXECUTE routine generates a call to the macro, which is executed immediately. This is evidenced by the result of the macro %PUT ❸ being written to the SAS Log before the result of the DATA step PUT statement ❹, which immediately follows the CALL EXECUTE:

```
in timing DSET=DEMOG ❸
dsn=DEMOG ❹
in timing DSET=MEDHIS ❸
dsn=MEDHIS ❹
in timing DSET=PHYSEXAM ❸
dsn=PHYSEXAM ❹
in timing DSET=VITALS ❸
dsn=VITALS ❹
```

- ❶ An observation is read containing a value for the variable DSN.
- ❷ A macro call is built as the argument for a CALL EXECUTE.
- ❸ The macro %TIMING is executed immediately. This includes the execution of the %PUT.
- ❹ The DATA step PUT statement is then executed.

As you might anticipate when both macro and non-macro statements exist, the macro statements are executed immediately, while the non-macro statements are stacked for execution later. Program 6.5.2c augments the %TIMING macro shown in Program 6.5.2b with a small DATA step. After execution the SAS Log shows that four DATA steps were generated and each was executed from the stack after the original DATA step had terminated.

#### Program 6.5.2c: Macro and Non-Macro Execution from a CALL EXECUTE

```
%macro timing2(dset=);
  %put In timing2 &=dset; ❻
  data silly; ❸
    x="&dset";
    put x=;
    run;
%mend timing2;
data _null_;
  set macro3.dbdir;
  call execute('%timing2(dset='||dsn||')'); ❼
  put dsn=; ❽
run;
```

A portion of the SAS Log shows the sequence of the executions:

```
In timing2 DSET=DEMOG ⑥
dsn=DEMOG ⑦
In timing2 DSET=MEDHIS ⑥
dsn=MEDHIS ⑦
In timing2 DSET=PHYSEXAM ⑥
dsn=PHYSEXAM ⑦
In timing2 DSET=VITALS ⑥
dsn=VITALS ⑦
NOTE: There were 4 observations read from the data set MACRO3.DBDIR.
NOTE: DATA statement used (Total process time):
      real time          0.01 seconds
      cpu time          0.01 seconds

NOTE: CALL EXECUTE generated line.
1  + data silly;           x="DEMOG";           put x=;           run; ⑧
x=DEMOG
NOTE: The data set WORK.SILLY has 1 observations and 1 variables.
NOTE: DATA statement used (Total process time):
      real time          0.01 seconds
      cpu time          0.01 seconds

2  + data silly;           x="MEDHIS";           put x=;           run; ⑧
x=MEDHIS
. . . Portions of the SAS Log are not shown . . . .
```

- ⑤ CALL EXECUTE is used to call the macro %TIMING2.
- ⑥ The %PUT in %TIMING2 is executed immediately.
- ⑦ The PUT in the DATA step is executed immediately after the CALL EXECUTE
- ⑧ The four DATA steps generated by %TIMING2 through the CALL EXECUTE are executed from the stack after the completion of the first DATA step.

**SEE ALSO:** Sisson (2016) uses the CALL EXECUTE routine to execute a macro that creates directories. Wilson (2015), Hughes (2016b), and Cheng *et al* (2016) each use CALL EXECUTE to execute a macro call.

### 6.5.3 Addressing Timing Issues

One of the things that make CALL EXECUTE problematic for many of the folks that use it, is the timing of events—what takes place and when.

Remember that the CALL EXECUTE routine resides in a DATA step that will be compiled, and that any macro references will be resolved during this compilation. Consequently, in the CALL EXECUTE statement in Program 6.5.3a, the macro %DOIT, which is unquoted, will execute BEFORE the DATA step is compiled.

**Program 6.5.3a: Execution of Macro Language Elements during DATA Step Compilation**

```
%macro doit;
  proc print data=a; run;
%mend doit;

data a;
  set macro3.dbdir;
  call execute(%doit);
run;
```

Because %DOIT executes before the DATA step is compiled, it can write code into the DATA step—in this case into the CALL EXECUTE:

```
data a;
  set datamgt.dbdir;
  call execute( proc print data=a; run; );
run;
```

The resultant CALL EXECUTE will fail because the argument of the CALL EXECUTE will be misinterpreted. The CALL EXECUTE routine can have only one argument and the variables PROC, PRINT, and so on, do not exist. As in the examples in Section 6.5.2, we prevent macro resolution during *DATA step compilation* by using single quotes. Restating the statement with single quotes will cause it to execute correctly:

```
call execute('proc print data=a;run;');
```

Or better yet:

```
call execute('%doit');
```

You will also need to be careful when using CALL SYMPUTX and the SYMGET function when accessing macro variables that are used in the same DATA step that uses CALL EXECUTE, and even in any code that is executed by CALL EXECUTE. As always the key will reside in the timing of the execution of the various statements.

The CALL EXECUTE in Program 6.5.3b further illustrates this point when it calls the macro %TEST. In it we correctly mask the macro call in the CALL EXECUTE routine, however still a couple of things happen that might not be anticipated.

**Program 6.5.3b: Testing Timing When Using CALL EXECUTE**

```
%macro test;
data _null_;
  put 'Calling SYMPUTX'; ①
  call symputx('x3',100); ②
  run;

%put Ready to compile DATA step for NEW; ③
data new;
  %put Compiling NEW; ④
  put 'Executing NEW'; ⑤
  y = &x3; ⑥
  run;
  title 'Data NEW';
  proc print data=new;
    run;
%mend test;

data _null_;
  call execute('%test'); ⑦
  run;
```

The macro %TEST contains two DATA steps, two macro %PUT statements, and a macro variable, which needs to be resolved. In the first DATA step a CALL SYMPUTX creates the macro variable &X3, and in the second DATA step this same macro variable is to be used. In this particular example it would be silly to use CALL EXECUTE just to call the macro %TEST, but using CALL EXECUTE enables us to further examine the timing of events when we do use CALL EXECUTE. Even on a simple example such as this one, you can see how the use of CALL EXECUTE changes event timing. If we were to apply our non-CALL EXECUTE experience to this macro or if we were to execute %TEST without using CALL EXECUTE, we might expect the following steps to occur in the order that they are numbered, however this is *not* what happens.

- ➊ During DATA step execution, write a note to the SAS Log using a PUT statement to indicate that the CALL SYMPUTX ➋ is being executed.
- ➋ Create the macro variable &X3.
- ➌ The %PUT statement indicates that the second DATA step is about to be compiled and executed.
- ➍ This %PUT, which follows the DATA statement is used to indicate that the compilation of the DATA step has begun. The macro variable &X3 should be resolved when it is encountered.
- ➎ The PUT statement will write to the SAS Log when the execution of the DATA NEW step has begun.
- ➏ Execute the assignment statement (y=100;) during DATA step execution.

When %TEST is executed by the CALL EXECUTE ➐, this is **not** the sequence of events and the results are not completely as we anticipated. The following SAS Log is generated. The numbered items in the SAS Log correspond to the numbered lines in the CALL EXECUTE.

```
...portion of SAS Log not shown...

Ready to compile DATA step for NEW ➃
Compiling NEW ➄
WARNING: Apparent symbolic reference X3 not resolved. ➆
NOTE: DATA statement used (Total process time):
      real time          0.00 seconds
      cpu time          0.00 seconds

NOTE: CALL EXECUTE generated line.
1  + data _null_;      put 'Calling SYMPUTX';      call symputx('x3',100);
run;

Calling SYMPUTX ➊ ➋
NOTE: DATA statement used (Total process time):
      real time          0.00 seconds
      cpu time          0.00 seconds

1  +
data new;
put 'Executing NEW';      y = 100; ➏      run;

Executing NEW ➎
NOTE: The data set WORK.NEW has 1 observations and 1 variables.
NOTE: DATA statement used (Total process time):
      real time          0.02 seconds
      cpu time          0.01 seconds

1  +
title 'Data NEW';
1  +
proc print data=new;      run;
```

```
NOTE: There were 1 observations read from the data set WORK.NEW.
NOTE: PROCEDURE PRINT used (Total process time):
      real time          0.02 seconds
      cpu time          0.00 seconds
```

- 7 Because %TEST is in single quotes, the macro call is immediately executed along with any macro references contained in the macro. The resulting and remaining text is placed on the input stack. It is this code that is then executed after the DATA step containing the CALL EXECUTE has executed.

As was shown in Program 6.5.2c, when %TEST is executed, if there are any macro statements or macro variables, they will immediately execute or resolve or both (they are not placed in the stack). This execution and resolution takes place *even across step boundaries*. This is a departure from code that is otherwise executed (without using CALL EXECUTE). Normally we would expect the first DATA step to be executed before the second is even compiled. We expect the macro variable &X3 to be created (❷) in the first step before it is needed in the second step (❶).

Instead, the SAS Log shows that the two %PUT statements are executed immediately ❸, ❹, and that an attempt is made to resolve &X3 ❺ before it has been created (this gives us the unresolved macro reference warning).

Once the macro statements have been executed, the two DATA steps execute. The macro variable is created ❻, ❼ and then used ❾, ❿. This time, when &X3 is needed, it exists and the macro variable resolves as it should (we got the value of &X3 that we wanted, but not in the way that we anticipated).

The timing issues described in Program 6.5.3b are a result of the immediate execution of the macro language elements of %TEST. The single quotes hide the macro call during DATA step compilation, but not during DATA step execution. If we instead use the %NRSTR quoting function (see Section 7.1 for more on quoting functions), CALL EXECUTE will write the macro call to the stack. This delays the execution of all elements of the macro until after the DATA step has completed execution. The DATA step becomes the following:

```
data _null_;
  call execute('%nrstr(%test)');
run;
```

You can also avoid the problems described in Program 6.5.3b if you execute the complete %TEST macro immediately through the use of the DOSUBL function (see Section 8.5.1). This function suspends the execution of the DATA step and immediately executes its argument, in this case the macro call for %TEST:

```
data _null_;
  rc=dosubl('%test');
run;
```

**MORE INFORMATION:** Section 11.5 discusses examples of the CALL EXECUTE in dynamic programming situations. The DOSUBL function, which is described in Section 8.5.1, is related to the CALL EXECUTE routine.

**SEE ALSO:** Timing issues associated with CALL EXECUTE are discussed in more detail in *SAS 9.4 Macro Language: Reference, Fourth Edition* (Chapter 8 pp. 103–105).

## 6.6 Testing Your Knowledge with Chapter Exercises

1. The PROC SQL step shown in Program 6.6.1 creates two macro variables &MWT and &MWT1; how do their stored values differ?

### Program 6.6.1: Storing Numeric Values in a Macro Variable Using PROC SQL

```
proc sql noprint;
  select min(weight)
    into :mwt
    from sashelp.class;
  select min(weight)
    into :mwt1 -
    from sashelp.class;
quit;
%put |&mwt|;
%put |&mwt1|;
```

2. The macro %REGIONRPT in Program 6.4.1b uses a WHERE statement to subset the MACRO3.CLINICS data set for each procedure. Rewrite the macro so that a separate data set is created for each region and then use these individual data sets for each procedure inside of the %DO loop, in other words eliminate the use of the WHERE statement. The most efficient program will read MACRO3.CLINICS just once or twice, creating ten data sets. (Hint: You will need to add a DATA step. Consider writing the DATA step without macros first, and then convert it to use the list of macro variables.)

```
data reg1 reg2.....;
  set macro3.clinics;
  if region='1' then output reg1;
  else if region='2' then output reg2;
  else if.....
  ...
run;
```

3. The suggested solution for Question 2 (Program A1.6.2) assumes that the variable REGION takes on the values of '1' through '10'. This is true, however, the solution would be more robust if it would also work for a region with a designation of 'AQ7' or something like that. Modify your coded solution for Question 2 so that the data set names reflect the value of the variable REGION in that data set. Remove any assumptions about the values of REGION ranging from '1' to '10'.
4. *Extra credit:* Give this exercise a try after you are comfortable with the techniques required by Exercises 2 and 3.
  - a. The SAS data set MACRO3.BIOMASS has both numeric and character variables. Create a macro that will convert all the numeric variables to character. If a variable is associated with a format (for example, DATE7.), use it in the conversion. All character variables should be passed through to the new data set and the variable names in the new data set should be the same as in the old data set.
  - b. Create a macro that is general enough to operate on any SAS data set.

**HINT 1:** The following DATA step will convert the variable AA from numeric to character.

```
data t2 (drop=__aa);
length aa $8;
set t1(rename=(aa=__aa));
aa = left(put(__aa,best9.));
run;
```

**HINT 2:** The following PROC CONTENTS will create the data set CONT, which contains the names (NAME) and type (TYPE {numeric=1 and character=2}) of the variables in the data set T1.

```
proc contents data=t1 out=cont nopolish;
run;
```

5. Use CALL EXECUTE to execute the %SPLIST macro once for each species stored in WORK.SPLIST, which is created in Program 6.6.5.

#### Program 6.6.5: Creating a Summary for Each Species

```
data splist;
  infile datalines truncover;
  input species $9.;
  datalines;
Parkki
Perch
Pike
run;
%macro splist(sp=);
title1 "Average Weight of &sp";
proc means data=sashelp.fish n mean;
  where species=&sp;
  var weight;
  run;
%mend splist;
```

# **Chapter 7: Using Macro Functions**

<b>7.1 Quoting Functions .....</b>	<b>132</b>
7.1.1 Using the %BQUOTE Function.....	135
7.1.2 %STR .....	137
7.1.3 Considerations When Quoting.....	137
7.1.4 Basic Types of Quoting Functions and Why We Care .....	142
7.1.5 A Bit about the %QUOTE and %NRQUOTE Functions.....	145
7.1.6 Removing Masking Characters .....	145
7.1.7 The %SUPERQ Quoting Function .....	146
7.1.8 Quoting Function Summary .....	148
7.1.9 Quoting Mismatched Symbols with the %STR and %QUOTE Functions .....	149
<b>7.2 Text Functions.....</b>	<b>150</b>
7.2.1 %INDEX .....	152
7.2.2 %LENGTH.....	153
7.2.3 %SCAN and %QSCAN .....	154
7.2.4 %SUBSTR and %QSUBSTR .....	157
7.2.5 %UPCASE and %QUPCASE .....	158
7.2.6 %LEFT and %QLEFT .....	159
7.2.7 %LOWCASE and %QLOWCASE .....	160
7.2.8 %TRIM and %QTRIM .....	161
<b>7.3 Evaluation Functions .....</b>	<b>162</b>
7.3.1 Explicit Use of %EVAL.....	162
7.3.2 Implicit Use of %EVAL.....	164
7.3.3 Using %SYSEVALF .....	166
<b>7.4 Using DATA Step Functions and Routines .....</b>	<b>169</b>
7.4.1 Using %SYSCALL .....	169
7.4.2 Using %SYSFUNC and %QSYSFUNC.....	170
7.4.3 Taking Advantage of Less Commonly Used DATA Step Functions .....	173
<b>7.5 Building Your Own Macro Functions .....</b>	<b>176</b>
7.5.1 Introduction .....	176
7.5.2 Building the Function.....	177
7.5.3 Using the Function.....	180
7.5.4 Returning a Value.....	181
<b>7.6 Other Useful User-Written Macro Functions.....</b>	<b>182</b>
7.6.1 One-Liners .....	182
7.6.2 Macro Functions with Logic .....	187
7.6.3 Functions for the DATA Step .....	190
<b>7.7 Testing Your Knowledge with Chapter Exercises .....</b>	<b>193</b>

Macro functions are similar to DATA step functions except that they operate on text strings and macro variables rather than character strings and data set variables.

Macro functions either change or provide information about the text strings that are the arguments. The following are general categories of macro functions:

#### **Quoting functions**

mask or remove meaning from special characters

#### **Text functions**

return information about text strings

#### **Evaluation functions**

perform arithmetic operations

#### **Bridging functions**

provide access to DATA step functions and routines

In addition to the many functions that are a part of the macro language, you can define your own very specific functions as well.

For the examples in this chapter and indeed for all of the code examples throughout the book, if you want to execute these sample programs, then be sure to follow the setup instructions. Remember that all of the data sets and programs are available for download, so you do not need to retype either the code or the data. For instructions on accessing and setting up the programs and data, see the “Example Code and Data” section within this edition’s “About This Book” front matter.

**MORE INFORMATION:** Section 7.6 discusses how to create your own macro functions. Several of the SAS autocall macros also behave like macro functions. See Chapter 10, “Building and Using Macro Libraries,” for details and examples.

**SEE ALSO:** Ian Whitlock has written a great deal about the macro language, and his paper on macro quoting (Whitlock, 2003a) and gives a very nice in-depth explanation of the quoting process.

A summary of some of the more commonly used macro functions can be found in Carpenter (2000a).

Once you become familiar with the macro language, it is fairly easy to write macros that behave like macro functions. See Whitaker (1989) and Carpenter (2002) for more on creating your own macro functions.

## **7.1 Quoting Functions**

Quoting functions are used to mask the meaning of words or characters that would otherwise be misinterpreted during the compilation or execution of the macro statement. This is a complex topic that is often only partially understood, even by some of the best macro programmers. Complex or not, all macro programmers should have at least a passing understanding of quoting in the macro language.

### **Why We Need to Quote**

Sometimes it is obvious what needs to be quoted and other times it will be much less obvious. In the following example we want to create a macro variable &P that contains three SAS statements that make up a PROC PRINT step. We want the shaded code to be stored in &P.

```
%let dsn = clinics;
%let p=proc print data=&dsn; var ht wt; run;;
```

In fact, this is not what is stored in &P. Because the first semicolon (the semicolon following &DSN) will be interpreted as the terminator of the %LET statement, the macro variable &P will contain only the PROC statement (without the semicolon).

```
proc print data=clinics
```

Not only will &P not contain what the programmer expects, the VAR statement will almost certainly cause a syntax error (the VAR HT WT; RUN; ; will be passed back to the base language for further processing). Clearly, we intend for the semicolons in the PROC PRINT step to be interpreted differently from the semicolon that terminates the %LET statement, however there is actually only one type of semicolon on our keyboard. We need to let SAS know which semicolon to use to terminate the %LET. In other words, we need to mask the special nature of all of the semicolons except the last one. This masking is done with quoting functions.

## What Macro Quoting Does

The semicolon, of course, is not the only character that can cause problems. The tables in Section 7.1.8 give a more complete list of the characters and combinations of characters that might need quoting.

Generally speaking, these are characters or words that can have special meaning or receive special attention when the code is parsed. I like to consider these characters as having “attitude.” Like some teenagers that display radical personality shifts as their hormones fluctuate, these characters sometimes demonstrate “attitude” by being misinterpreted. In the example above, the semicolon used to close or terminate the PROC PRINT statement is misinterpreted (from our perspective); the semicolon has attitude.

Quoting functions are used when we need to mask or delay the attitude of these characters. The quoting functions do this by marking the string contained in the quoting function and then physically placing invisible special non-printable characters on either side of the character that is to be masked. You rarely see any indication that these characters even exist (Section 7.1.3 shows you how to see them if you need to do so), and you usually do not need to worry about them. Once they are placed around a character, they remain unless removed (Section 7.1.6 on %UNQUOTE discusses how to remove the marks).

## Simple Examples of Quoting Functions

The following overview highlights some simple examples that use quoting functions. For the following examples assume that the macro variable &DSN has been defined as shown here.

```
%let dsn=clinics;
```

### Without a Quoting Function

**Example:**   %let p=proc print data=&dsn; run;;  
**&P:**           proc print data=clinics

Syntax errors are likely as &P contains the incorrect text.

### %STR

**Example:**   %let p=%str(proc print data=&dsn; run;);  
**&P:**           proc print data=clinics; run;  
&P contains the correct text.

### %NRSTR

**Example:**   %let p=%nrstr(proc print data=&dsn; run;);  
**&P:**           proc print data=&dsn; run;  
&DSN is not resolved even when the PROC PRINT statement is executed.

### %UNQUOTE

**Example:**   %let p=%nrstr(proc print data=&dsn; run;);

**&P:**           proc print data=&dsn; run;

**Apply the %UNQUOTE function to &P:**

              %unquote(&p)

**Resolves to:** proc print data=clinics; run;

The macro variable &DSN is resolved and the resolved unquoted value of &P contains the correct text.

## The History (or Lineage) of Quoting Functions

In the beginning of quoting there was the %STR function. Since this function has been around the longest and perhaps since it has the shortest name, it is probably the most commonly used quoting function.

However, it does not provide all the quoting capabilities that we need. To enhance the capabilities of the %STR function, it was augmented with a special masking character (%) that enables it to quote some special characters (see Section 7.1.9) that it does not otherwise quote.

These additions to the %STR function were not enough, so the %QUOTE function was added to the macro language to handle those situations that required quoting during macro execution. However, %QUOTE lacks the capability of directly quoting the symbols that come in pairs, so the family of quoting functions was still not complete.

The shortcomings of the %QUOTE were made up for with the inclusion of the blind quoting function, %BQUOTE. This function is designed to quote a wide range of characters, including those that can become mismatched. This function essentially eliminated the need for %QUOTE, which is now rarely used.

The NR versions of %STR, %QUOTE, and %BQUOTE were included to enable quoting in those situations when you want to control how the & and % signs in the quoted text are to be handled.

When the text contains macro references that you *do not* want resolved, the %SUPERQ function can be used. This is the strongest of the quoting functions; however, because of how it is applied (see Section 7.1.7), the advantage of %SUPERQ over %NRBQUOTE is minimal.

## Quoting Functions

The following list gives a brief overview of some of the functions and their behavior. More complete descriptions are given in the sections that follow.

**Table 7.1: Quoting Function Overview**

Function	Description
%BQUOTE	Removes meaning from unanticipated special characters (except & and %) that are resolved during execution
%NRBQUOTE	Removes meaning from unanticipated special characters (including & and %) during execution (one attempt is made to resolve macro references)
%STR	Removes meaning from special characters (except % and &) during compilation
%NRSTR	Removes meaning from special characters (including % and &) at compilation and prevents resolution of macro references
%SUPERQ	Prevents any resolution of the value of a macro variable
%UNQUOTE	Undoes quoting

It is not necessary to have a complete understanding of each of these functions, but rather a general understanding of what they do. You will use some of these functions much more often than others. It has been this author's experience that the %BQUOTE and %NRSTR functions are the most useful, however

for many macro programs the %STR, %NRSTR, and %SUPERQ are the most commonly used. %SUPERQ can be especially useful because it masks all special characters.

Also, with the availability of %BQUOTE and %NRBQUOTE, it is unlikely that you will find a use for the now dated %QUOTE and %NRQUOTE functions.

**MORE INFORMATION:** Although not considered quoting functions *per se*, a number of macro functions that return text values can also be used to mask special characters in the value returned by the function. These functions include, %QLEFT, %QUPCASE, %QSUBSTR, and %QSYSFUNC. See Section 7.2 for descriptions of these functions.

**SEE ALSO:** Rosenbloom and Carpenter (2011) contrast various quoting functions in situations with resolved special characters. Howell (2015) gives a nice summary and overview of macro quoting.

Bercov (1993) contains an easy-to-read summary of macro quoting, and First (2001a) gives a brief overview of the topic. Another overview with a number of examples is given by Burlew (2014, Chapter 14).

A very detailed and understandable explanation of macro quoting functions is given by O'Connor (1999). O'Connor was the lead developer of the macro language at SAS for a number of years, and her insights are invaluable.

Whitlock (2003a) has written a great deal about the macro language and his paper on macro quoting gives a very nice in-depth explanation of the quoting process.

You can find detailed explanations of macro quoting as well as examples that contrast compilation versus execution time quoting functions in Chapter 7, “Macro Quoting,” in the *SAS 9.4 Macro Language Reference, Fourth Edition*.

Carpenter (1999a) discusses various aspects of quoting functions and includes additional examples.

Although there really is nothing ‘easy’ about the topic, Russ Tyndall has a nice summary of macro quoting in his 2014 blog entry “Macro Quoting Made Easy”

<http://blogs.sas.com/content/sgf/2014/08/15/macro-quoting-made-easy/>

### 7.1.1 Using the %BQUOTE Function

This probably ought to be the first quoting function that you reach for when a quoting need arises. Also called the blind quote, this function is generally used to remove meaning from unanticipated characters in resolved text during macro execution. It is especially useful if the text string was entered by a user through an application, and you might not have been able to trap all possible characters that might cause the macro code to fail.

Assume that the macro variable &METHOD has been assigned the following value:

THE DOCTOR'S NEW THERAPY
--------------------------

Because there is an unmatched single quote ('), it is very likely that when &METHOD is resolved SAS would produce an error. This is demonstrated with a simple %LET statement.

%let method2 = &method; ①
---------------------------

- ① When the %LET is executed the &METHOD macro variable will be resolved before the assignment can be made. Essentially the %LET contains a single quote on the right side of the equal sign.

%let method2 = THE DOCTOR'S NEW THERAPY; ①
--

The mismatched quote will mask the semicolon that closes the %LET and will almost certainly create syntax problems. Because the %LET statement will not be completed until sometime after the next single quote is found, the resulting syntax errors will point to mismatched quotes in statements other than the %LET.

To get around this problem, you could use %BQUOTE to mask the single quote in the resolved value of &METHOD.

```
%let method2=%BQUOTE (&method);
```

The special meaning will be removed from the single quote when &METHOD is resolved, and it will, therefore, not cause syntax problems.

Sometimes the programmer is surprised by what may need to be quoted. It is important to consider the text as it stands at each step (compilation and execution). The following macro creates a DROP statement that can then be inserted into a DATA step.

#### Program 7.1.1a: Use of the %BQUOTE Quoting Function

```
%macro drop(droplist);
  %if %bquote(&droplist) ne %then ❷
    %bquote(drop &droplist;); ❸
  %mend drop;
%drop(x1-x3 lastname frstname ssn) ❹
```

- ❷ You must use a quoting function in case the user passes a string into the macro that would result in an invalid comparison. This might include multiple items, an item that could be confused with a logical operator (AND, OR, or NOT), or even a variable list that is seen as an arithmetic operation (x1–x3).
- ❸ The DROP statement itself must be quoted as well so that the semicolons will be interpreted correctly. In this statement the semicolon associated with the DROP statement is masked, while the second and final semicolon terminates the %IF statement.
- ❹ This call to %DROP would generate the following DROP statement.

```
drop x1-x3 lastname frstname ssn;
```

Without the %BQUOTE in macro %DROP ❷, &DROPLIST would be resolved before the %IF expression was evaluated. If, as in these examples, the resulting string contains any of the special symbols mentioned above, the macro %IF expression could become invalid. For this reason the version of %DROP in Program 7.1.1b will not work correctly.

#### Program 7.1.1b: Without the %BQUOTE the Macro Is Vulnerable to Failure

```
%macro drop(droplist=);
  %if &droplist ne %then ❸
    %bquote(drop &droplist;);
  %mend drop;
%drop(droplist=x1-x12)
```

- ❸ The dash (–) in the list has special meaning in the expression, and the resulting %IF expression will not be syntactically correct.

```
%if x1-x12 ne %then ....
```

The dash is interpreted as a minus sign, and since there is an implied arithmetic operation, %EVAL is called. Of course, since X1 and X12 are not integers, %EVAL cannot perform the subtraction, and an error stating, in part, that a “character operand was found in the %EVAL function” is written to the SAS Log. In the first version of %DROP the %BQUOTE is masking this dash and it is not misinterpreted as a subtraction symbol.

**MORE INFORMATION:** The %BQUOTE function is used to mask single quotes in the macro %DB2DATE (Section 7.6.1e) and in Programs 9.8.2 and 9.8.4.

## 7.1.2 %STR

This is likely the most commonly used quoting function. Popularity, however, does not necessarily make it the best quoting function. Probably its popularity is due to force of habit (it was the first quoting function to be introduced into the macro language), and because it is easy to type on the QWERTY keyboard. It removes meaning from most special characters (except % and &) at compilation and is most commonly used to remove meaning from commas and semicolons, and to preserve blanks and null spaces. When the percent sign (%) is used as a special masking symbol (see Section 7.1.9), the %STR function can also handle characters that normally come in pairs, such as parentheses and mismatched quotes.

In Program 5.2.1 the following %IF statement is used to compare the value of the macro variable &VLIST to a null value.

```
%if &vlist ne %then %sortit(dset=&indata, bylist=&vlist);
```

While this coding is both acceptable and common, it makes some programmers uncomfortable to not have anything on the right side of the equal sign. The %STR function can be used in this situation as a place holder.

```
%if &vlist ne %str() %then %sortit(dset=&indata, bylist=&vlist);
```

In this case, since the comparison is to be made against a null value, there is no space between the parentheses in the %STR function. Of course, the %BQUOTE function could also have been used here.

The %STR and %BQUOTE functions are not interchangeable. %BQUOTE is designed to mask characters in resolved code (during macro execution—see Section 7.1.4), while %STR masks characters during the compilation phase. Although they do not mask exactly the same list of characters, in open code these two functions will often behave similarly when masking constant text. However, because macros are compiled, inside of a macro definition, the %BQUOTE function does not mask constant text the same way as it does in open code. Be careful. The general rule of thumb is that %STR should be used to mask special characters in constant text (see Program 7.1.3f) and %BQUOTE when masking values in resolved text (see Program 7.1.3e).

**MORE INFORMATION:** The %STR function is used in the example that appears in Section 7.2.3 to preserve both a blank and a null value.

**SEE ALSO:** Chung and King (2009) show various ways to make comparisons with a null value and they discuss which are the most robust.

## 7.1.3 Considerations When Quoting

Because the topic of macro quoting is not particularly straightforward, there are some things that you should be thinking about when you get in a programming situation that requires the use of one of the quoting functions.

### Programming to Avoid Quoting

As in the cartoons where the characters literally paint themselves into a corner, it is not unusual for macro programmers to use quoting functions only after painting themselves into a coding corner. Very often a little programming forethought can avoid the need for the use of a quoting solution.

The macro %EXIST in Program 7.1.3a, is loosely based on a macro of the same name found in some early documentation on SAS macro processing. It creates the global macro variable &EXIST that takes on the

values of YES or NO, depending on whether the stated data set (&DSN) exists. You can then query &EXIST to determine how subsequent steps should be executed.

#### Program 7.1.3a: Using %STR to Mask Semicolons in a DATA Step

```
%macro exist(dsn=);
%global exist;
%if &dsn ne %then %str( ❶
  data _null_;
  stop; ❸
  set &dsn; ❷
  run;
); ❶
%if &syserr=0 %then %let exist=yes; ❹
%else %do;
  %let exist=no;
  %put WARNING: PREVIOUS ERROR USED TO CHECK FOR PRESENCE ;
  %put WARNING- OF DATASET &DSN & IS NOT A PROBLEM;
%end;
%mend exist;
%exist(dsn=sashelp.clxxxx)
```

- ❶ The %STR function is used to mask the semicolons of the statements making up the DATA step. This enables us to conditionally execute the DATA step using a macro %IF.
- ❷ During DATA step compilation, the SET statement is used to check if the data set can be opened (essentially this checks to see whether it exists). If the data set does not exist, the compilation of this statement will cause an error and the automatic macro variable &SYSERR ❹ will be set to a nonzero value.
- ❸ The STOP statement terminates the execution of the step before any data is read.
- ❹ The automatic macro variable &SYSERR will contain a zero when the data set named in &DSN exists.

Program 7.1.3b shows that the use of the %STR quoting function ❶ could have been avoided altogether by using a %DO block instead of the %STR function.

#### Program 7.1.3b: Using a %DO Block to Mask a DATA Step

```
.... code not shown ....
%if &dsn ne %then %do; ❺
  data _null_;
  if 0 then set &dsn;
  stop;
  run;
%end; ❺
.... code not shown ....
```

- ❺ By using a %DO block to mask the semicolons of the DATA step, the quoting function is no longer needed and the coding is more intuitive.

In several of the remaining examples in this section quoting is used to solve a number of coding situations. Some of these situations could have been avoided by rethinking the problem.

**MORE INFORMATION:** A more sophisticated version of the %EXIST macro can be seen in Program 7.5.2c.

### Quoting before or after Passing Values

There are times when the text that requires quoting is to be passed as a macro parameter. The macro variable &TTL below contains a single quote that can, depending on how the macro variable is used, cause a problem.

#### Program 7.1.3c: Using a Macro Variable Containing a Single Quote

```
data _null_;
call symputx('ttl','Tom's Truck');
run;
```

The resolved value of &TTL will be fine in TITLE1 because it will be enclosed in double quotes. However, in TITLE2 quotes are not used and the TITLE statement will not be properly defined.

```
title1 "&ttl";
title2 &ttl;
proc print data=sashelp.class;
run;
```

We might want to protect our macros from unanticipated characters, in this case a single quote, by using quoting functions. Both of the following titles will work correctly and will give the same result because the single quote mark has been masked using %BQUOTE.

```
title1 "%bquote(&ttl)";
title2 %bquote(&ttl);
proc print data=sashelp.class;
run;
```

Of course, using the %BQUOTE with every macro variable is not practical. When passing values into a macro, you might consider quoting the parameter before it is passed. In the following macro, the TITLE statements and the PROC PRINT step from Program 7.1.3c have been incorporated into the macro %PPRINTIT.

#### Program 7.1.3d: Passing Special Characters as Parameters

```
%macro pprintit(txt=);
  title1 "&txt";
  title2 &txt;
  proc print data=sashelp.class;
  run;
%mend pprintit;
```

If the passed parameter (&TXT) contains an unmatched quote, the macro %PPRINTIT will have the same issues as the TITLE statements shown in Program 7.1.3c. The macro call shown here for %PPRINTIT, however, will have its own problems.

```
%pprintit(txt=&ttl)
```

Remember that the macro variable &TTL is resolved before %PPRINTIT is called. The macro call with the resolved value would include the single quote.

```
%pprintit(txt=Tom's Truck)
```

Again, the single quote causes a problem, this time in the macro call (before it is even passed to the TITLE statements). Quoting the resolved value solves this problem as well those in the TITLE statements.

```
%pprintit(txt=%bquote(&ttl))
```

Quoting the parameter in a macro call can also be useful when the resolved value contains commas. Consider the following value of &LST and the call to %PPRINTIT.

```
13  %let lst = a, b, c;
14
15  * This macro call will fail;
ERROR: All positional parameters must precede keyword parameters.
16  %pprintit(txt=&lst)
```

We receive an ERROR message because portions of the resolved value in the macro call are misinterpreted as positional parameters.

```
%pprintit(txt=a, b, c)
```

Again, the use of macro quoting to mask the commas solves the problem.

```
%pprintit(txt=%bquote(&lst))
```

Quoting the parameters prior to calling the macro is not always necessary; certainly the macro calls will be easier to construct if the quoting can be handled inside the macro. The macro %STCODES shown below is passed the postal code abbreviation of a state and determines if the state code is for California (CA).

#### **Program 7.1.3e: Masking Special Characters inside of a Macro**

```
%macro stcodes(stcode=WA);
  %if CA=&stcode %then %put &=stcode: State is California;
  %else %put &=stcode: State is NOT California;
%mend stcodes;
```

The macro works fine until we pass in a value for Oregon. The OR will cause an error because the resolved value of &STCODE is seen as a Boolean operator.

```
51  %stcodes(stcode=OR)
ERROR: A character operand was found in the %EVAL function or %IF condition
      where a numeric operand is required. The condition was: CA=&stcode
ERROR: The macro STCODES will stop executing.
```

The OR has attitude, and we can mask the resolved value by using a quoting function. The quoting function could be applied in the macro call, as was done in Program 7.1.3d, or it can be placed in the %IF statement where the problem is encountered:

```
%macro stcodes(stcode=WA);
  %if CA=%bquote(&stcode) %then %put &=stcode: State is California;
  %else %put &=stcode: State is NOT California;
%mend stcodes;
```

In more complex situations %BQUOTE might not be sufficient to mask all special characters sufficiently. Rosenbloom and Carpenter (2011) demonstrate a number of alternate situations where various quoting functions might or might not be helpful.

#### **How to See the Invisible Marks**

Things that are invisible tend to be hard to see. Things that are hard to see often go unnoticed. How do we notice when items have been quoted? We know that the masking characters added by the quoting functions are not only invisible, but that they also become a part of the text. Usually, this is a good thing. In Program 7.1.3f the macro variable &OWNER contains a masked single quote.

#### **Program 7.1.3f: Surfacing Hidden Masking Characters**

```
%let owner = %str(Tom%'s Truck);
```

As was shown earlier (Program 7.1.3c), without masking the single quotation mark, the text in the assignment would be misinterpreted. However, once the text has been quoted, the invisible marks remain and whenever we use &OWNER, even in another macro variable assignment, we can do so without a problem, because the single quote has been masked. The use of the percent sign in front of the single quote is further explained in Section 7.1.9.

```
%let vehiclecolor = &owner is blue;
```

The invisible quotation marks remain to protect us and the resolved value.

Because the quotation marks are invisible, it is sometimes hard to know if a macro variable contains any quoted elements. If you want to verify that the quotation marks exist, or if you just need to determine if something has been quoted, you can often use the MLOGIC system option to make these symbols visible in the SAS Log.

With MLOGIC turned on, the macro call %STCODES (STCODE=AK) is shown in the SAS Log for the execution of Program 7.1.3g. When using the techniques discussed here, many, but not all, fonts will surface the masking characters as blanks. Here the font “Courier New” has been selected as the SAS Log font. Other fonts would likely cause the masking characters to appear differently.

#### Program 7.1.3g (SAS Log): Surfacing Masking Characters

```

98  %macro stcodes(stcode=WA);
99    %if CA=%str(&stcode) ❶ %then %put &=stcode: State is California;
100   %else %put &=stcode: State is NOT California;
101  %mend stcodes;
102
103 options mlogic; ❷
104 /* These macro calls work as expected;
105 %stcodes(stcode=AK)
MLOGIC(STCODES): Beginning execution.
MLOGIC(STCODES): Parameter STCODE has value AK
MLOGIC(STCODES): %IF condition CA=_&stcode_ is FALSE ❸
MLOGIC(STCODES): %PUT &=stcode: State is NOT California
STCODE=AK: State is NOT California
MLOGIC(STCODES): Ending execution

```

- ❶ The %STR quoting function is used instead of %BQUOTE. %STR is a compilation time quoting function and its usage allows the masking characters to be revealed in the SAS Log.
- ❷ The MLOGIC system option is turned on.
- ❸ When translated into this book, the invisible masking marks are shown as underscores (\_). The symbol might be different depending on your operating system, version of SAS, and selected font. A number of fonts simply display the masking characters as blanks. In the original SAS Log for this example, before pasting the SAS Log into the word processor, the underscores appeared as small boxes.

Since MLOGIC only applies to values in a %IF expression, the macro %ISITQUOTED can be used to check an individual macro variable for quoting.

#### Program 7.1.3h: Surfacing Masking Characters in a Macro Variable

```
%macro isitquoted(var=);
  %if &var ne %then %put &var;
%mend isitquoted;
```

The macro variable &OWNER has been created using %STR. We can surface the masking characters using the macro %ISITQUOTED.

```
options mlogic;
%let owner = %str(Tom%'s Truck);
%let vehiclecolor = &owner is blue;
%isitquoted(var=&vehiclecolor)
```

The SAS Log shows the following:

```
119 options mlogic;
120 %let owner = %str(Tom%'s Truck);
121 %let vehiclecolor = &owner is blue;
122 %isitquoted(var=&vehiclecolor)
MLOGIC(ISITQUOTED): Beginning execution.
MLOGIC(ISITQUOTED): Parameter VAR has value _Tom_s Truck_ is blue
MLOGIC(ISITQUOTED): %IF condition &var ne is TRUE
MLOGIC(ISITQUOTED): %PUT &var
Tom's Truck is blue
MLOGIC(ISITQUOTED): Ending execution.
```

The underscores give some indication as to what is quoted.

Notice that when the %PUT statement is used to display &VAR, indications of the masking characters do not appear. It is possible, however, to use the %PUT statement to display these characters, even without using MLOGIC, by using the %PUT statement options that list macro variables (\_LOCAL\_, \_GLOBAL\_, \_USER\_, and \_ALL\_).

**Figure 7.1.3i: Showing Masking Characters Using %PUT and the Terminal Font**

```
129 %put _user_;
GLOBAL OWNER ♦Tom's Truck♦
GLOBAL VEHICLECOLOR ♦Tom's Truck♦ is blue
```

Remember that the special characters shown in Figure 7.1.3i may be different depending on your operating system, version of SAS, and available fonts.

#### 7.1.4 Basic Types of Quoting Functions and Why We Care

As has already been pointed out, each quoting function has its own attributes and behaviors. You should have an understanding of the differences between compilation time versus execution time functions, as well as whether the function resolves its argument through rescanning.

##### NR Functions (for Example, %NRSTR and %NRQUOTE)

Of special interest in the text strings to be quoted are those that contain the special macro symbols % and &. Normally, these symbols indicate references to macro calls and macro variables that must be resolved prior to the execution of the function. These references are resolved by rescanning the text as many times as it takes to resolve the usage of the % and &. Sometimes this is not how you want to have these types of references handled. Sometimes you do not want these references resolved at all. Sometimes you want to pass the macro call or macro variable without resolution. In each of these cases, you need a function designated as a no-rescan or no-resolve (NR) function. Most of the quoting functions come in pairs (the name either does or does not start with the letters NR, such as, %STR and %NRSTR).

These two forms of the quoting functions perform essentially the same operation except that the function that begins with the letters NR will handle % and & differently than those that do not start with NR. When a quoting function whose name starts with NR is used, the macro processor will not see the % and & as the special characters in exactly the same way that it normally does. It is important to note, however, that the NR quoting functions do not all remove the meaning in the same way for each of the quoting functions.

Nor does the NR necessarily completely prevent the resolution of a macro reference. The topic is complicated, and some examples are included here to highlight some of these differences.

The %NRSTR function behaves in the same way as %STR except that meaning is also removed from the % and the & at compile time. In Program 7.1.4a, the macro variable &CITY is not resolved in the %PUT because the special meaning has been removed from the &.

#### **Program 7.1.4a: Using NR Quoting Functions**

```
%let city = Miami;
%put %nrstr(&city) is on the water.;
```

The SAS Log shows that the macro variable &CITY remains unresolved.

```
&city is on the water.
```

The %NRBQUOTE function is the no-resolve equivalent of the %BQUOTE function. Like the %NRSTR function, the %NRBQUOTE masks & and %; however, unlike the %NRSTR, it will first attempt to resolve the macro references and will then quote the result. In the following example the macro variable &STORE is given the value of A&P. Assume that the macro variable &P has not been defined.

```
%let store = A&P;
%put The store name is &store;
```

When the %LET is executed, a warning (Apparent symbolic reference P not resolved.) will be issued, and the unresolvable result (A&P) is placed into &STORE. Whenever &STORE is used it will be resolved to A&P, the macro facility will again attempt to resolve &P and another warning will be issued. Therefore, the %PUT statement will result in a second warning. However, if the %NRBQUOTE is used when &STORE is created only the first warning from the %LET will be issued and subsequent uses of &STORE will not result in additional warnings as the &P is not rescanned.

```
%let store = %nrbquote(A&P);
%put The store name is &store;
```

The use of the %NRSTR quoting function would not have resulted in any warnings because no attempt is made to resolve the &P.

```
%let store = %nrstr(A&P);
%put The store name is &store;
```

#### **Function Application at Compilation versus Execution**

Because macro statements are compiled and then executed, you may, on occasion, need to control when the text string is to be quoted. With the exception of %STR and %NRSTR, each of the quoting functions performs their action during macro execution. When the function is applied during compilation (%STR and %NRSTR), only the resultant text is passed to the processor. All of the other quoting functions are applied during the execution of the macro. For these functions, the entire call to the function and its associated text is passed to the macro processor where the function is applied when the statement is executed.

This will rarely be an issue for most macro programmers. Usually, you need to worry about compilation versus execution only if text that is resolved during compilation becomes syntactically incorrect or if it is misinterpreted during execution. Generally, when you want to quote the current value of the argument (typically constant text as opposed to a macro variable reference) the %STR and %NRSTR functions will be used. While the other quoting functions work with the resolved values.

The %LET statements in Program 7.1.4b demonstrate these differences. The macro variable &LEFTLIST is intended to contain a left-justified and comma separated list of variable names. The comma causes the

%LEFT function to fail because the comma is perceived as an argument separator rather than part of the list. The use of a quoting function such as %STR function prevents the error.

#### Program 7.1.4b: Compilation versus Execution Time Quoting

```
%* The comma causes an error with %LEFT;
%let leftlist = %left(    height, weight);

%* %STR prevents the error in %LEFT;
%let leftlist = %left(%str(    height, weight));
```

If the comma separated list is first stored in a macro variable (&LIST), before using it as the argument to the %LEFT function, an error will still be generated as &LIST will be resolved before the %LEFT function is executed.

```
%let list = height, weight;
%let leftlist = %left(    &list);
```

In this situation the %STR function will not prevent the error.

```
%let leftlist = %left(%str(    &list));
```

Because %STR (and %NRSTR) is a compilation time function, it will be applied to &LIST and not to the resolved value of &LIST. Since the other quoting functions are execution time functions they will be applied to the resolved value of &LIST. My preference is for the %BQUOTE function.

```
%let leftlist = %left(%bquote(    &list));
```

When &LIST resolves, the comma will be masked by the %BQUOTE and %LEFT will execute successfully.

We can also see the difference between compilation time and execution time in Program 7.1.4c. Here we want to compare a symbol (' | ') with the letters CA. The comparison is of course false, but we want to see whether or not the symbol has been successfully masked. If it remains unmasked the comparison will cause an error.

#### Program 7.1.4c: Masking a Symbol

```
%macro tst;
%if %str(|)=CA %then %put equal;
%else %put not equal;
%mend tst;
%tst
```

This comparison works as the %STR function is operating on constant text. However, if the symbol had been stored in a macro variable, then %STR would not have masked it and an error would have resulted. This is because the %STR operates before the resolution of &STATE.

```
%let state=|;
%if %str(&state)=CA %then %put equal;
```

Because &STATE is resolved during the execution of the macro, using the %BQUOTE function to mask the symbol would be successful.

```
%let state=|;
%if %bquote(&state)=CA %then %put equal;
```

The value of &STATE is determined during execution of the macro. Consequently, an execution time quoting function is needed.

**MORE INFORMATION:** Although %LEFT acts like a function, it is actually an autocall macro supplied with SAS. Read more about this and other SAS supplied autocall macros in Section 10.6.

**SEE ALSO:** Cheng et al (2016) use %NRSTR to mask a macro call when using the CALL EXECUTE routine.

### 7.1.5 A Bit about the %QUOTE and %NRQUOTE Functions

The %QUOTE and %NRQUOTE functions are very similar to %STR and %NRSTR in terms of what characters are masked. The primary difference is that they are applied during the execution of the macro statement rather than during macro compilation. With the availability of the stronger %BQUOTE and %NRBQUOTE functions, these two functions are now only rarely used.

Like the %STR and %NRSTR functions, you can mark mismatched quotes and parentheses so that they too will be masked (see Section 7.1.9) when using these two functions.

**SEE ALSO:** Whitlock (1999c) uses the %QUOTE function in an example that limits the number of observations in an output data set.

### 7.1.6 Removing Masking Characters

Once a quoting function has been applied, its effects can remain associated with the text, even in subsequent usages. There are several ways to remove the masking characters that have been inserted through the use of various quoting functions. The primary way to remove quoting is to use the %UNQUOTE function.

In Program 7.1.6 the call to the macro variable &CITY is masked by the %NRSTR quoting function.

#### Program 7.1.6: Using %UNQUOTE

```
%let city = Miami;
%let nrcity = %nrstr(&city); ①
%let unq = %unquote(&nrcity); ②

%put &=city &=nrcity &=unq;
```

Examination of the SAS Log shows that &NRCITY resolves to &CITY, however because of the masking done by %NRSTR, &CITY does not resolve further.

```
66  %put &=city &=nrcity &=unq;
CITY=Miami NRCITY=&city UNQ=Miami ③
```

- ① &NRCITY is defined using the %NRSTR function.
- ② Since the %UNQUOTE is used to counter the effects of the %NRSTR function, &UNQ will not contain the same value as &NRCITY.
- ③ Because of the use of %NRSTR, &CITY cannot be resolved further when &NRCITY is resolved. However, when the %UNQUOTE function is applied to &NRCITY its value (&CITY) is seen as a macro variable that can also be resolved.

When &NRCITY is used in the %PUT there is no indication that the value that it contains is quoted. A visual inspection of its value (&CITY) does not give any clue as to why &CITY failed to resolve. Sometimes we can use the techniques discussed in Section 7.1.3 (How to see the invisible marks) to show that text has been quoted.

**MORE INFORMATION:** Section 7.1.3 includes a discussion on methods that you can use to determine if macro variables contain quoted text. Program 7.2.8 uses the %UNQUOTE function before trimming a value. The fourth example in Program 12.1.2e uses the %UNQUOTE when building a FILENAME statement.

**SEE ALSO:** Carpenter (1998) and Rosenbloom and Carpenter (2011) both include an example of quoted strings that can become problematic.

### 7.1.7 The %SUPERQ Quoting Function

The %SUPERQ function operates only on the values of macro variables and it masks all items that might require quoting at macro execution. The argument to the function is the name of a macro variable (without the ampersand) or text that resolves to the name of a macro variable.

%SUPERQ provides the ultimate in quoting protection and is often used with macro variables that might contain text that is supplied by the user, such as text that might be encountered through the use of the %INPUT statement, the %WINDOW/%DISPLAY statements, and &SYSBUFFR. Several of the examples shown in *SAS 9.4 Macro Language Reference, Fourth Edition* (pp. 91–94) refer to the use of &SYSBUFFR. It can also be very helpful when data values that contain special characters have been written to macro variables in either an SQL step or through the use of the SYMPUTX routine (this situation is simulated in Program 7.1.7a).

In Program 7.1.7a, the macro variable &HASCALL contains a call to the macro %DOIT. We would like to write the contents of the &HASCALL macro variable to the SAS Log without executing the macro call.

#### Program 7.1.7a: Demonstrate the Use of %SUPERQ

```
data _null_;
  call symputx('hascall', 'NOTE: Call macro %doit');
  call symputx('call2',   'NOTE: Call macro %doit');
  run;

%put &hascall; ①
%put %superq(hascall); ②
```

- ① When this %PUT executes, it will write the value of &HASCALL to the SAS Log, however as it is resolved, the call to macro %DOIT will surface and the call to %DOIT will be executed.
- ② You can prevent the execution of %DOIT by masking the % sign using the %SUPERQ function. Notice that when you use the %SUPERQ function, the macro variable name is stated without the ampersand. %SUPERQ assumes that its argument is the name of a macro variable.

The argument to %SUPERQ can contain an ampersand. The argument is resolved, and if the resolved value is a macro variable one more resolution takes place before the quoting.

```
681 %put %superq(&hascall); ③
WARNING: Apparent invocation of macro DOIT not resolved.
ERROR: Invalid symbolic variable name NOTE CALL MACRO %DOIT.
```

- ③ Using an ampersand with HASCALL would have caused an error because the resolved value of &HASCALL is not a valid macro variable name.

Macro variables that contain the name of the macro variable of interest can be used, however how we address the macro variable depends on what it contains and how we want it to be resolved. Typically, macro variables that hold the name of another macro variable would be addressed using a triple ampersand – either &&&VAR or &&VAR&I.

```
%let callname = hascall;
/* &&&callname resolves to &hascall;
%put %superq(&callname); ④
```

- ④ The argument to %SUPERQ uses a single & (the first two ampersands, which delay resolution are not needed). &CALLNAME resolves to HASCALL, which is exactly what we want.

When working with indexed macro variable lists of the form &&VAR&I, the argument for %SUPERQ might or might not need the first two ampersands depending on whether the indexed macro variable contains the name of another macro variable or the text to be quoted. In this example, &CALL2 contains the text to be quoted – the leading double ampersands are not needed.

```
%let i=2;
%put %superq(call&i);
```

When the indexed macro variable holds the name of the macro variable the leading ampersands will be needed.

```
693 %let hold2 = hascall;
694 %put %superq(hold&i); ⑤
hascall
695 %put %superq(&&hold&i); ⑥
NOTE: Call macro %doit
```

- ⑤ the argument HOLD&I is resolved to HOLD2, which is recognized as a macro variable and resolved to HASCALL. Although HASCALL is the name of a macro variable, %SUPERQ does not reread a second time (when no ampersands are present), and HASCALL is not further resolved.
- ⑥ Within the argument of %SUPERQ multiple rescans will take place as long as ampersands are present. &&HOLD&I resolves to &HOLD2, which in turn resolves to HASCALL. %SUPERQ can now resolve the macro variable HASCALL (this is the first and only scan performed by %SUPERQ).

The technique discussed in Program 5.2.1 compared the macro variable &VLIST to a null value by leaving the right side of the comparison operator blank.

```
%if &vlist ne %then %sortit(dset=&indata, bylist=&vlist);
```

While this technique is usually successful, it can fail if &VLIST happens to contain any of the special characters that can require quoting. Chung and King (2009) discuss various methods for the comparison of a macro variable to a null value. They show that the most robust test of a null value uses a combination of %SUPERQ and the %SYSEVALF functions.

#### Program 7.1.7b: Testing for Null Values

```
%macro test4null(val);
%if %sysevalf(%superq(val)=,boolean) %then %put the value is null;
%mend test4null;

%test4null(abc)
%test4null()
```

Remember that the argument to %SUPERQ is a reference to a macro variable. Odd things can happen if your resolved value contains a mixture of constant text and macro references. The code in Program 7.1.7c is adapted from a question in a LinkedIn discussion on macro quoting. Notice that three macro variables are defined, and that %SUPERQ references the first (&PUB).

#### Program 7.1.7c: Understanding %SUPERQ References

```
options symbolgen;
data _null_;
call symputx('pub','Fox&Hounds');
call symputx('hounds',' and &dogs');
call symputx('dogs','beagles');
run;

%put %superq(pub); ⑦
%put %superq(&pub); ⑧
```

- ⑦ %SUPERQ references the macro variable PUB without an ampersand. Consequently, the resolved value (Fox&Hounds) is quoted and not resolved further – even though &HOUNDS is a defined macro variable.
- ⑧ %SUPERQ references the macro variable PUB with an ampersand. SAS is expecting the resolved value of &PUB to be a macro variable name. FOX cannot be further resolved, however &HOUNDS can be. Since we are building a macro variable name, the scanner must resolve HOUNDS to determine what other text is needed to finish the name that starts with FOX. Because of the embedded spaces in the resolved code, a valid macro variable name cannot be constructed and when all the resolutions are completed an error is generated.

```

119  %put %superq(pub); ⑦
Fox&Hounds
120  %put %superq(&pub); ⑧
SYMBOLGEN: Macro variable PUB resolves to Fox&Hounds
SYMBOLGEN: Macro variable HOUNDS resolves to and &dogs
SYMBOLGEN: Macro variable DOGS resolves to beagles
ERROR: Invalid symbolic variable name FOX BEAGLES.

```

If there had been no embedded spaces in the macro variable definitions, the resolution could have resulted in a valid macro variable name.

```

data _null_;
call symputx('pub','Fox&Hounds');
call symputx('hounds','and&dogs');
call symputx('dogs','beagles');
call symputx('foxandbeagles','Hunt Day');
run;
%put %superq(&pub);

```

In this version of the code, &PUB eventually resolves to FOXANDBEAGLES, which is a defined macro variable, and the %PUT will write ‘Hunt Day’ to the SAS Log.

```

SYMBOLGEN: Macro variable PUB resolves to Fox&Hounds
SYMBOLGEN: Macro variable HOUNDS resolves to and&dogs
SYMBOLGEN: Macro variable DOGS resolves to beagles
127  %put %superq(&pub);
Hunt Day

```

**MORE INFORMATION:** The %SYSEVALF function is introduced in Section 7.3.3.

**SEE ALSO:** Mason (2016) uses %SUPERQ to resolve a macro variable that forms a part of the name of the desired macro variable. Gondara (2014) uses %SUPERQ in a robust comparison for a null value.

### 7.1.8 Quoting Function Summary

Items that might need quoting can be placed into one of three groups, A, B, or C. Whether or not a given symbol or mnemonic needs to be quoted will of course depend on its usage, and not all of the quoting functions handle the items in these three groups equally.

**Table 7.1.8a: Groups of Symbols That May Need Quoting**

Group	Items	Type
A	+ - * / < > = ^ ; , ~ #   blank	Symbols
	AND OR NOT EQ NE LE LT GE GT	Comparison operators
B	& %	Macro triggers
C	' " ( )	Unmatched and unmarked symbols normally expected in pairs, such as quotation marks or parentheses

The macro quoting functions deal with these three groups differently. Table 7.1.8b shows which groups of symbols can be quoted using the various quoting functions.

**Table 7.1.8b: Groups Quoted by the Various Quoting Functions**

Function	Groups Affected	Works At
%STR	A	Macro compilation
%NRSTR	A, B	
%QUOTE	A	Macro execution
%NRQUOTE	A, B	
%BQUOTE	A, C	Macro execution
%NRBQUOTE	A, B, C	
%SUPERQ	A, B, C	Macro execution (prevents resolution)

**MORE INFORMATION:** See Section 7.1.9 for a discussion of a method of marking symbols in Group C when using %STR and %QUOTE.

**SEE ALSO:** Table 7.6 (p. 94) in *SAS 9.4 Macro Language Reference, Fourth Edition* contains a similar summary.

## 7.1.9 Quoting Mismatched Symbols with the %STR and %QUOTE Functions

A number of symbols are usually expected in pairs (see group C in Table 7.1.8a). These include the single quote, double quote, opening parenthesis, and closing parenthesis. Errors will usually be generated if you specify only one-half of the pair in a string. This can especially be a problem if you want to include one of these mismatched symbols in a macro variable. Fortunately, some of the macro quoting functions can successfully mask these characters.

Similar problems can be encountered when attempting to use these functions to mask the symbols & and %. When you are using the %STR or %QUOTE functions (or their NR analogs), and you need to mask a

symbol from Group B or C, you can precede the symbol with a %. This allows these functions to mask the symbol.

**Table 7.1.9: Masking Group C Characters in the %STR and %QUOTE Functions**

Problem	Notation	Example	Value Stored
Unmatched single quote	%'	%let x=%str(tr=a%');;	tr=a';
Unmatched double quote	%"	%let t=%str(title %"FIRST);	title "FIRST
Unmatched left parenthesis	%('	%let a=%str(log%(12);	log(12
Unmatched right parenthesis	%)	%let b=%str(345%));	345)
Percent sign next to a valid quote or parenthesis	%%%	%let p=%str(title "20%%");	title "20%";

**MORE INFORMATION:** These masking characters are used with %STR in the second example of the %DB2DATE macro in Program 7.6.1e.

**SEE ALSO:** Table 7.1.9 is based on Table 7.4 (p. 86) in *SAS 9.4 Macro Language Reference, Fourth Edition*

## 7.2 Text Functions

Macro text functions either change or provide information about the text string that is provided as one of their arguments. These functions are analogous to similar character functions in the DATA step. However, it is important to remember the differences between these macro functions and their DATA step counterparts. DATA step functions always work on character strings and DATA step variables. Macro functions are applied to text strings and macro variables and are never applied to the values of DATA step variables.

There are not a great many of these functions in the macro language. Some of those that are more commonly used are shown in this section. You should not consider those discussed in this book to be an exhaustive list of macro text manipulation functions.

Macro text functions come in two basic flavors. Since the usage is the same for both types, the typical macro programmer does not need to even know that there is a difference. There are true macro functions (see Table 7.2a), and there are autocall macros supplied with SAS that mimic macro functions (see Table 7.2b and Section 10.6).

**Table 7.2a: Macro Text Functions**

Section	Macro Function	Analogous DATA Step Function	Task
7.2.1	%INDEX	INDEX	Locate first occurrence of a text string.
7.2.2	%LENGTH	LENGTH	Count characters.
7.2.3	%SCAN %QSCAN	SCAN	Search for the <i>n</i> th word in a text string.
7.2.4	%SUBSTR %QSUBSTR	SUBSTR	Select text based on position.
7.2.5	%UPCASE %QUPCASE	UPCASE	Convert to uppercase.

Notice that several of these functions have two forms, such as %SCAN and %QSCAN. Functions whose names start with Q (quoting) remove the meaning from special characters such as the ampersand (&), percent sign (%), and from mnemonic operators in values that are returned by the function. Values returned by these functions are always masked, including macro triggers (& and %).

All of the functions that return text and are named without the Q, return unquoted text. This behavior is as if there is an implied %UNQUOTE function on the returned value.

You might be familiar enough with the macro language to know that there is another group of text functions that mimic DATA step character functions. Functions such as %LEFT, %TRIM, %LOWCASE, %VERIFY, and others are not true macro functions. In fact, they are autocall macros, written by SAS Institute developers, which mimic true functions. You can write your own macros that will behave like functions by using three simple rules (see Section 7.5).

**Table 7.2b: Autocall Macros That Mimic Macro Functions**

Section	Macro Function	Analogous DATA Step Function	Task
7.2.6	%LEFT %QLEFT	LEFT	Left-justify a text string.
7.2.7	%LOWCASE %QLOWCASE	LOWCASE	Convert all characters to lowercase.
7.2.8	%TRIM %QTRIM	TRIM	Truncate any trailing blank characters.
10.6.1	%VERIFY	VERIFY	Locate first of occurrence of text not specified

**MORE INFORMATION:** Autocall macros and autocall macro libraries are discussed in more detail in Section 10.6.

## 7.2.1 %INDEX

The %INDEX function searches the first argument (*argument1*) for the first occurrence of the text string in the second argument (*argument2*). If the target string is found, the position of its first character is returned as the function's numeric response.

### SYNTAX:

```
%INDEX(argument1, argument2)
```

### VALUE RETURNED:

Position of the text in the second argument (0 if not found)

In Program 7.2.1a three words are stored in the macro variable &X. The %INDEX function is then used to search in &X for the text characters 'Tall', and the positon of the T is then stored in the macro variable &Y.

### Program 7.2.1a: Using the %INDEX Function

```
%let x=Long Tall Sally;
%let y=%index(&x,Tall);
%put Tall can be found at position &y;
```

Notice that the second argument, Tall, is not in quotes. &Y is resolved in the %PUT statement and its value displayed in the SAS Log.

```
6   %put Tall can be found at position &y;
Tall can be found at position 6
```

Remember that quote marks are not used to denote constant text in the macro language as they are in the DATA step. If quotes had been used in the definition of &Y, the %INDEX function would not have found any matching text (a 0 would be returned) as the quote marks themselves would be included in the search pattern. This is an important point, be sure that you understand this critical difference between the macro language and processing in the DATA step.

It is also common for both arguments to be macro variables. The previous example could easily be rewritten using two macro variable arguments and still return the same result.

```
%let srch = Tall;
%let x=Long Tall Sally;
%let y=%index(&x,&srch);
%put &srch can be found at position &y;
```

Because %INDEX is a function, it can be used as part of an expression, and it is not unusual to use it as part of a logical comparison. The macro %CHECK in Program 7.2.1b is a simplified version of a macro that monitors an ongoing process. Periodically, %CHECK is executed and &VALUE is examined.

### Program 7.2.1b: Using %INDEX in a Logical Expression

```
%macro check(value=);
  %if %index(&value,emergency) %then
    %put ERROR: *** Critical Emergency***;
  %else %put NOTE: *** Process in bounds ***;
%mend check;
%check(value=watch engineering emergency)
```

The %INDEX function returns a value greater than 0 when the text in &VALUE is found. The %IF will be evaluated as true whenever the text in &VALUE contains the word 'emergency'.

**SEE ALSO:** Aboutaleb (1997b) uses the %INDEX function in an example that deals with character variable strings. Whitlock (1999c) uses the %INDEX function to check for a two-level data set name.

## 7.2.2 %LENGTH

The %LENGTH function determines the length (number of characters) of its argument. The number of detected characters is then returned. The %LENGTH function counts all characters including leading and embedded blanks from the left and through to the last non-blank character. Unlike the DATA step LENGTH function, which will always return a value of at least 1, when the argument for %LENGTH is a null string, the value 0 is returned.

### SYNTAX:

```
%LENGTH(argument)
```

### VALUE RETURNED:

Number of characters in *argument*  
0 if *argument* is a null string

The number of characters in either a *libref* or a *fileref* name cannot exceed 8 characters. This restriction can be checked using the %LENGTH function.

```
%if %length(&locref)=0 or %length(&locref)>8 %then %do;
```

Because the %LENGTH function returns a 0 for null strings, this comparison also checks to make sure that &LOCREF is not null.

Normally the %LENGTH function will not count trailing blanks, however it will do so when a quoting function has been used to mark the trailing blanks. Program 7.2.2 demonstrates how %LENGTH can be used to count trailing blanks that have been quoted.

### Program 7.2.2: Counting Trailing Blanks

```
%let b1 = x      x;❶
%let lenb1 = %length(&b1); ❷
%let b2 = %qsubstr(&b1,1,&lenb1-1); ❸
%let lenb2 = %length(&b2); ❹
%put b1 |&b1| &lenb1;
%put b2 |&b2| &lenb2;
```

- ❶ &B1 is created with a series of embedded blanks.
- ❷ The length of &B1 includes the embedded blanks.
- ❸ The %QSUBSTR function (see Section 7.2.4) is used to remove the last character in &B1. Because the %QSUBSTR is used, the trailing blanks are not trimmed by the %LET statement.
- ❹ The length of &B2 should be one less than the length of &B1.
- ❺ The SAS Log shows that the length of &B1 is 12 characters.
- ❻ The length of &B2 includes the trailing blanks that have been preserved using the quoting function.

```
205 %put b1 |&b1| &lenb1;
b1 |x           x| 12 ❺
206 %put b2 |&b2| &lenb2;
b2 |x           | 11 ❻
```

In common usage we expect trailing blanks to be eliminated when being assigned to variables using the %LET and we expect that trailing blanks will not be counted by the %LENGTH function. This example serves to illustrate that we need to be careful when making these kinds of assumptions.

**MORE INFORMATION:** The %QSUBSTR function is introduced in Section 7.2.4.

**SEE ALSO:** Aboutaleb (1997b) uses the %LENGTH function in an example that deals with character variable strings.

### 7.2.3 %SCAN and %QSCAN

The %SCAN and %QSCAN functions both search a text string (*argument1*) for the *n*<sup>th</sup> word (*argument2*) and return its value. If *argument3* is not otherwise specified, the same word delimiters are used as in the DATA step SCAN function. For an ASCII system, these include the following (for EBCDIC, the ~ is substituted for the ^):

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

%QSCAN masks the significance of special characters in the returned value.

#### SYNTAX:

```
%SCAN(argument1, argument2 <,delimiters <,modifiers> >)
%QSCAN(argument1, argument2 <,delimiters <,modifiers> >)
```

#### VALUE RETURNED:

*n*<sup>th</sup> word, where *n* is *argument2*. A null value is returned if the *n*<sup>th</sup> word does not exist.

### Using Delimiters with %SCAN

In Program 7.2.3a the macro variable &X is broken up into words using the %SCAN function.

#### Program 7.2.3a: Using %SCAN to Select Words

```
%let x=XYZ.A'BC'/XYY;
%let word=%scan(&x,3); ①
%let part=%scan(&x,1,z); ②
%put word is &word and part is &part;
```

The SAS Log shows that the %SCAN function has selected the appropriate words.

```
107 %put word is &word and part is &part;
      word is XYY and part is XY
```

- ① No delimiter has been specified, so the period and slash will separate the words.
- ② Notice that the third argument (delimiter list) is not enclosed in quotes as it would be in the DATA step SCAN function. If the third argument is specified, only those characters specified will be interpreted as word delimiters.

#### Program 7.2.3b: Using a Text Character as the Word Delimiter

```
%let x=XYZ.A'BC'/XYY;
%let slash=%scan(&x,1,/);
%put The one word before slash is &slash;
```

The SAS Log shows that the period is no longer seen as a word delimiter.

```
125 %put The one word before slash is &slash;
      The one word before slash is XYZ.A'BC'
```

It is important to re-emphasize that the word delimiters should not be quoted. Remember that in the macro language quote marks are not interpreted as parsing characters as they are elsewhere within SAS. Using them might not cause an error, however you may not get the anticipated results.

#### Program 7.2.3c: Quoting the Delimiter List

```
%let x=XYZ.A'BC'/XYY;
%let two=%scan(&x,2,'/'); ③
%let three=%scan(&x,3,'/'); ④
%put The second word is &two;
%put The third word is &three;
```

The SAS Log shows that the quotes themselves are being treated as word delimiters:

```
218 %put The second word is &two;
The second word is BC
219 %put The third word is &three;
The third word is XYY
```

- ③ The first quote mark is seen as a word delimiter and BC is returned as the second word.
- ④ XYY is returned as the third word, showing that two adjacent word delimiters do not subtend a null value. You can cause adjacent word delimiters to subtend a null value by using a function modifier (see Program 7.2.3f).

The %QSCAN function is usually needed when you want to return a value that contains an ampersand, percent sign, or other special character that might otherwise be misinterpreted. The %SCAN will return these characters unquoted, even if they have been previously masked. Program 7.2.3d uses the %NRSTR function to mask the ampersands stored in &STRING. The %SCAN and %QSCAN functions are then used to retrieve the second word (&DSN).

#### Program 7.2.3d: Using %QSCAN to Mask an Ampersand

```
%let dsn = clinics;
%let string = %nrstr(*&stuff*&dsn*&morestuf);

%let wscan = %scan(&string,2,*);
%let wqscan = %qscan(&string,2,*);

%put &wscan &wqscan;
```

In the SAS Log we can see that the word returned by the %SCAN function has been resolved, while the result of the %QSCAN remains quoted and unresolved.

```
229 %put &wscan &wqscan;
clinics &dsn
```

Both functions return the value &DSN, but because the meaning of the & is not masked by %SCAN, &DSN is resolved to clinics before it is assigned as the value of &WSCAN.

The macro %CNTVAR in Program 7.2.3e counts the number of variable names that are contained in the macro variable &KEYFLD. In this example, each variable name is saved in a macro variable, such as &VAR2, as is the overall count of variables (&CNT).

#### Program 7.2.3e: Using %SCAN to Parse a List of Names

```
%macro cntvar(keyfld=);
  %global cnt; ⑨
  %let I = 1;
  %do %until(%scan(&keyfld,&I,%str( )⑤)=%str())⑥;
    %global var&I; ⑨
    %let var&I = %scan(&keyfld,&I,%str( ));
```

```
%let I = %eval(&I + 1); 7
%end;
%let cnt = %eval(&I-1); 8
%mend cntvar;
%cntvar(keyfld=region clinnum sex)
```

- 5** The word delimiter in both of these %SCAN function calls is a blank character, and the %STR function is used to preserve the space.
- 6** Notice that the second %STR function in the %DO %UNTIL statement does not encompass a blank because it is used to test for a null string. When the %SCAN function attempts to retrieve a word beyond the scope of the list, it returns a null value. This causes the %DO %UNTIL to terminate.
- 7** The counter ( &I ) is incremented. The %EVAL function is introduced in Section 7.3.1.
- 8** The counter was incremented one too many times and must be decremented by one to yield the correct word count.
- 9** The macro variables created in this macro are written to the global symbol table. Section 7.5 shows you how to avoid the use of the global symbol table by writing macro functions.

After running the macro shown, you might need to know the name of the final variable in the list so that it can be used in FIRST. and LAST. processing. You could use the following statement in a subsequent DATA step to select for unique observations:

```
if first.&&var&cnt and last.&&var&cnt;
```

Normally, the %SCAN function counts words from left to right, however you can use negative numbers as the second argument in %SCAN. Negative values count words from the right rather than from the left, and the following would return the last word in &KEYFLD without first knowing the number of words in &KEYFLD.

```
%let last = %scan(&keyfld,-1,%str( ));
```

This would enable us to rewrite the IF statement from above without knowing the number of words and without first using the %CNTVAR macro.

```
if first.%scan(&keyfld,-1,%str( )) and last.%scan(&keyfld,-1,%str( ));
```

## Using Modifiers with %SCAN and %QSCAN

Modifiers can be specified in the optional fourth argument to the %SCAN and %QSCAN functions. The specified modifiers effect the way that word delimiters are specified and used. Modifiers consist of a series of one or more space separated letters.

Examples of possible modifiers include the following modifiers:

- b scan words from right to left (same as using a negative word number)
- m multiple adjacent word delimiters separate a null word (&Y2 in Program 7.2.3f))
- q mask delimiters between quotes

In Program 7.2.3f the modifiers b, m, and q are used.

### Program 7.2.3f: Using the %SCAN Function with Modifiers

```
%let x = XYZ.aB/ABC//def\'a.b';
%let y1 = %scan(&x,1,,m b);
%let y2 = %scan(&x,2,,m b);
%let y3 = %scan(&x,3,,m b);
%put &=y1 &=y2 &=y3;

%let y4 = %qscan(&x,2,,,b);
%let y5 = %qscan(&x,2,,,b q);
```

```
%put &=y4 &=y5;
```

The SAS Log shows the application of the %SCAN function modifiers.

```
364 %put &=y1 &=y2 &=y3;
Y1=def\`a.b' Y2= Y3=ABC
. . . portions of the SAS Log not shown . . .
368 %put &=y4 &=y5;
Y4=aB/ABC//def\`a Y5=XYZ
```

**MORE INFORMATION:** The macro %WORDCNT in Program 7.3.2c uses a slightly different approach to count the number of words.

**SEE ALSO:** The %SCAN function is used in a similar manner in Roberts (1997). Izrael, Hoaglin, and Battaglia (2000) use the %SCAN to rank a list of numbers. Whitlock (1999c) uses the %QSCAN function in an example that limits the number of observations in an output data set. Lund (2003a) uses a -1 as the second argument to find the last BY variable in a list.

## 7.2.4 %SUBSTR and %QSUBSTR

Like the DATA step SUBSTR function, the %SUBSTR and %QSUBSTR macro functions return a portion of the string in the first *argument*. The substring starts at the *position* in the second argument and optionally has a *length* of the third argument.

### SYNTAX:

```
%SUBSTR(argument, position <,length>)
%QSUBSTR(argument, position <,length>)
```

### VALUE RETURNED:

Substring of *argument*

As is the case with most other macro functions, each of the three arguments can be a text string, macro variable, expression, or a macro call. If you do not specify a value for *length*, a string that contains all of the characters from *position* to the right end of the argument is returned. Unlike the %SCAN function, the second argument (*position*) cannot be negative.

### Program 7.2.4a: Using the %SUBSTR Function

```
%LET CLINIC=BETHESDA;
%IF %SUBSTR(&CLINIC,5,4) = ESDA %THEN %PUT *** MATCH ***;
%ELSE %PUT *** NOMATCH ***;
```

The %PUT statement will print \*\*\* MATCH \*\*\* in the SAS Log because &CLINIC has the value ESDA in characters five through eight.

The %QSUBSTR function enables you to return unresolved references, as well as other special characters, to macros and macro variables.

### Program 7.2.4b: Returning Quoted Text

```
%let dsn=clinics;
%let string = %nrstr(*&stuff*&dsn*&morestuf);

%let sub = %substr(&string,9,5);
%let qsub = %qsubstr(&string,9,5);

%put &=sub &=qsub;
```

The SAS Log shows that &QSUB contains the unresolved value of &DSN.

```
394 %put &=sub &=qsub;
SUB=clinics* QSUB=&dsn*
```

When the first argument to these functions contains trailing blanks, they are removed (there is an implied %TRIM) *before* the subsetting takes place. This is a changed behavior from some earlier versions of SAS.

#### Program 7.2.4c: Trailing Blanks and the %SUBSTR and %QSUBSTR Functions

```
data _null_;
  name='Joe      ';
  call symput('name',name); ①
  run;
%let sub = %substr(&name,1,4); ②
%put |&sub|;
%let qsub = %qsubstr(&name,1,4); ②
%put |&qsub|;
%let bqsub = %qsubstr(%bquote(&name),1,4); ③
%put |&bqsub|;
```

- ① The CALL SYMPUT routine does not truncate the value of NAME (SYMPUTX would have trimmed the value before assigning it to the macro variable &NAME).
- ② Because the value of &NAME is truncated before the subsetting takes place, a WARNING is issued and the trimmed value of &NAME is returned.
- ③ You can preserve the trailing blanks by quoting them using the %BQUOTE quoting function.

```
WARNING: Argument 3 to macro function %SUBSTR is out of range.
128 %let sub = %substr(&name,1,4);
129 %put |&sub|; ②
|Joe|
130 %let qsub = %qsubstr(&name,1,4);
WARNING: Argument 3 to macro function %QSUBSTR is out of range.
131 %put |&qsub|; ②
|Joe|
132 %let bqsub = %qsubstr(%bquote(&name),1,4);
133 %put |&bqsub|; ③
|Joe|
```

**MORE INFORMATION:** %QSUBSTR is used in an example with the %LENGTH function in Section 7.2.2.

## 7.2.5 %UPCASE and %QUPCASE

The %UPCASE macro function converts all characters in the *argument* to uppercase. This function is especially useful when you need to compare text strings that might have inconsistent case.

### SYNTAX :

```
%UPCASE(argument)
%QUPCASE(argument)
```

### VALUE RETURNED:

*ARGUMENT* in all capital letters

In Program 7.2.5 we want to detect the data set name and enable the user to differentially include a KEEP= option in the PROC PRINT statement. The %UPCASE function controls for variations in text that is supplied by the user in the macro call.

#### Program 7.2.5: Using %UPCASE in a Logical Comparison

```
%macro printit(dsn=);
  * use a KEEP for CLINICS;
  %if %upcase(&dsn)=CLINICS %then
    %let keep=(keep=lname fname ssn);
  %else %let keep=;
  proc print data=&dsn &keep;
    title "Listing of %upcase(&dsn)";
    run;
%mend printit;
```

When %PRINTIT is called with the MPRINT system option turned on, the SAS Log shows the code generated by the macro. Notice that the original case is maintained in the DATA= option on the PROC PRINT statement.

```
426 %printit(dsn=cLinICs)
MPRINT(PRINTIT):   * use a KEEP for CLINICS;
MPRINT(PRINTIT):   proc print data=cLinICs (keep=lname fname ssn);
MPRINT(PRINTIT):   title "Listing of CLINICS";
MPRINT(PRINTIT):   run;
```

---

#### 7.2.6 %LEFT and %QLEFT

Like the DATA step LEFT function, this macro has a single argument that it returns without any leading blanks.

##### SYNTAX:

```
%LEFT(argument)
%QLEFT(argument)
```

##### VALUE RETURNED:

The *argument* without any leading blanks

Program 7.2.6 shows how %LEFT and %QLEFT handle different text combinations involving leading blanks.

#### Program 7.2.6: Using %LEFT and %QLEFT

```
data _null_;
  x=5;
  store = ' A&P'; ②
  citystate = ' Seattle, WA'; ③
  call symput('five',x); ①
  call symputx('store',store); ②
  call symputx('cs',citystate); ③
  run;

%put Unjustified |&five|;
%put Left Justified |%left(&five)|;
```

- ① When converting numbers into text the SYMPUT routine creates a right-justified string (SYMPUTX creates a left-justified string). The %LEFT function left-justifies this string.

```
35 %put Unjustified |&five|;
Unjustified |      5|
36 %put Left Justified |%left(&five)|;
Left Justified |5|
```

- ② &STORE contains leading blanks and an embedded ampersand.

```
%put Left Justified |%left(&store)|;
%put QLeft Justified |%qleft(&store)|;
```

The text is correctly left-justified, however the &P causes a series of warnings in the SAS Log. Both the %LEFT and the %QLEFT functions issue these warnings, which are generated because of the internal functions that are called during the justification process.

```
43  %put Left Justified |%left(&store)|;
WARNING: Apparent symbolic reference P not resolved.
Left Justified |A&P|
44  %put QLeft Justified |%qleft(&store)|;
WARNING: Apparent symbolic reference P not resolved.
QLeft Justified |A&P|
```

Although it appears that both of these functions return the same result, that is not actually the case. The %QLEFT masks the ampersand in &P so that it will not cause more warnings.

- ③ The macro variable &CS contains a comma that will be misinterpreted as an argument delimiter by both the %LEFT and %QLEFT functions.

```
/* The first two %LET statements fail;
%let lcity1 = %left(&cs);
%let lcity2 = %qleft(&cs);
%let lcity3 = %qleft(%bquote(&cs));
%put |&lcity3|;
```

The %BQUOTE function can be used to mask the comma, which frees up the %QLEFT (or the %LEFT) function to left-justify the text.

Because of the way that the %LET statement makes its assignments, leading and trailing blanks are automatically removed. Since we are not using the %LEFT function, the comma does not cause a problem.

```
56  %let juststore = &cs;
57  %put |&juststore|;
|Seattle, WA|
```

**MORE INFORMATION:** The code associated with the %LEFT function is described in Section 10.6.2.

### 7.2.7 %LOWCASE and %QLOWCASE

The %LOWCASE and %QLOWCASE autocall macro functions can be used to convert any uppercase characters to lowercase characters. The function accepts a single argument, which is translated internally using the %INDEX and %SUBSTR functions.

**SYNTAX:**

```
%LOWCASE(argument)
%QLOWCASE(argument)
```

**VALUE RETURNED:**

The *argument* in all lowercase

Program 7.2.7 demonstrates the use of %LOWCASE by applying it to text with both upper and lower cases.

**Program 7.2.7: Using %LOWCASE**

```
%let mixed = SAS Macro Language;
%let lower = %lowcase(&mixed);
%put &lower;
```

The resulting SAS Log shows that the macro variable &LOWER contains all lowercase characters.

```
63  %put &lower;
    sas macro language
```

**7.2.8 %TRIM and %QTRIM**

Since trailing blanks will not usually be stored in a macro variable, and because macro variable storage is dynamically allocated, you will generally not need to use these functions. In the %LEFT function examples in Section 7.2.6, you will notice that trimming was not needed after left justification. That said, it is still possible for a macro variable to contain trailing blanks.

**SYNTAX:**

```
%TRIM(argument)
%QTRIM(argument)
```

**VALUE RETURNED:**

The *argument* without any trailing blanks

Trailing blanks can be stored in a macro variable in a number of ways. These include assignment from data set values (through CALL SYMPUT or PROC SQL's INTO clause).

**Program 7.2.8: Trimming Trailing Blanks**

```
data _null_;
  name='Joe      ';
  call symput('name',name);
  run;
%let qname = %bquote(Sam      );
%put |&name|;
%put |&qname|;
%put |%trim(&name)|;
```

The SAS Log shows the trailing blanks and their removal by %TRIM:

```
43  %let qname = %bquote(Sam      );
44
45  %put |&name|;
|Joe      |
46  %put |&qname|;
|Sam      |
```

```

47
48   %put |%trim(&name)| ;
|Joe|

```

Both the %TRIM and %QTRIM functions will remove the trailing blanks whether they were inserted using SYMPUT or the %BQUOTE quoting function.

Just as it can be used to left justify text, a %LET statement can also be used to trim trailing blanks, however they will not be removed if they were established using a quoting function.

```

/* Trim using %LET;
%let name = &name;
%let qname = &qname;
%let qname2 = %unquote(&qname);
%put |&name|;
%put |&qname|;
%put |&qname2|;

```

The SAS Log shows that the trailing blanks were removed by the %LET for &NAME, but not for &QNAME, which was established using a quoting function. When the %UNQUOTE function is used, the %LET is able to remove the trailing blanks.

```

51   %let name = &name;
52   %let qname = &qname;
53   %let qname2 = %unquote(&qname);
54   %put |&name|;
|Joe|
55   %put |&qname|;
|Sam    |
56   %put |&qname2|;
|Sam|

```

## 7.3 Evaluation Functions

Because there are no numbers or numeric variables in the macro language (there is only text and macro variables that contain text), numeric operations such as arithmetic and logical comparisons can become problematic. Certainly SAS and the computer that it is running on can easily perform these kinds of operations, however in the macro language we often need to identify situations when these operations are needed. This is done through the use of evaluation functions, which are used to bridge the gap between text and numeric operations. The DATA step does not have, and indeed does not need, analogous functions because numeric and character values are intrinsically different.

The evaluation functions are used

- To evaluate arithmetic and logical expressions
- Inside and outside of macros
- During logical comparisons to specify TRUE or FALSE (Evaluation functions return a value of 1 for logical expressions if the condition is true, 0 if it is false)
- To perform integer and floating-point arithmetic.

The requests for these functions are either explicit (called by name by the user) or implicit (used automatically by the macro language without being directly requested by the user).

### 7.3.1 Explicit Use of %EVAL

The %EVAL function always performs integer arithmetic. Even when the result of the arithmetic operation is not an integer (for example, the division of 7 by 3, which results in a floating point number), the value returned by the %EVAL function will still be an integer.

Like most languages that distinguish between integer and non-integer values, the macro language has specific rules that define integers. Certainly we would not expect the number 1.8 to be an integer, but what

about 1.0? It turns out that the mere presence of the decimal point is sufficient to cause a number to be interpreted as a non-integer. The numbers 1.0 and even 1. are both seen as non-integers and could not be used in integer arithmetic.

**SYNTAX:**

```
%EVAL (argument)
```

**VALUE RETURNED:**

Arithmetic result of the *argument*

In Program 7.3.1a the %EVAL function is called to perform arithmetic operations. The code uses %EVAL to add the value of 1 to &X, which in this case contains 5.

**Program 7.3.1a: Using %EVAL**

```
%let x=5;
%let y=&x+1;
%let z=%eval(&x+1);
%let zy=%eval(&y);
%put    &=x    &=y    &=z    &=zy;
```

The %PUT statement shows that arithmetic operations are detected and applied when the %EVAL evaluation function is used, but not otherwise:

```
188  %put    &=x    &=y    &=z    &=zy;
X=5      Y=5+1      Z=6      ZY=6
```

Unlike the rest of SAS, the macro language makes a distinction between integer and non-integer numbers. Since the %EVAL function can only operate on integers, non-integer arithmetic is not allowed. Use of non-integers in the %EVAL function will cause an error:

```
%let a=%eval(&x+1.8);
```

The error is generated because the decimal point prevents the number from being interpreted as an integer. When the macro language is unable to interpret the value as an integer, it is seen as a character (which is not allowed as part of the argument to the %EVAL function):

```
198  %let a=%eval(&x+1.8);
ERROR: A character operand was found in the %EVAL function or %IF condition
where a numeric operand is required. The condition was: 5+1.8
```

Although non-integer arithmetic is not allowed, integer arithmetic that results in a non-integer value is allowed. Non-integer results are truncated using the same rules as the INT function in the DATA step.

```
199  %let d = %eval(7 / 3);
200  %put &=d;
D=2
```

When the macro IN comparison operator (see Section 5.2.3) is used outside of a %IF, an explicit %EVAL function is required. In the first example for Program 7.3.1b the comparison is successful and the macro variable &INLIST will take on the value of 1.

**Program 7.3.1b: Using the %EVAL Function with the IN Operator**

```
%let state = CA;
%let list = CA WA ID NV;

%let inlist = %eval(&state in &list);
%put &=inlist;
```

Some states can cause the comparison to fail. If the list were to include the postal code for Oregon (OR), Indiana (IN) or Nebraska (NE), the codes themselves would be misinterpreted as being comparison operators. The following code returns an error in the %EVAL function.

```
%let state = CA;
%let list = OR CA IN NE WA ;
%let inlist = %eval(&state in &list);
```

The SAS Log shows an ERROR indicating that the assignment in the %LET was unsuccessful.

```
88   %let inlist = %eval(&state in &list);
ERROR: Operand missing for IN operator in argument to %EVAL function.
89   %put &=inlist;
INLIST=
```

The error is eliminated by quoting the list of postal codes with the %BQUOTE quoting function.

```
%let inlist = %eval(&state in %bquote(&list));
%put &=inlist;
```

**MORE INFORMATION:** Sections 5.3.3 and 5.3.4 provide other examples of the explicit use of the %EVAL function.

### 7.3.2 Implicit Use of %EVAL

Sometimes the %EVAL function will be used by the macro language even when it is not explicitly included in the code. This is an implicit use of %EVAL and there are a number of factors that will cause its insertion.

One potentially confusing use of the implied %EVAL occurs in some logical comparisons. If either of the values to be compared are non-integer, an alphabetic comparison is made, however when the comparison is between two integers a numeric, rather than an alphabetic, comparison is made.

#### Program 7.3.2a: Implied Use of %EVAL for Integer Comparisons

```
%macro chkwt(wt1, wt2);
  %if &wt1 > &wt2 %then %let note = heavier;
  %else %let note = lighter;
  %put First weight is &note.  &wt1 &wt2;
%mend chkwt;

%chkwt(1,2)
%chkwt(10,9)
%chkwt(2.1,2.2)
%chkwt(10.0,9.0)
```

Notice that the third and fourth calls to %CHKWT have non-integer parameters. When one or both of the values are not integers, the values will be compared alphabetically. Because 1 comes before 9 alphabetically, 10.0 is seen as smaller (alphabetically) than 9.0 in the fourth comparison.

```
241 %chkwt(1,2)
First weight is lighter 1 2
242 %chkwt(10,9)
First weight is heavier 10 9
243 %chkwt(2.1,2.2)
First weight is lighter 2.1 2.2
244 %chkwt(10.0,9.0)
First weight is lighter 10.0 9.0
```

The incorrect determination in the last comparison is not seen as a SAS mistake, but rather as a user error for working with non-integers! Remember that when determining whether a number is or is not an integer,

the macro interpreter bases the decision on the presence or absence of any non-integer characters, and the decimal point itself is a non-integer character.

In fact, the implied %EVAL function is used for all comparisons in the %IF statement – integer or not, but regardless, remember that for non-integers an alphabetic comparison takes place. It is as if the programmer had written the comparison using an explicit %EVAL:

```
%if %eval(&wt1 > &wt2) %then %let note = heavier;
```

We can see this more clearly in Program 7.3.2b, which shows a series of comparisons with and without an explicit %EVAL.

#### Program 7.3.2b: Demonstrating the Implied %EVAL

```
%macro tryit;
%if 5 %then %put is 5; ①
%if %eval(6) %then %put is 6; ①
%*if 7. %then %put is 7.; ②
%*if %eval(8.) %then %put is 8.; ②
%if %eval(9.=9.) %then %put is 9.; ③
%if 10.=10. %then %put is 10.; ④
%mend tryit;
%tryit
```

- ① Because only an integer is involved, both of these %IF statements are executed successfully.
- ② These %IF statements have been commented because both cause an error. Neither statement has a logical comparison, nor is the expression an integer. This means that the expression is a character and the %EVAL function cannot have a character argument.
- ③ The %EVAL is used explicitly on this comparison of two character values. Although it is a character comparison, the result (0 or 1) is numeric. Although this is a character comparison, the %EVAL function can be applied successfully.
- ④ Here the %EVAL is implied. Again, although the comparison is on two character values, the result is numeric, and because it is a comparison the %EVAL executes successfully.

In the following %IF the comparison is against an integer and a non-integer. As the comparison is being evaluated, the -1 is seen as an integer, but when the 2.0 is seen the -1 is then converted internally to character and a character comparison is made.

```
%if -1 < 2.0 %then %put test;
```

The minus sign can also be problematic. In the following comparison the minus sign must be used with a numeric value, however since .7 is not an integer, the conversion cannot take place and this causes results in an error in the %EVAL.

```
%if 1.2 < -.7 %then %put test;
```

An implicit use of the %EVAL also occurs within arguments of some functions. Consider an iterative %DO loop where the index variable is incremented by some number from a starting value through the loop's upper bound:

```
%do val = 1 %to 9 %by 2;
```

Since the %DO uses integers, each of the numbers has an implied %EVAL associated with it:

```
%do val = %eval(1) %to %eval(9) %by %eval(2);
```

Also, for the index variable (&VAL) to be incremented, an implied %EVAL is used behind the scenes:

```
%let val = %eval(&val + 2);
```

The implied use of the %EVAL function allows arithmetic expressions within the %DO statement itself.

#### Program 7.3.2c: Implied Use of %EVAL in the Iterative %DO Statement

```
%macro try;
%let x=2;
%do i = &x+5 %to &x+10 %by &x;
  %put &=i;
%end;
%mend try;
%try
```

Some of the arguments of some functions are also assumed to be numeric. The second argument of the %SCAN function (see Section 7.2.3) for instance is the word number. The implied %EVAL would enable us to specify the word number with an arithmetic operator.

```
%scan(&list, &x+1, %str())
```

The macro function %WORDCNT in Program 7.3.2d counts the number of blank separated words in the parameter that is passed into the macro. The second argument of the %QSCAN function has an arithmetic operation; however, the %EVAL has not been explicitly used. This works because SAS assumes that this argument is numeric and performs the arithmetic using an implicit %EVAL.

#### Program 7.3.2d: The Implied Use of %EVAL in the %QSCAN Function

```
%macro wordcnt(string=);
%local string wcnt;
/* The word count is stored in wcnt;
%let wcnt=0; ①
%do %until(%qscan(&string,&wcnt+1②,%str( ))=%str());
  %let wcnt=%eval(&wcnt+1);
%end;
&wcnt
%mend wordcnt;

/* example:;
%put count is %wordcnt(string=aa bb cc);
```

- ① Unlike in the %CNTVAR macro shown in Program 7.2.3e, the counter is initialized to zero. This simplifies the macro and prevents the need to “correct” the total at the end of the macro.
- ② Notice that the word number in the %QSCAN function is incremented through the use of an implicit %EVAL function.

Any of the macro functions that have implicitly numeric arguments (like the second argument of the %SCAN function) automatically invoke an implicit %EVAL. These include the following:

%DO	%IF-%THEN	%SUBSTR
%DO %UNTIL	%SCAN	%QSUBSTR
%DO %WHILE	%QSCAN	

### 7.3.3 Using %SYSEVALF

You can get around the integer limitations of %EVAL by using the floating point evaluation function, %SYSEVALF. This function can be used to perform non-integer arithmetic as well as logical comparisons, and it will even return a non-integer result from an arithmetic operation.

#### SYNTAX:

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

The *expression* is any arithmetic or logical expression that is to be evaluated, and it might contain macro references.

The second argument, *conversion-type*, is an optional conversion to apply to the value that is returned by %SYSEVALF. Because this function can return non-integer values, problems could occur in other macro statements that expect integers, but use this function's result. Therefore, when you need the result of this function to be an integer, use one of the conversion types.

A specification of the *conversion-type* converts the value that is returned by %SYSEVALF to another form. Possible values of *conversion-type* are shown in Table 7.3.3.

**Table 7.3.3: Conversion Types for the %SYSEVALF Function**

Conversion Type	Returned Value
BOOLEAN	0 if the result of the expression is 0 or missing 1 if the result is any other value
CEIL	Round to the next largest whole integer
FLOOR	Round to the next smallest whole integer
INTEGER	Truncate the decimal fraction

The CEIL, FLOOR, and INTEGER (which can be abbreviated as INT) conversion types act on the expression in the same way as the functions with the same name do in the DATA step. Program 7.3.3a demonstrates the results of various conversion types that are returned by the %SYSEVALF function.

#### Program 7.3.3a: Using %SYSEVALF

```
%let x = %sysevalf(7/3);
%put Using SYSEVALF &=x;
%let x = %sysevalf(7/3,boolean);
%put Using Boolean &=x;
%let x = %sysevalf(7/3,ceil);
%put Using Ceil &=x;
%let x = %sysevalf(7/3,floor);
%put Using Floor &=x;
%let x = %sysevalf(1/3);
%put Less than one &=x;
%let x = %sysevalf(1+.);
%put Handling a missing result &=x;
%let x = %sysevalf(1+,boolean);
%put Boolean on a missing result &=x;
```

The SAS Log shows the results of these calls to the %SYSEVALF function:

```
293 %let x = %sysevalf(7/3);
294 %put Using SYSEVALF &=x;
Using SYSEVALF X=2.33333333333333
295 %let x = %sysevalf(7/3,boolean);
296 %put Using Boolean &=x;
Using Boolean X=1
297 %let x = %sysevalf(7/3,ceil);
298 %put Using Ceil &=x;
Using Ceil X=3
299 %let x = %sysevalf(7/3,floor);
300 %put Using Floor &=x;
Using Floor X=2
301 %let x = %sysevalf(1/3);
302 %put Less than one &=x;
Less than one X=0.33333333333333
```

```

303 %let x = %sysevalf(1+.);
NOTE: Missing values were generated as a result of performing an operation
on missing values
      during %SYSEVALF expression evaluation.
304 %put Handling a missing result &x;
Handling a missing result X=.
305 %let x = %sysevalf(1+,boolean);
NOTE: Missing values were generated as a result of performing an operation
on missing values
      during %SYSEVALF expression evaluation.
306 %put Boolean on a missing result &x;
Boolean on a missing result X=0

```

The %SYSEVALF function can also be used for the comparisons of non-integer values. In macro %CHKWT in Program 7.3.2a, comparisons of non-integer values, even if they are numeric, are evaluated alphabetically. We can force a numeric comparison, even for non-integers, by using the %SYSEVALF function in the modified version of %CHKWT shown in Program 7.3.3b.

#### Program 7.3.3b: Using %SYSEVALF to Force a Numeric Comparison

```

%macro chkwt(wt1, wt2);
  %if %sysevalf(&wt1 > &wt2) %then %let note = heavier;
  %else %let note = lighter;
  %put First weight is &note.  &wt1 &wt2;
%mend chkwt;

%chkwt(1,2)
%chkwt(10,9)
%chkwt(2.1,2.2)
%chkwt(10.0,9.0)

```

The SAS Log shows that the comparison is now a numeric one for all values of &WT1 and &WT2:

```

312 %chkwt(10.0,9.0)
First weight is heavier  10.0  9.0

```

%SYSEVALF can also be used to build date, time, and datetime constants in the macro language. The following calls to %SYSEVALF convert the date (&DATE) and datetime constants to SAS date and datetime values.

#### Program 7.3.3c: Using Date and Datetime Constants with %SYSEVALF

```

%let time = 18nov2015:12:25:10;
%put &=time;
%put %sysevalf("&time"dt);
%let date = %sysevalf('31oct2015'd);
%put &=date;

```

The SAS Log shows that the conversions have been successfully made:

```

322 %let time = 18nov2015:12:25:10;
323 %put &=time;
TIME=18nov2015:12:25:10
324 %put %sysevalf("&time"dt);
1763468710
325 %let date = %sysevalf('31oct2015'd);
326 %put &=date;
DATE=20392

```

Date, time, and datetime constants can be used within the %SYSEVALF function, because it is expecting its argument to be numeric.

**MORE INFORMATION:** A date constant is converted using the PUTN function in Program 7.4.2a.

Although %SYSEVALF does not support a ROUND conversion type, rounding can be achieved by using the %SYSFUNC function in conjunction with %SYSEVALF (see Program 7.4.3).

The %SLEEP macro in Program 7.6.1j uses nested calls to %SYSEVALF to work with datetime constants.

## 7.4 Using DATA Step Functions and Routines

Two macro functions enable you to execute a majority of the functions and routines that are available in the DATA step as if they were a part of the macro language. Also known as bridging functions, because they bridge the gap between the DATA step and the macro language, these include %SYSCALL which calls DATA step routines, and %SYSFUNC which executes DATA step functions.

DATA step functions and routines typically operate against data set variables and constant strings, which are not available to the macro language. Instead, when you use these functions with %SYSFUNC and %SYSCALL, you will use them with macro variables and text values.

Since there are fewer CALL routines and because these tend to be used more infrequently than the other DATA functions, it is likely that you will tend not to use %SYSCALL very often. %SYSFUNC on the other hand is very useful and is used very often.

### 7.4.1 Using %SYSCALL

You can use %SYSCALL to execute CALL routines that are ordinarily called only from within the DATA step. Nearly all CALL routines, either user-written with PROC FCMP, or supplied with SAS, can be used with %SYSCALL. The primary exceptions include: LABEL, VNAME, SYMPUT, and EXECUTE.

#### SYNTAX:

```
%SYSCALL call-routine <(routine-arguments )> ;
%QSYSCALL call-routine <(routine-arguments )> ;
```

When you use %SYSCALL, the arguments will be macro variables specified without the ampersand. In Program 7.4.1a, the SYSTEM routine is used to execute the Windows operating system directory command.

#### Program 7.4.1a: Using %SYSCALL and the SYSTEM Routine

```
%let cmd = dir;
%syscall system(cmd);
```

Be sure to notice that the routine's argument is a macro variable *without the ampersand*. Inserting the directory command, DIR, directly instead of the macro variable &CMD would not have worked because DIR does not resolve to the name of a macro variable.

If you need to create a macro variable that contains a random number, you can use any of the numerous random number routines. Most of these exist as both functions and CALL routines. Program 7.4.1b assigns a uniformly distributed random number to the macro variable &RAND.

#### Program 7.4.1b: Using %SYSCALL to Generate Random Numbers

```
%let seed=0;
%let rand=0 ;
%put seed is &seed pseudo random number is &rand;
%syscall ranuni(seed,rand);
%put seed is &seed pseudo random number is &rand;
```

The SAS Log shows how the seed and random number are updated by the RANUNI routine:

```

57 %let seed=0;
58 %let rand=0;
59 %put seed is &seed pseudo random number is &rand;
seed is 0 pseudo random number is 0
60 %syscall ranuni(seed,rand);
61 %put seed is &seed pseudo random number is &rand;
seed is 328594136 pseudo random number is 0.15301356844278

```

Both arguments to RANUNI (&SEED and &RAND) must be initialized before the routine is called. The SEED value must be an integer, but the initial random number can be any number. The value of both macro variables, &SEED and &RAND, will be changed by the call to RANUNI. An initial seed of 0 will pull a value from the computer's clock as the initial seed used by RANUNII.

**MORE INFORMATION:** The macro %SYMCHECK uses %SYSCALL to execute the SET routine when reading the SASHELP.VMACRO view in Program 7.6.2c. The macro %BUILDVARLIST in Section 12.4.2e uses %SYSCALL to execute the RXFREE routine.

**SEE ALSO:** Maldonado et al (2014) generates a random number using the RANUNI function.

## 7.4.2 Using %SYSFUNC and %QSYSFUNC

These two macro functions greatly increase the list of functions that are available to the macro language by making available most of the DATA step functions. Not only does %SYSFUNC give us access to most DATA step functions, it can fundamentally change which of these functions we take advantage of. There are quite a few DATA step functions that will rarely, if ever, be used in the DATA step that become hugely valuable to the macro programmer. Because the macro language tends to use different functions than are used in the DATA step, it becomes critically important for the macro programmer to be well-versed in the use of DATA step functions.

### SYNTAX:

```
%SYSFUNC(function-name ( function-arguments ) <,format >)
%QSYSFUNC(function-name( function-arguments ) <,format >)
```

The first argument is the DATA step function that is to be executed and in the second optional argument you may supply a format that is to be applied to the result of the function.

Not all of the DATA step functions may be used with %SYSFUNC; however, most are available. You will notice slight variations in how the arguments to the functions are specified when they are used with %SYSFUNC. For the most part, the differences will be due to the differences between how the DATA step uses variable names and quoted strings, and the macro language's specification of macro variables and text (without quotes).

### Using %SYSFUNC to Avoid Using a DATA \_NULL\_ Step

Prior to the introduction of %SYSFUNC, a DATA \_NULL\_ step was often used to take advantage of DATA step functions. For the most part this type of programming has disappeared, however it is still sometimes seen either in legacy programs or in code generated by programmers unfamiliar with %SYSFUNC.

These DATA \_NULL\_ steps tend to read no data, create no data, and only exist to create a macro variable that is based on the result of a DATA step function. Program 7.4.2a is an example of a DATA \_NULL\_ step that solely exists to take advantage of the PUT function.

**Program 7.4.2a: Using a DATA \_NULL\_ Step to Create a Macro Variable**

```
data _null_;
date = put("&sysdate9" d, worddate18.);
call symputx('newdt', date);
run;
%put &=newdt;
```

This entire DATA step can be reduced to a single line of code by taking advantage of %SYSFUNC. Not only is the code more efficient, but it is easier to follow. Program 7.4.2b makes this conversion directly using %SYSFUNC. I have yet to encounter a situation where this type of DATA \_NULL\_ step cannot be replaced with a usage of %SYSFUNC.

**Converting Dates**

The format of the automatic macro variable &SYSDATE9 can be converted to the WORDDATE. format through the use of the PUTN function.

**Program 7.4.2b: Using %SYSFUNC with the PUTN Function**

```
%put &=sysdate9;
%let newdt = %sysfunc(putn("&sysdate9" d, worddate18.));
%put &=newdt;
```

The SAS Log shows that the format of the date has been converted:

```
121 %put &=sysdate9;
SYSDATE9=11FEB2015
122 %let newdt = %sysfunc(putn("&sysdate9" d, worddate18.));
123 %put &=newdt;
NEWDT=February 11, 2015
```

The PUTN function is used with %SYSFUNC to convert the date constant to an internal SAS date. This date is then converted back to text using the WORDDATE18. format. PUTN is the execution time analog of the PUT function, and it expects the first argument to be numeric. This enables us to use the date constant. The PUT function cannot be used with %SYSFUNC.

**MORE INFORMATION:** A date constant is also used in the %SYSEVALF function in Program 7.3.3c.

**SEE ALSO:** Burnett-Isaacs (2016a) uses the INPUTN function as well as the PUTN function to convert dates.

**Specifying Dates in Titles**

Program 7.4.2c shows three ways to add the current date to a TITLE statement. Although the automatic macro variable &SYSDATE is easy to use, it usually has to be reformatted. More importantly, since &SYSDATE and &SYSDATE9 only indicate the date that the current SAS session started, they might not be current. We could use a DATA \_NULL\_ step and the DATE function to create a formatted macro variable, however as was noted in Program 7.4.2a, the DATA step can be avoided by using the %SYSFUNC macro function.

**Program 7.4.2c: Using %SYSFUNC to Place a Date in a Title**

```
data _null_;
today = put(date(), worddate18.);
call symputx('dtnull', today);
run;

title1 "Using Automatic Macro Variable SYSDATE &sysdate";
title2 "Date from a DATA _NULL_ &dtnull";
title3 "Using SYSFUNC %sysfunc(date(), worddate18.)";
```

The three titles will correctly display the three dates, however the use of %SYSFUNC in the TITLE3 statement removes the need for the DATA \_NULL\_ step and simplifies the code.

```
Using Automatic Macro Variable SYSDATE 11FEB15
Date from a DATA _NULL_ February 11, 2015
Using SYSFUNC February 11, 2015
```

Notice that the usage of %SYSFUNC in TITLE3 does not create and use a macro variable. From the macro language perspective the call to %SYSFUNC is not a complete statement, and that is OK! The macro facility is quite happy to process code fragments like this one. It is the resolved code that ultimately matters. Actually we are quite used to seeing macro variable names being processed, and the name, such as &DTNULL, is not a complete macro statement either.

In the titles shown above you may be able to discern some leading spaces before the date in the second and third titles. These are caused by the date string being right-justified. While not an issue in most ODS destinations, if you need to remove the leading spaces, you can use the LEFT and TRIM functions. However, if you are not careful, you may encounter a couple of problems.

The first problem is that function calls cannot be nested within %SYSFUNC. Fortunately, this is rather easily handled because you can nest %SYSFUNC requests.

Secondly, the resolved values of interior calls to %SYSFUNC are used as arguments to the outer calls. When the resolved value contains special characters, especially commas, they can be misinterpreted. The following revised TITLE3 will not work because the interior %SYSFUNC results in a formatted value that contains a comma:

```
title3 "Using SYSFUNC %sysfunc(left(%sysfunc(date(), worddate18.)))";
```

After the inner %SYSFUNC is executed, the result is passed to the LEFT function:

```
title3 "Using SYSFUNC %sysfunc(left( February 11, 2015))";
```

Because of the comma in the date, the LEFT function will see two arguments (it is expecting exactly one), and the message “too many arguments” is generated.

You can use the %QSYSFUNC function to mask special characters in the resolved text string that is passed to the outer function. Using %QSYSFUNC masks the comma in the date and eliminates the problem in the LEFT function:

```
title3 "Using SYSFUNC %sysfunc(left(%qsysfunc(date(), worddate18.)))";
```

## Using %SYSFUNC to Delete a File

The macros in Programs 7.4.2d and 7.4.2e have been adapted from an example that can be found on p. 141 of *SAS 9.1 Macro Language: Reference, First Edition* (2004). The macro %DELFILE can be used to delete a file (c:\temp\agefreq.xml). More important than these examples themselves is the demonstration that, in conjunction with %SYSFUNC, you can use DATA step functions that would only be seldomly used in the DATA step, but that are extremely useful when used with the macro language.

### Program 7.4.2d: Using %SYSFUNC to Delete a File

```
%macro delfile(fname=);
%local filrf rc;
%let filrf=__fref; ①
%if %sysfunc(fileexist(&fname)) %then %do; ②
  /* The file exists, establish the fileref;
  %let rc=%sysfunc(filename(filrf,&fname)); ③
  /* Delete the file;
  %let rc=%sysfunc(fdelete(&filrf)); ④
  /* Clear the fileref;
  %let rc=%sysfunc(filename(filrf)); ⑤
%end;
```

```
%else %put File Not Found;
%mend delfile;

%delfile(fname=c:\temp\agefreq.xml) ⑥
```

- ① A name for a temporary *fileref* is assigned to the macro variable &FILRF.
- ② The name of the file to be deleted has been passed into the macro as the parameter &FNAME, and the FILEEXIST function is used to determine if this file exists. If it does, then it is to be deleted.
- ③ The FDELETE function ( ④ ) requires the use of a *fileref* and this is established using the FILENAME function. When used with %SYSFUNC the FILENAME function expects the first argument to be a macro variable; hence, you would get an error if you used an ampersand as in:

```
%let rc=%sysfunc(filename(&filrf,f:\mystuff\biomass.raw));
```

Had we used an ampersand, as in the previous statement, &FILRF would resolve to \_\_FREF, and SAS would look for a macro variable with that name (which it will not find).

The documentation is not clear as to which functions expect macro variable arguments to be specified without the ampersand when they are used with %SYSFUNC. Fortunately, few functions do.

- ④ The FDELETE function deletes the file associated with the *fileref* named by the macro variable &FILRF.
- ⑤ Once the file has been deleted, we clear the *fileref*.
- ⑥ The name of the file, including the physical path is passed into the macro as a named parameter.

Program 7.4.2e is another variation of %DELFILE. Rather than using the FILEEXIST function ( ② ), the FILENAME function along with the FEXIST function is used to detect that the file does indeed exist.

#### Program 7.4.2e: Using %SYSFUNC and the FEXIST Function

```
%macro delfile(fname=);
  %local filrf rc;
  %let filrf=__fref;
  /* Establish the fileref;
  %let rc=%sysfunc(filename(filrf,&fname)); ⑦

  /* Delete the file if it exists;
  %if &rc = 0 and %sysfunc(fexist(&filrf)) %then ⑧
    %let rc=%sysfunc(fdelete(&filrf));
  %else %put File Not Found;

  /* Clear the fileref;
  %let rc=%sysfunc(filename(filrf));
%mend delfile;

%delfile(fname=c:\temp\agefreq.xml)
```

- ⑦ When the path specified in &FNAME does not exist the return code &RC will be nonzero.
- ⑧ The use of the FEXIST function actually checks that the *fileref* points to an existing file. Unlike the FILEEXIST function the FEXIST and FDELETE functions expect a *fileref* as the argument, not a macro variable, consequently an ampersand appears in the argument.

**SEE ALSO:** Langston (2015b) demonstrates the use of FDELETE.

### 7.4.3 Taking Advantage of Less Commonly Used DATA Step Functions

The availability of %SYSFUNC and %QSYSFUNC in the macro language opens numerous possibilities for the macro programmer. Although a great many of the DATA step functions will rarely if ever be used

in the DATA step, many of these same DATA step functions are like gold to the macro programmer. Although you might never find a need for the FILENAME function in the DATA step, Programs 7.4.2d and 7.4.2e demonstrate a clear need for this function as well as FEXIST, FDELETE, and FILEEXIST in the macro language. To be a good macro programmer you absolutely must know your DATA step functions. And you must learn them not only from the DATA step perspective, but also from the macro language perspective. The examples in this section demonstrate the use of some additional DATA step functions that you might find especially useful when programming in the macro language.

## Using the PATHNAME Function

The following macro uses the PATHNAME function to retrieve the path that is associated with a *libref*. This path is then used to build a new *libref* with a different engine. Since the LIBNAME statements include an ENGINE option, the copied data sets will be rewritten using the new engine. If the new engine is one associated with SAS/ACCESS, the output data set might even be converted to a format other than SAS.

### Program 7.4.3a: Using PATHNAME with %SYSFUNC

```
%macro engchng(libref=sasuser,
              engine=v6,
              dsn=);
  * libref - library containing the data set(s) of interest;
  * engine - output engine for this &dsn
  * dsn    - name of data set to copy
  *;

  * Create a libref for the stated Engine;
  libname dbmsout clear;
  libname dbmsout &engine "%sysfunc(pathname(&libref))"; ①

  * Copy the SAS data set(s) using the alternate engine;
  proc datasets nolist; ②
    copy in=&libref out=dbmsout;
    select &dsn;
    run;
    quit;
  libname dbmsout clear;
%mend engchng;

*****;
%engchng(libref=macro3,engine=excel,dsn=biomass clinics)
```

- ① The PATHNAME function is used to return the physical path, which is associated with an established *libref* (&LIBREF). A new *libref* (DBMSOUT) is then established using the alternate engine (&ENGINE) and the original location.
- ② PROC DATASETS uses the new engine in the OUT= option to copy the selected member(s) listed in &DSN.

Notice that the *libref* (&LIBREF) in the PATHNAME function call is not in quotes (as it would be in the DATA step). Remember that arguments to macro functions are always text, so the quotes are not necessary. As you use DATA step functions with %SYSFUNC, you should expect the behavior of many of the arguments of the DATA step functions to vary slightly in ways such as this.

## Generating PATTERN Statements Using PUTN

In the following example, the macro %PATTERN uses %SYSFUNC with the PUTN function to build a series of PATTERN statements that subdivide the spectrum of gray scales into equal divisions. The PATTERN statements are to be used with a PIE chart, and the macro determines the number of distinct slices that will be needed.

**Program 7.4.3b: Using %SYSFUNC with PUTN**

```
%macro pattern(dsn=,pievar=);
%local g i j nslice;

* Determine the number of unique values of the
* variable that will be used to determine the slices;
proc sql noprint;
select count(distinct &pievar) into :nslice ❸
  from &dsn;
run;

%do j = 1 %to &nslice;
  /* Create &nslice pattern statements;
  %let i=%sysevalf(255/(&nslice+1)*&j,floor); ❹
  %let g=%sysfunc(putn(&i,hex2.)); ❺
  pattern&j v=psolid c=gray&g r=1; ❻
  %end;
  %mend pattern;

%pattern(dsn=macro3.biomass, pievar=station)
```

- ❸ The number of unique values of the plotting variable (&PIEVAR) is saved in &NSLICE.
- ❹ There are 256 (this is  $16^2$ ) potential shades of gray. Since they gradually darken from white to black, they can be separated into &NSLICE equally spaced shadings.
- ❺ Each gray scale is identified using its hexadecimal number. The PUTN function, which is called by %SYSFUNC, performs the decimal to hexadecimal conversion.
- ❻ The HEX value, which is stored in &G, becomes a part of the name of the color.

The generated PATTERN statements can be seen when MPRINT is turned on.

```
MPRINT(PATTERN):   pattern1 v=psolid c=gray13 r=1; ❻
MPRINT(PATTERN):   pattern2 v=psolid c=gray27 r=1;
MPRINT(PATTERN):   pattern3 v=psolid c=gray3A r=1;
MPRINT(PATTERN):   pattern4 v=psolid c=gray4E r=1;
```

The %PATTERN macro has the huge advantage of automatically separating the gray scale spectrum into the correct number of equally spaced shadings. The user does not need to know how many shades are needed or how to calculate the arcane values that the scale uses.

## Rounding Numbers

Although %SYSEVALF does not support a ROUND conversion type, rounding can be achieved by using the ROUND DATA step function with %SYSFUNC. The following statement will round the arithmetic result to the nearest tenth:

```
%let r = %sysfunc(round(%sysevalf(7/3), .1));
```

**MORE INFORMATION:** The view SASHELP.VSLIB also contains path information, and you can use it instead of the PATHNAME function. The SASHELP views are discussed in Section 11.2.1.

The %SLEEP macro in Section 7.6.1 uses nested %SYSFUNC calls to work with datetime constants.

**SEE ALSO:** You can find a summary description and the syntax for DATA step functions that you can use with %SYSFUNC and %QSYSFUNC in Appendix 4, in *SAS 9.4 Macro Language: Reference, Fourth Edition*.

A short introduction to %SYSFUNC using functions such as FILENAME, DOPEN, DCLOSE, DNUM, DREAD, PUTN, and PUTC is provided by Kenney (1999).

Yindra (1998) includes numerous examples that use %SYSFUNC in a variety of interesting and very readable macro tools. Murphy (2003b) uses %SYSFUNC to gather information about the metadata (number of observations, number of variables, variable names, types of variables) of a data set.

Several examples are shown by Yu (1998) that use %SYSFUNC to read and write external files, while Chen (2003) discusses a number of ways to read external files. Lund (2003b) uses %SYSFUNC to return a number of external file attributes.

%SYSFUNC is used with the PUTN function to calculate hexadecimal to decimal conversions in Watts (2003a).

Burlew (1998, p.127), Stokke (2000); Izrael, Hoaglin, and Battaglia (2000), Mao (2003), Lund (2003b), and Hamilton (2000) each use %SYSFUNC with a GETOPTION function. Rook and Yeh (2001) use it with the PATHNAME, FEXIST, and FDELETE functions.

Teberg (2002) applies the %SYSFUNC function to the INTNX function to work with dates of files.

The date example used to introduce %SYSFUNC above is expanded in Carpenter (2000b). Palmer (2001) also uses %SYSFUNC to augment titles and footnotes with dates.

A variety of DATA step functions, such as VARTYPE, VARNAME, and FILEEXIST, are used with %SYSFUNC and %QSYSFUNC by Olaleye (1998), Borgerding and Chai (2000), Wang (2003), and Bramley (2001).

## 7.5 Building Your Own Macro Functions

The SAS macro language includes a number of macro functions, such as %UPCASE, %QSCAN, and %SUBSTR. In addition, a number of autocall macros (see Section 10.6), such as %LEFT, %TRIM, and %VERIFY, are also supplied that act like macro functions. Macro functions are especially useful as programming tools because the function call is replaced directly by the result of the call. Many macro programmers are unaware that they too can also write useful macro functions.

Fortunately, macro functions are not difficult to program—not difficult, that is, if you know the simple techniques required to turn a macro into a macro function. This section shows you how to create a macro function, the statements that you must avoid, and the technique used to “pass” the result of the function back to the calling program.

### 7.5.1 Introduction

Macros are often used as programming tools to answer specific questions about the programming environment. A part of this process is passing information back from the macro that determines the information to the macro that needs it. A typical macro solution is to have the macro create one or more macro variables that are added to the global macro symbol table. Since these variables are then available throughout the SAS session, they can later be accessed as required. The disadvantage of this approach is that the macro variables must be global and therefore the programmer risks collisions with other global macro variables that might be in use by the program or application. A further disadvantage is the need to use macro variables to pass information in the first place. These disadvantages can often be eliminated by converting the macro into a macro function.

The following macro, which is a variation of a macro that was published in the *SAS Guide to Macro Processing, Version 6*, can be used to determine if a data set exists (see Program 7.1.3a for more explanation of a similar macro). It uses a DATA \_NULL\_ step, the automatic macro variable &SYSERR, and a global macro variable (&EXIST). When the macro is called, the DATA step is compiled including the SET statement. During the DATA step’s compilation phase, SAS determines if the data set (&DSN)

exists, and if it does not exist, the macro variable &SYSERR will be loaded with a value other than zero. Because of the STOP statement, no data are actually read during DATA step execution.

#### Program 7.5.1: Testing for the Existence of a Data Set

```
%macro exist(dsn);
  %global exist;
  %if &dsn ne %then %do;
    * An unknown data set causes a
    * compile error that is reflected
    * in the SYSERR macro variable;
    data _null_;
      stop;
      set &dsn;
      run;
  %end;
  %if &syserr=0 %then %let exist=YES;
  %else %let exist=NO;
%mend exist;
```

After the macro has been called, branching and execution decisions can be made based on the value stored in the global macro variable &EXIST. For example, assume that in a DATA step a user wants to conditionally execute a block of code depending on the existence of the data set SASUSER.BIGDAT. The macro call and associated statements might be as follows:

```
%exist(sasuser.bigdat)

data .....,;
set.....;
if "&exist"="YES" then do;
...more SAS statements...
```

The macro %EXIST is first called (this creates and loads the global macro variable &EXIST) and then the global macro variable &EXIST is later tested. The user is responsible for not having another global macro variable named &EXIST; the user must know that the macro variable created by the macro is named &EXIST, and that &EXIST will take on the values of YES and NO. The macro works, and it is reasonably efficient, but it is NOT a macro function. The discussion in Section 7.5.2 demonstrates how this macro differs from a macro function, and what needs to be done in order to convert it into a macro function.

---

#### 7.5.2 Building the Function

In order for a macro to mimic a macro function, it must be usable within macro statements, and it must be able to build base language code, including macro code, directly. In Program 7.5.1, the macro %EXIST *must* be called outside of the DATA step. Since %EXIST generates its own DATA step, the following macro call would cause problems in the DATA step logic. This macro therefore does not mimic a macro function. Execution of %EXIST within a DATA step would cause errors. Essentially this code is attempting to embed one DATA step within another.

```
data .....,;
set.....;
%exist(sasuser.bigdat)/* causes an error */
if "&exist"="YES" then do;
...more SAS statements...
```

#### Macro Function Attributes

Creating a macro that mimics a macro function is fairly straightforward if you understand the three simple attributes that a macro function must have:

- All statements in the macro must be macro statements.
- The macro should create no macro variables other than those that are local to that macro.
- The macro should resolve to the value that is to be returned.

Remember that a function generally returns a value. In the %EXIST macro in Program 7.5.1 this value was transferred out of the macro through the use of the global symbol table. However, the second rule restricts us to macro variables with a local scope. Understanding how a value is to be passed out of a macro while restricting macro variables to a local scope, is crucial to understanding how macro functions operate. For many macro programmers, not comprehending how to do this prevents them from writing effective macro functions.

Let's look at each of these three rules.

### **Use Only Macro Statements**

The %SYSFUNC macro function is a wonderful tool when building your own macro functions. It enables you to use DATA step functions without using the DATA step. In the Program 7.5.1 version of the macro %EXIST the DATA step is used to ascertain whether the data set named in &DSN exists. Since the DATA step reads no data, creates no data, and is only used to build a macro variable, it can be completely replaced with a call to %SYSFUNC. The following version of the macro %EXIST eliminates the use of the DATA step by using the EXIST DATA step function.

#### **Program 7.5.2a: Using the EXIST Function**

```
%macro exist(dsn);
%global exist;
/* Check if &DSN has been created;
%if %sysfunc(exist(&dsn)) %then %let exist=YES;
%else %let exist=NO;
%mend exist;
```

Since this version of %EXIST does not use a DATA step, the macro call could now be placed within the same DATA step that uses the &EXIST macro variable. Although the macro call for Program 7.5.1 from within the DATA step caused problems in the example at the start of Section 7.5.2, the following code, which uses the Program 7.5.2a version of %EXIST, will now work:

```
data ....;
set....;
%exist(sasuser.bigdat)
if "&exist"="YES" then do;
...more SAS statements...
```

Notice that even the comments in the macro function should be macro comments. In this usage of %EXIST it really does not matter whether the comment is a macro comment, but as we continue to build the macro function, it becomes crucial that we use only macro-style comments.

### **Create Only Local Macro Variables**

Since the previous version of %EXIST still creates a global macro variable (&EXIST), the macro is still not a macro function. The %LOCAL statement causes macro variables to be assigned to the local symbol table, and should be used for all of the macro variables that are defined within the macro function. Use of the %LOCAL statement ensures that there will be no collisions between macro variables defined within the macro and those that might already exist on the global symbol table.

The problem, of course, is that these local macro variables cannot be accessed outside of the macro. How then does one pass values out of a macro without creating global macro variables? The answer to this question is at the heart of the solution of how to build macro functions.

### **The Macro Call Resolves to the Value to Be Returned**

Rather than creating a macro variable that is made available later (often through the global symbol table), we can let the call to the macro resolve to the value of interest. This way the %EXIST (SASUSER.BIGDAT)

macro call will be literally replaced in the code stream by whatever value is to be tested. In the following version of %EXIST the macro call will be replaced by either YES or NO.

#### Program 7.5.2b: Returning Text

```
%macro exist(dsn);
  /* Check if &DSN has been created;
   %if %sysfunc(exist(&dsn)) %then YES;
   %else NO;
%mend exist;
```

Notice that the action for the %IF-%THEN statement (YES) and the %ELSE statement (NO) contains only text. Since a YES or NO is not a macro statement, it cannot be executed by the macro processor (as a %LET statement would be), and it is essentially just “left behind.” When the macro expression is true (%SYSFUNC and the EXIST function find the data set named in &DSN), the whole %IF-%THEN/%ELSE resolves to YES. Since this is not a macro statement, it is effectively passed on to the base language.

Now we can ask the DATA step IF-THEN question by calling the %EXIST macro from within the IF statement. The macro call will be replaced by either YES or NO:

```
data ....;
set....;
if "%exist(sasuser.bigdat)"="YES" then do;
...more SAS statements...
```

If the data set SASUSER.BIGDAT exists the macro call in the IF statement is replaced with the word YES:

```
data ....;
set....;
if "YES"="YES" then do;
...more SAS statements...
```

When we can anticipate the behavior of the functions that are called from within the macro by %SYSFUNC, we can write even more sophisticated macro functions. The DATA step function EXIST, for instance, returns a nonzero positive value when the data set is located. That means, of course, that we can test for that value directly. In Program 7.5.2b this test was made using a %IF within the macro.

Program 7.5.2c takes this one step further by performing the check outside of the macro rather than within it, and this eliminates the %IF-%THEN/%ELSE. This version of %EXIST contains only macro statements and it establishes no macro variables, neither global nor local. It further reduces what the user needs to know in order to use the macro by eliminating the YES/NO and, in the process, the need for the macro %IF-%THEN/%ELSE statement is also eliminated.

#### Program 7.5.2c: Returning a True or False Value

```
%macro exist(dsn);
  /* The following sysfunc call results
   * in a non-zero value when the data
   * set exists;
   %sysfunc(exist(&dsn))
%mend exist;
```

It is the last line of the above macro that establishes it as a macro function. The call to %SYSFUNC will execute the EXIST function. The result of that execution will be a number, that is, 0 or 1. Since this value is not a macro statement, it will be left behind (passed out of the macro), and this value can then be tested for directly by the calling program.

Writing your macro functions this way enables you to use the macro as part of either a DATA step statement or a macro statement. In a DATA step IF-THEN/ELSE we might compare the returned value to 0 (false):

```
if %exist(sasuser.bigdat) ne 0 then do;
```

Or better yet, just take advantage of the fact that the %EXIST macro will always return a true or false:

```
if %exist(sasuser.bigdat) then do;
```

If the data set exists, the comparison in the IF-THEN statement reduces to a 1:

```
if 1 then do;
```

And, of course, 1 is true.

When used in the DATA step, the %EXIST macro is a bit silly since it could have been coded directly using the EXIST function and without the macro language at all:

```
if exist(sasuser.bigdat) then do;
```

This of course would miss the point of the discussion - that building macro functions need not be difficult and that following a few simple rules is all that is required to make functions that can simplify your programming tasks.

### 7.5.3 Using the Function

The first definition of the %EXIST macro shown in the “Introduction” (Program 7.5.1) has several limitations that have already been discussed. In addition, this macro might not be usable in some macro environments. Again, this is a direct consequence of the fact that it must run a DATA step before it can build the macro variable &EXIST. This means that this macro could never be used in a macro that was itself being used as a macro function. Furthermore, the user would run the risk that the sequence of the execution of macro statements and DATA steps might cause problems (base language statements are executed after macro language statements and consequently the macro variable &EXIST might not have been created when it is needed).

Excluding the non-macro statements and building a macro function, such as the one shown in the final version of %EXIST (Program 7.5.2c), enables its use in both the macro language and the base SAS language. As a macro function %EXIST can also be used in macro language statements such as the following:

```
%if %exist(sasuser.bigdat) %then %do;
```

Because of this flexibility the macro function becomes a building block that can be used in the construction of other macro functions. For instance, a function designed to return the number of observations in a data set would first need to ascertain whether the data set exists.

Of course the final version of the %EXIST macro has become so simplified that we might as well have used the %SYSFUNC directly, and in the process created code that is a bit more efficient:

```
%if %sysfunc(exist(sasuser.bigdat)) %then %do;
```

While a macro programmer familiar with %SYSFUNC and the EXIST function would indeed probably not go to the trouble of creating the %EXIST macro, there might be other programmers in your group who are not that knowledgeable. By creating this and other macro tools, and by placing them in a macro library, you can expand the capabilities of everyone in your group—especially those who might not yet understand the finer points of creating macro functions.

**MORE INFORMATION:** Section 7.6 has a number of macro function examples. The AUTOCALL macros described in Section 10.6 are written as macro functions.

**SEE ALSO:** Much of the material in this section was taken from Carpenter (2002). Pete Lund (2000b, 2000a, 2001a, 2001b, and 2001c) has a series of clever tools written as macro functions. Larsen (2001) demonstrates a macro function to center text within a PUT statement. Hamilton (2001) discusses a macro function that returns the number of observations in a data set after first checking to see if the data set exists.

### 7.5.4 Returning a Value

In most of the macro functions in this book the value to be returned is “left behind” by the macro. Very often this takes the form of a macro language fragment, such as a macro variable or a macro function call.

In Program 7.5.2c the macro language function %SYSFUNC is used to form this fragment:

```
%sysfunc(exist(&dsn))
```

Clearly this is not a statement, and the macro language is quite happy to process this code fragment. Remember that the macro facility handles code snippets like this all the time. It is the returned value and how that returned value is used that will make the resultant code successful.

As you study the examples of macro functions in Section 7.6 and elsewhere in this book, watch how the value is returned. Very often macro programmers who are attempting to learn how to write functions, get confused on this point, and by failing to pick up on this key concept, they miss out on one of the primary advantages of writing macro functions.

In Programs 7.5.1 and 7.5.2a the global symbol table is used to pass a value out of a macro. Discussion around those examples shows why this is a less than ideal solution to the problem of passing a value out of the macro. That does not make this type of solution wrong, but certainly less than ideal. Another less common, but similar solution is to take advantage of the arcane macro variable assignment rules (see Section 14.5 for more on these rules). When a macro variable is created in a macro where the %LOCAL statement is not used, and the macro variable already exists in a higher table, the new value will be written to the higher table. This is demonstrated in Program 7.5.4, which defines &name in the current symbol table and then redefines the same macro variable in a macro (%HIGHER).

#### Program 7.5.4: Passing Back to the Next-Higher Table

```
%macro higher;
  %let name=fred;
  %mend higher;

%let name=Susie;
%higher
%put &=name;
```

Examination of the SAS Log shows that the macro variable &name defined in the %HIGHER macro is written, not to the local table, but to the next higher table. You will sometimes see macro programmers take advantage of these assignment rules when passing values out of the macro. The SAS Log shows that the final value of &NAME is the value assigned in the macro %HIGHER.

```
65  %let name=Susie;
66  %higher
67  %put &=name;
NAME=fred
```

**MORE INFORMATION:** The macro %GETAUTOPATH in Program 10.4.2d uses this technique to return a path string from within a macro.

## 7.6 Other Useful User-Written Macro Functions

When learning how to perform a task it is sometimes helpful to see some simple or otherwise interesting examples. This section includes a number of macro functions that should provide some insight into how you can go about creating your own macro functions.

### 7.6.1 One-Liners

Macros that mimic functions can sometimes be written in a single line of code. Sometimes known as *one liners*, not only can these macros further demonstrate the techniques used to write macro functions, they can also serve to remind us that the macro facility does not necessarily require complete statements. Macro expressions, function calls, and macro variables can all be executed and resolved even when they are not a part of a larger statement. Indeed, it is this capability that allows us to *pass back* values from macro functions. While you are examining the macros in this section, take time to notice and understand how these code fragments are resolved and executed and how the result is passed out of the macro.

#### Does the Data Set Exist?

The %EXIST function shown in Program 7.5.2c (shown here without the comments) returns a nonzero value if the data set (&DSN) exists. Notice that the call to the %SYSFUNC function stands alone. It is not part of a statement and there is no semicolon. When the %SYSFUNC executes the EXIST function, the result will be non-macro code, and the macro call will effectively resolve to the value to be passed back.

##### Program 7.6.1a: Does the Data Set Exist?

```
%macro exist(dsn);
  %sysfunc(exist(&dsn))
%mend exist;
```

Although it would be easier and incrementally more efficient to use the EXIST function directly if we are executing in a DATA step, this macro function could be used as part of either a DATA step statement or a macro statement. In a %IF-%THEN/%ELSE we might use the macro call directly:

```
%if %exist(sasuser.bigdat) %then %do;
```

#### Calculating a Factorial

The factorial of a number is the number multiplied by each integer between itself and one. Five factorial (often symbolized as 5!) would be  $5*4*3*2*1 = 120$ . The factorial calculation is often needed when calculating combinations and permutations of groups.

There are three functions that deal directly with the calculation of factorials:

##### FACT

calculates factorials

##### COMB

calculates combinations

##### PERM

calculates permutations.

Each of these functions can now also be used in a macro function. The %FACT macro calls the FACT function, as shown in Program 7.6.1b.

##### Program 7.6.1b: Calculating a Factorial

```
%macro fact(n);
  %sysfunc(fact(&n))
%mend fact;
```

The related calculation of the number of combinations ( $n$  things taken  $r$  at a time) uses the COMB function, as shown in Program 7.6.1c.

#### Program 7.6.1c: Calculating a Combination

```
%macro comb(n,r);
  %sysfunc(comb(&n,&r))
%mend comb;
```

**MORE INFORMATION:** The PERM function can also be used to calculate factorials as well as the number of permutations. A macro that uses the PERM function is shown in Program 7.6.2a.

### Working with Dates

In Program 7.4.2c %SYSFUNC was used to add the current date to a title. In that program the suggested solution was to use two %SYSFUNC function calls.

```
title3 "Using SYSFUNC %sysfunc(left(%qsysfunc(date(),worddate18.)))";
```

The process can be simplified by creating a macro to do the same job. In Program 7.6.1d, the %QLEFT and %QTRIM autocall macro functions are used to return a left-justified date.

#### Program 7.6.1d: Left-Justified and Trimmed Date

```
%macro currdate(fmt);
  %qtrim(%qleft(%qsysfunc(date(),&fmt)))
%mend currdate;
```

Because %CURRDATE is a macro function, it can be used in the TITLE statement directly:

```
title3 "Current date is %currdate(worddate18.)";
```

In this macro each of the called functions returns a quoted result. This masks the comma returned by the WORDDATE. format. If left unmasked, the comma would have caused the %LEFT and %TRIM functions to fail. In this example, the %TRIM could have been used instead of %QTRIM as the date string is being used in a title. By using %QTRIM the comma in the final string is masked, and this allows %CURRDATE to be used in situations other than just in titles.

Like %CURRDATE, the %DB2DATE macro in Program 7.6.1e builds a text string based on the current date; however, it is built in a form that can be utilized in an SQL pass-through to a DB2 table.

The macro resolves to the quoted date string in the form of “YYYY-MM-DD-00.00.00”, which DB2 will interpret as a datetime value as of midnight on today’s date.

#### Program 7.6.1e: Returning a DB2 Datetime Value

```
%macro db2date;
  %unquote(%bquote(')%sysfunc(date(),yymmddd10.)%bquote(-00.00.00'))
%mend db2date;
```

The following SQL step shows how this macro might be used in an SQL pass-through to DB2:

```
proc sql;
  connect to odbc (&db2j);
  create table sincemidnight
    as select * from connection to odbc (
      select *
        from mydba.hospital1
        where proddate > %db2date
        for fetch only);
```

```
%put &sqlxmsg;
disconnect from odbc;
quit;
```

When executed on December 21, 2015, the WHERE clause becomes the following

```
where proddate > '2015-12-21-00.00.00'
```

The DB2 expects this datetime value to be in quotes. These are included in the returned string through the use of the %BQUOTE quoting function. The quoting function masks the single quote until after the %SYSFUNC has executed. Instead of using the %BQUOTE function, the macro %DB2DATE could have been written using the %STR quoting function. However, since %STR does not mask the single quote ('), it must use a special masking character ( % ) to precede special characters to be masked that would not otherwise be masked.

```
%macro db2date;
  %unquote(%str(%')%sysfunc(date(), yymmdd10.)%str(-00.00.00%'))
%mend db2date;
```

**MORE INFORMATION:** The use of the % as a masking character in the %STR function is described in Table 7.1.9. In Program 12.1.2e the %TSLIT autocall macro function is used to avoid the problems associated with having macro language elements surrounded by single quotes.

**SEE ALSO:** Lund (2003b) has a macro that returns the current date.

## Building a Logical Expression

The macro function %FUZZRNGE can be used to create a logical expression based on the parameters which are a base value and a displacement value. An expression is then built that will check for values within the inclusive range of the base value  $\pm$  the displacement value.

If we needed to build an expression such as (5 le age & age le 9) in an IF statement. The range for the variable AGE is  $7 \pm 2$ .

The statement

```
if %fuzzrnge(age, 7, 2) then do;
```

becomes

```
if (5 le age & age le 9) then do;
```

Three values are passed into the macro %FUZZRNGE: the variable name (&VAR), the base value (&BASE), and the displacement (&DISP).

### Program 7.6.1f: Building a Logical Expression

```
%macro fuzzrnge(var,base,disp);
  (%eval(&base-&disp) le &var & &var le %eval(&base+&disp))
%mend fuzzrnge;
```

Because this macro uses the %EVAL function, both explicitly and implicitly, it will work only for integer values.

**MORE INFORMATION:** A macro language caveat associated with composite expressions of a range of values is discussed in Section 14.3.6.

## Putting the Computer to Sleep

The macro %SLEEP uses the SLEEP function (available on the Windows and UNIX operating systems) and can be used to pause a SAS program for a specified time. The number of seconds that SAS is to be suspended is noted in the &TIME parameter. Because the execution of the SLEEP function itself is the desired result, this function does not pass any value back to the calling program.

### Program 7.6.1g: Suspending the Execution of SAS

```
%macro sleep(time=5);
  %local rc;
  %let rc = %sysfunc(sleep(&time));
%mend sleep;

%sleep(time=60)
```

The version of %SLEEP in Program 7.6.1g will suspend execution of SAS for about 1 minute. The DATA step SLEEP function returns the number of seconds of sleep, and this value has been stored in the local macro variable &RC (which is ultimately not used). You could make use of this returned value by passing it back to the calling program.

### Program 7.6.1h: Using the Value Returned by the SLEEP Function

```
%macro sleep(time=5);
  %sysfunc(sleep(&time))
%mend sleep;

%put WARNING: Computer slept for %sleep(time=15) seconds;
```

A slightly more sophisticated version of %SLEEP enables the user to specify the units of time to suspend the execution of SAS.

### Program 7.6.1i: Naming Time Units

```
%macro sleep(time=5,unit=seconds);
  /* Units can be seconds, minutes, or hours;
  %local timeval;

  %if &unit eq %then unit=seconds;
  %if %upcase(&unit)=SECONDS %then %let timeval=1;
  %else %if %upcase(&unit)=MINUTES %then %let timeval=60;
  %else %if %upcase(&unit)=HOURS   %then %let timeval=3600;

  %sysfunc(sleep(&time,&timeval))
%mend sleep;

%put WARNING: Computer slept for %sleep(time=1,unit=minutes) seconds;
```

The %PUT will indicate that the SAS session was suspended for 60 seconds:

```
156 %put WARNING: Computer slept for %sleep(time=1,unit=minutes) seconds;
WARNING: Computer slept for 60 seconds
```

Notice that this macro takes advantage of the second optional argument of the SLEEP function which acts like a multiplier of the first argument.

Rather than tell SAS how long to sleep you may want to tell it a time to wake up from a suspended session. The %WAKEUP macro also uses the SLEEP function; however, rather than passing it the duration of inactivity, it accepts the SAS datetime value at which it will wake up. The parameter must be in a datetime18. form. The current time is subtracted from the wake-up time, which is converted to a SAS

datetime value using the standard datetime constant conversion (dt). Notice the use of the nested calls to the %SYSEVALF function.

#### Program 7.6.1j: Wakeup SAS at a Specified Time

```
%macro wakeupat(time=);
  %local rc;
  %let rc = %sysfunc(sleep(
    %sysevalf(%sysevalf("&time"dt)-
      %sysfunc(datetime()))));
%mend wakeupat;
```

The following macro call causes SAS to “sleep” until the afternoon of the third of March in 2016:

```
%wakeupat(time=03mar2016:16:51:00)
```

#### Retrieve the Last Word in a List

The %LISTLAST macro returns the last word in the argument. The list is reversed making the last word the first. This is then retrieved using the %SCAN and the selected word is then put back into the original order using a second call to the REVERSE function. Notice that each call to REVERSE is through a %SYSFUNC, and that the %SYSFUNC calls are nested.

#### Program 7.6.1k: Retrieving the Last Word in a List of Words

```
%macro listlast(list);
  %sysfunc(reverse(%scan(%sysfunc(reverse(&list)),1,%str( ))))
%mend listlast;
```

Although Program 7.6.1k is a great example of nested calls to %SYSFUNC, and while you should understand what it is doing, it is much more complicated than is necessary. In the current versions of SAS, when the second argument of the %SCAN function is negative, the %SCAN function works from right to left, rather than from left to right (the b modifier could also be used – see Program 7.2.3f). The %LISTLAST macro could therefore be specified as in Program 7.6.1l by using a negative word number.

#### Program 7.6.1l: Using %SCAN with a Negative Word Number

```
%macro listlast(list);
  %scan(&list,-1,%str( ))
%mend listlast;
```

**MORE INFORMATION:** The %SCAN function is discussed in more detail in Section 7.2.3.

**SEE ALSO:** Pete Lund (2003a) uses the %SCAN function with a negative word number to select the last word in a list.

#### Searching for Words

The macro function %INDEX searches for occurrences of the second argument. It does not, however, take notice of word boundaries. The %INDEXW macro function mimics the INDEXW function in the DATA step and returns the position of the word specified in the second argument.

#### Program 7.6.1m: Mimicking the INDEXW Function

```
%macro indexw(list,wrd);
  %sysfunc(indexw(&list,&wrd))
%mend indexw;

%let mylist=concatinated catalonia cat club;
%let myword=cat;
%put &myword is at position %indexw(&mylist,&myword);
```

The SAS Log shows that the function returns the starting position (24) of the word cat. Positions of embedded instances of ‘cat’ are not returned:

```
29  %put &myword is at position %indexw(&mylist,&myword);
cat is at position 24
```

**MORE INFORMATION:** The %REVSCAN function in Section 7.6.3 returns the word number rather than the position of the word.

## Converting Number Systems

Although we typically use the decimal number system for most of our calculations, SAS and other languages often use others. Of these, the binary and hexadecimal system are very common. The %RGBHEX macro can be used to convert three RGB (Red/Green/Blue) color components ranging from 0 to 255 in decimal to a single hexadecimal character code needed for making color assignments in SAS.

### Program 7.6.1n: Creating an RGB Hexadecimal color value

```
%macro RGBHex(rr,gg,bb);
%sysfunc(compress(CX
  %sysfunc(putn(%sysfunc(round(&rr)),hex2.))
  %sysfunc(putn(%sysfunc(round(&gg)),hex2.))
  %sysfunc(putn(%sysfunc(round(&bb)),hex2.))))
%mend RGBHex;
```

Source: Perry Watts.

This function uses %SYSFUNC, along with the PUTN and ROUND functions, to do the conversion:

```
43  %put The 50,150,250 RGB color is %rgbhex(50,150,250);
The 50,150,250 RGB color is CX3296FA
```

**MORE INFORMATION:** The macro in Program 7.4.2a uses %SYSFUNC with the PUTN function to perform date conversions.

**SEE ALSO:** Additional conversions and other macros that work with color systems within SAS are presented by Perry Watts in 2003a and 2003b.

---

## 7.6.2 Macro Functions with Logic

The macros in Section 7.6.1 require only a single code fragment. While these are especially useful when explaining how macro functions return values, usually your macro functions will be more complex.

Generally, the macro functions that you write will require the use of logic, %DO-loop processing, and internal calculations. Remember to do as follows:

- Use only code that will be processed by the macro facility (no DATA steps).
- Make all intermediate macro variables local.
- Return the value by “leaving it behind.”

## Calculating Permutations

The macro function %PERM returns the number of permutations within a group or set of objects. In the macro below the number of permutations is based on N things taken R at a time. If the second argument is not used, the function will return the factorial (see Program 7.6.1b); however, in some versions of SAS, if the comma separating the two arguments is included and no second argument is provided, then the function

will return a missing value. The %PERM macro in Program 7.6.2a uses %IF-%THEN/%ELSE processing to detect when the second parameter has not been supplied.

### **Program 7.6.2a: Returning Either the Factorial or the Permutations**

```
%macro perm(n,r);
  %if &r ne %then %sysfunc(perm(&n,&r));
  %else %sysfunc(perm(&n));
%mend perm;
```

The SAS Log can be used to reveal the results of several executions of the %PERM macro function:

```
168  %put perm3X2 %perm(3,2);
perm3X2 6
169
170  %put perm3X1 %perm(3,1);
perm3X1 3
171
172  %put perm3!  %perm(3,);
perm3! 6
173
174  %put perm5X2 %perm(5,2);
perm5X2 20
```

The version of %PERM in Program 7.6.2a will generate errors if it is passed inappropriate arguments. If you are not going to be able to control for inappropriate values, the macro itself should be robust enough to do the checking for you. The version of %PERM in Program 7.6.2b performs these error checks.

### **Program 7.6.2b: Performing Checks on the Arguments of the Function**

```
%macro perm(n,r);
  /* Check for improper values;
   * N & R must be positive integers;
   * N must be > R when R is provided;
   * Calculate the factorial when R is not provided;
  %if &r ne %str() %then %do;
    %if %sysevalf(%sysevalf(&n,integer) ne &n) ①
      or %sysevalf(&n < 1) ②
      or %sysevalf(%sysevalf(&r,integer) ne &r) ①
      or %sysevalf(&r < 1) ②
      or %sysevalf(&r > &n) ③ %then .; ④
    %else %sysfunc(perm(&n,&r));
  %end;
  %else %sysfunc(perm(&n)); ⑤
%mend perm;
```

Since we might have non-integer arguments, %SYSEVALF is used for each of the comparisons.

- ① Check that the argument is an integer.
- ② Make sure that the integer is 1 or larger.
- ③ Verify that *N* is greater than or equal to *R*.
- ④ Verify that the criteria have been met. Failure to meet the criteria results in a missing value.
- ⑤ Calculate the factorial when a value is not provided for &R.

### **Return the Word Number from a List of Words**

Given the word number the %SCAN function returns the selected word from a list. The reverse-scan function (%REVSCAN) shown here returns the word number given the word itself.

**Program 7.6.2c: Return a Word Number from a List of Words**

```
%macro revscan(list, word);
  %local wcnt wnum;
  %let wcnt=0;
  %let wnum=0;
  /* Determine the word number in a list of words;
  %do %while(%scan(&list,%eval(&wcnt+1),%str( )) ne %str() & &wnum=0); ❶
    %let wcnt = %eval(&wcnt+1);
    %if %upcase(%scan(&list,&wcnt,%str( )))=%upcase(&word) %then
      %let wnum=&wcnt;
    %end;
    &wnum ❷
  %mend revscan;
```

- ❶ The %DO %WHILE statement is used to step through the list (&LIST) using the %SCAN function. &WCNT is used to select one word at a time and compare it to the test word (&WORD). The loop is terminated either when the list is exhausted or when the word has been found (&WNUM ne 0).
- ❷ Once the word is found the word number is passed back.

The SAS Log demonstrates the use of %REVSCAN using a list of drug codes, which are held in the macro variable of &DOSECODE.

```
1111 %let dosecode = a1 b1 b2 c b2;
1112
1113 /* qb2 is not found and returns a 0;
1114 %put qb2 %revscan(&dosecode,qb2);
qb2 0
1115
1116 /* b2 is the third word in &DOSECODE and a 3 is returned;
1117 %put b2 %revscan(&dosecode,b2);
b2 3
```

If the word appears more than once %REVSCAN returns the number of its first occurrence; Q7.7.6 (question 6 at the end of this chapter), addresses this “feature.”

**SEE ALSO:** Mao (2003) uses the %LENGTH function with the %SCAN in a %DO %WHILE statement to determine when the last word has been found.

**Repeating Text**

The macro %REPEAT in Program 7.6.2d uses the REPEAT function to generate a series of repeated characters. Blanks can be specified as the repeat character by passing a null value as the first argument. Unlike the DATA step REPEAT function, which places the original character and then repeats it *n* times, the %REPEAT macro places the character(s) exactly the number of times indicated by the second argument.

**Program 7.6.2d: Repeat a List of Characters**

```
%macro repeat(char,times);
  %let char = %quote(&char);
  %if &char eq %then %let char = %str( );
  %sysfunc(repeat(&char,&times-1))
%mend;
```

Source: Pete Lund, Looking Glass Analytics.

The call %REPEAT(abc,3) produces

abcabcabc
-----------

**SEE ALSO:** The %REPEAT macro is used by Lund (2000a) to produce customized comments in the SAS Log.

### Compare Text Strings of Unequal Length

In the DATA step we can use the colon operator to modify how a comparison is to be made. The colon operator compares two strings using the length of the shortest. There is no equivalent operator in the macro language; however, the %COLONCMPR macro shown in Program 7.6.2e makes the same types of comparisons.

The parameters are the two text strings that are to be compared and the comparison operator to use. The macro determines the shorter of the two strings and uses the %QSUBSTR function to truncate the longer of the two. The comparison is made on the uppercase values of the text strings.

#### Program 7.6.2e: Comparing Equal Length Text Portions

```
%macro coloncmp(left,op,right);
  %local width;

  %if &op = %str() %then %let op==; ①

  /* determine shorter of left and right;
  %let width = %sysfunc(min(%length(&left),%length(&right))); ②
  %upcase(%qsubstr(&left,1,&width) &op %qsubstr(&right,1,&width)) ③
  %mend coloncmp;
```

- ① If &OP, the comparison operator, is not specified it is set to equal (=).
- ② The minimum length of the two sides (&WIDTH) is determined. This value is then used to subset the longer string.
- ③ Although %QSUBSTR is applied to both sides it will change only the longer of the two. The %UPCASE function is then used to convert both sides to uppercase. If a case sensitive comparison is desired the %UPCASE should be removed.

The use of this macro is demonstrated through the use of a comparison in the %TRYIT macro:

```
%macro tryit;
  %let a = Smith;
  %let b = sm;
  %if %coloncmp(&a,=,&b) ④ %then %put &a =: &b is true;
%mend tryit;
%tryit
```

- ④ The macro is designed to be called within a %IF statement. For the above example, the comparison in the %IF is true and the SAS Log shows that the %PUT is executed.

```
1221 %tryit
Smith =: sm is true
```

---

### 7.6.3 Functions for the DATA Step

Macro functions return a value based on one or more arguments. Section 7.5 shows you how to build macro functions and there are a number of examples in Sections 7.6.1 and 7.6.2. Most of these macro functions can be used either in macro code or in the DATA step. There are situations, however, when the techniques described in these sections will not work. If you are coding a DATA step and you need to pass the resolved value of a PDV variable into a macro call, such as a macro function, the timing will work against you. The macro will not execute during DATA step execution. The macros shown in this section discuss how to build a macro function that passes back code rather than a result.

While these macros are not, strictly speaking, “macro functions,” the macros in this section mimic functions by building SAS DATA step code. These can be very useful when the macro is only going to be used within a DATA step. By using this approach you can pass back DATA step code; consequently, you are not required to use %SYSFUNC to handle all DATA step functions. While the particular macros shown in this section are limited to the DATA step, the techniques can be used elsewhere within SAS to build code.

Relative to the macro language the point is that these macros build code, which is compiled, and is then available during the execution of the DATA step.

## Incrementing a Date

Have you ever needed to increment a date by a number of years? The macro %ADDYEARS in Program 7.6.3a returns a SAS date that is offset by a user-supplied number of years.

### Program 7.6.3a: Incrementing a Date by Whole Years

```
%macro AddYears(base,add);
  intnx('month',&base,&add*12) + (day(&base) - 1)
%mend;
```

Source: Pete Lund, Looking Glass Analytics.

In this macro the INTNX function is used to advance a date from &BASE by &ADD\*12 months. Because by default the INTNX function returns the start of an interval, the number of days within the current month is then added to the new calculated value. Then since this effectively adds the first of the month twice, one day is subtracted:

```
1281 data _null_;
1282 today=date();
1283 yearfromnow = %addyears(today,1);
MPRINT(ADDYEARS): intnx('month',today,1*12) + (day(today) - 1)
1284 put yearfromnow= date9.;
1285 run;
```

The SAS Log showing a use of this macro on March 10, 2015 demonstrates that the date has been advanced by one year and more importantly that the macro has generated code that contains a call to the INTNX function.

In the current versions of SAS the INTNX function supports the use of alignment options to handle ‘same day’ increments. A simplified version of %ADDYEARS can take advantage of the ‘same’ alignment option:

```
%macro AddYears(base,add);
  intnx('year',&base,&add,'s')
%mend;
```

These two versions of %ADDYEARS are not quite the equivalent. Advancing a leap day (February 29) by a year would result in an invalid date (there will never be two consecutive years with a leap day of 2/29). The first version of %ADDYEARS will return a March 01, while the second will return a February 28.

We could not have rewritten this macro as strictly a macro function. Well, actually it can be rewritten, however its usage would have to change.

```
%macro AddYears(base,add);
  %sysfunc(intnx(year,&base,&add,s),worddate18.)
%mend addyears;

%put %addyears(%sysfunc(today()),1);
```

The use of %SYSFUNC means that this macro will be executed when it is called. It will return a result, but not DATA step code (as do the other versions of this macro). This will work fine when we use it in the macro language; however, using this version of %ADYYEARS in a DATA step will fail:

```

1293 data _null_;
1294 today=date();
1295 yearfromnow = %adyyears(today,1);
ERROR: Argument 2 to function INTNX referenced by the %SYSFUNC or %QSYSFUNC
macro function is
      not a number.
ERROR: Invalid arguments detected in %SYSCALL, %SYSFUNC, or %QSYSFUNC
argument list. Execution of %SYSCALL statement or %SYSFUNC or %QSYSFUNC
function reference is terminated.
MPRINT(ADYYEARS):
1296 put yearfromnow= date9.;
1297 run;

```

The failure is due to the fact that the macro is executed before the variable TODAY exists. It needs a value for TODAY during execution. This is not a problem for the earlier versions of this macro as they do not attempt to resolve TODAY, they just write code that will eventually be executed by the DATA step.

## Grouping Days into Weeks

This macro can be used to create a new variable with a constant value for all observations that fall within the same 7-day interval. For this example we want to be able to group all samples collected within a given sampling week. The desired sampling week runs from Saturday to Friday, and any sample that is taken within that period will be considered to have been taken on the same date.

The start and end days for the week interval is specified by using a shift operator on the interval specification in the INTNX function. A shift value of 7 would result in an interval of ‘week.7’, which specifies a week starting on Saturday and ending on Friday.

### Program 7.6.3b: Creating a Grouping Variable Based on the Week

```

*****
Macro EOW          Garth Helf  24 October 2003
Create new variable in a DATA step containing week
ending date for a SAS date variable. Argument we_day
is day the week starts:
 1=Sunday (default) to 7=Saturday.
*****
%macro eow(dayvar=magdate ①, eowvar=magweek ②,
           eowformt=date9., we_day=1 ③);
  &eowvar=intnx("week.&we_day", &dayvar, 0,"end" ④);
  format &eowvar &eowformt;
%mend eow;

```

Source: Garth Helf, Hitachi Global Storage Technologies.

- ① &DAYVAR holds the name of the incoming date variable (default is MAGDATE).
- ② &EOWVAR holds the name of the new variable, which will have a constant date for all dates within a 7-day interval. The default is MAGWEEK.
- ③ &WE\_DAY indicates an interval shift operator. The interval ‘week.2’ will cause the week to start with a Monday rather than a Sunday. If the shift operator has a value of 7, the variable specified by &EOWVAR ② will always be a Friday and all the dates within a given Saturday-to-Friday range will have the same value.
- ④ The END alignment operator requests that INTNX return the date of the end of the week rather than the start of the week. When alignment and interval shift operators are both used, the shift operator is applied first. This means that a week starting on Monday ('week.2') will end on Sunday, and a Sunday date will be returned.

The %EOW macro is used in the following DATA step to create the variable SAMPLEDATE:

```
data biomass;
  set macro3.biomass;
  %eow(dayvar=date, eowvar=sampleddate, we_day=7)
  weekday = weekday(sampleddate);
run;
```

The SAS Log shows that the %EOW macro has written two lines of code into the DATA step:

```
1330  data biomass;
1331    set macro3.biomass;
1332    %eow(dayvar=date, eowvar=sampleddate, we_day=7)
MPRINT(EOW):   sampleddate=intnx("week.7", date, 0,"end");
MPRINT(EOW):   format sampleddate date9.;
1333  weekday = weekday(sampleddate);
1334  run;
```

## 7.7 Testing Your Knowledge with Chapter Exercises

1. Check all answers that are TRUE. Macro text functions can be used to do the following:
  - a. Perform character searches
  - b. Determine the length of an argument
  - c. Scan an argument for specific words
  - d. Translate lowercase characters to uppercase characters
  - e. None of the above
  - f. All of the above
2. (True or False) Quoting functions remove the meaning of specific characters either during macro compilation or at macro execution.
3. (True or False) The %STR quoting function removes meaning from special characters (except % and &) during macro compilation.
4. Among other things the macro %CNTVAR in Program 7.2.3e counts the number of words in a macro string. Modify this macro so that it does the following:
  - o enables the user to optionally pass the delimiter used to separate words
  - o uses a %DO %WHILE statement instead of a %DO %UNTIL statement
  - o correctly handles a string with no words
  - o complies with the rules of macro functions
5. Write a macro function that will return the name portion of a permanent SAS data set of the form of libref.name; for example, %nameonly(SASCLASS.BIOMASS) returns BIOMASS. Your macro solution should use the %INDEX, %LENGTH, and %SUBSTR macro functions.  
**Extra Credit:** Create an alternate macro solution that uses only one function and has only one statement.
6. The macro %REVSCAN (see Program 7.6.2d) returns the word number of the first word in the list that matches the desired word. Rewrite %REVSCAN to return the last occurrence; that is, %REVSCAN(aa b ab b, b) should return a 4 rather than a 2.
7. The %ALLYR macro in Program 5.3.2a uses data sets that include the two-digit year as part of the data set name. That version of the %ALLYR macro will fail for years before 2010. Chapter 5 Question 10 solves the problem by using four-digit years. Rewrite %ALLYR so that it can accept two digit years as parameters, and so that it is robust enough to operate on all years from before 1999 and through the present.

# **Chapter 8: Discovering Even More Macro Language Elements**

<b>8.1 Even More Macro Functions .....</b>	<b>196</b>
8.1.1 Accessing System Environmental Variables Using %SYSGET .....	196
8.1.2 %SYSMEXECDEPTH and %SYSMEXECNAME .....	199
8.1.3 Assessing Macro Existence and Execution Status with %SYSMACEXEC and %SYSMACEXIST .....	200
8.1.4 Determining Product Availability Using %SYSPROD .....	201
8.1.5 Checking Up on Macro Variable Scopes.....	203
<b>8.2 Even More Macro Statements .....</b>	<b>204</b>
8.2.1 Extending the Use of %SYMDEL .....	204
8.2.2 Using the %GOTO and %label Statements Appropriately.....	206
8.2.3 Using %WINDOW and %DISPLAY .....	208
8.2.4 Extending %SYSEXEC with Examples .....	211
8.2.5 Deleting Macro Definitions with %SYSMACDELETE .....	212
8.2.6 Making Macro Variables READONLY .....	213
<b>8.3 Even More Automatic Macro Variables .....</b>	<b>214</b>
8.3.1 Passing VALUES into SAS Using &SYSPARM .....	214
8.3.2 Learning More about Deciphering Errors.....	216
8.3.3 Taking Advantage of the Parameter Buffer.....	220
8.3.4 Using &SYSNOBS as an Observation Counter .....	223
8.3.5 Using &SYSMACRONAME.....	224
8.3.6 Using &SYSLIBRC and &SYSFILRC .....	224
<b>8.4 Even More System Options.....</b>	<b>225</b>
8.4.1 Memory Control Options .....	225
8.4.2 Preventing New Macro Definitions with NOMCOMPILE .....	226
<b>8.5 Even More DATA Step Functions and Statements .....</b>	<b>226</b>
8.5.1 DOSUBL Function.....	226
8.5.2 Deleting Macro Variables with CALL SYMDEL .....	228
8.5.3 Using SYMEXIST, SYMGLOBAL, and SYMLOCAL .....	229

In this chapter a second look is taken at a number of types of macro language elements, such as functions and options that have been introduced throughout this book. Here you will find elements of the macro language that tend to be less commonly used, not necessarily because they are less important, but for the most part, the elements noted in this chapter have a narrower focus and therefore a more limited utility.

As you read through this chapter you will notice that the examples tend to highlight the usage of the element being described. The examples are not intended to be ‘practical’ in and of themselves, but are instead designed to demonstrate certain aspects of the elements being discussed.

For the examples in this chapter and indeed for all of the code examples throughout the book, if you want to execute these sample programs, then be sure to follow the setup instructions. Remember that all of the data sets and programs are available for download, so you do not need to retype either the code or the data.

For instructions on accessing and setting up the programs and data, see the “Example Code and Data” section within this edition’s “About This Book” front matter.

**SEE ALSO:** A number of these newer features are discussed by Langston (2015a).

## 8.1 Even More Macro Functions

Although you will probably not reach for the functions in this section often, when you need them, you will tend to *really* need them. They enable you to interface with the operating system and to track the progress of your macro application.

### 8.1.1 Accessing System Environmental Variables Using %SYSGET

Just as SAS uses macro variables, operating systems use a similar system of symbolic variables known as *environmental variables*. SAS takes advantage of these environmental variables in a number of ways, and this is usually to store information that has some connection between SAS and the operating system.

Sometimes we would like to access the information stored in these environmental variables, and we can do just that by using the %SYSGET macro function, which is similar to the SYSGET DATA step function.

Environment variables can be set either through the operating system or by SAS, and since these variables can provide a link between SAS and the operating system, they can be a valuable interface tool when writing macros.

**SYNTAX:**

```
%SYSGET(environmentalvariablename)
```

**VALUE RETURNED:**

Value held by the environmental variable

Probably the most difficult part about using this function is knowing what environmental variables exist, and how the information that those environmental variables hold will be helpful. A number of environment variables are available to the user; however, they vary by operating system, and can be additionally tailored when SAS is invoked. Your SAS Companion and SAS Online Doc go into some detail on setting environment variables, either through SAS or through the operating system.

#### Environmental Variables Created by the Configuration File

When the configuration file is executed at SAS initialization, a number of environmental variables are created. In the configuration file, under Windows, the keyword SET is used to name the environmental variables. Here is a portion of a SASv9.cfg file (SAS9.4 under Windows) that creates the SASAUTOS environmental variable:

```
-SET SASROOT "C:\Program Files\SASHome2\SASFoundation\9.4" ①
-SET SASAUTOS (
    "!SASROOT\core\sasmacro" ②
    "!SASROOT\aacomp\sasmacro"
    "!SASROOT\accelmva\sasmacro"
    "!SASROOT\assist\sasmacro"
    . . . portions of this statement are not shown . . .
```

- ① The SASROOT environmental variable is defined.
- ② The SASROOT environmental variable is used in the definition of the SASAUTOS environmental variable. The SASAUTOS environmental variable can be used as a *fileref* in SAS programs. It can also be retrieved using the %SYSGET function. The value stored in SASAUTOS can be surfaced by using

the %SYSGET function, and a portion of the SAS Log showing the usage of %SYSGET with the SASAUTOS environmental variable is shown here:

```
275  %put %sysget(sasautos);
(
  "!SASROOT\core\sasmacro"
  "!SASROOT\aacomp\sasmacro"
  "!SASROOT\accelmva\sasmacro"      "!SASROOT\assist\sasmacro"
  . . . portions of the LOG are not shown . . .
```

You can see some of the environment variables that SAS has created and their current values by viewing the value of the SET system option in SASHELP.VOPTIONS.

#### **Program 8.1.1a: Viewing Selected Environmental Variables**

```
proc print data=sashelp.voption(where=(optname='SET'));
run;
```

#### **Finding the SAS Executable File Location**

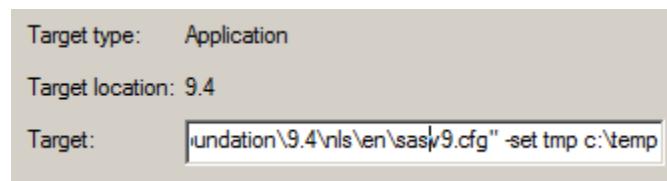
If you are writing code that will be used across operating systems or for different versions of SAS, you may need to know the location of the SAS executable file. For some operating systems, like Windows, this location information is stored in the !SASROOT environment variable, and the %SYSGET function can be used to determine this value directly. To create a macro variable that contains the full path to the executable file for the current OS and version of SAS, the %SYSGET is used to retrieve the current value of !SASROOT.

```
%let sasloc = %sysget(sasroot)\sas.exe;
```

#### **Accessing Environmental Librefs and Filerefs**

One common use of environmental variables is to associate locations (paths or directories) with a name. Usually, the LIBNAME or FILENAME statements are used to create this association from within SAS, but if the association is created outside of SAS, the programs can become more location independent and may require less maintenance when moved from machine to machine. You can ask SAS to interpret an environmental variable as a *libref* or *fileref*. The SASAUTOS environmental variable is used as a *fileref* in the SASAUTOS system option when setting up the autocall library (see Section 10.1.3). Under Windows you can set the environmental variable in the properties section of the SAS shortcut by using the –SET initialization option (see Section 14.6 for more on SAS initialization options).

**Figure 8.1.1: Using the –SET Initialization Option to Create an Environmental Variable**



Once it has been created, this environmental variable can be used as a *libref*, even though it will not show up on your list of libraries. TMP now refers to the directory C:\TEMP.

#### **Program 8.1.1b: Using an Environmental Variable as a libref**

```
proc print data=tmp.oldtest;
run;
```

The method used to set the environmental variables will vary among operating systems. Consult the SAS Companion for your OS.

## Returning the Name and Path of the Currently Executing Program

When executing an existing SAS program from the Enhanced Editor within the Display Manager (Windows), SAS knows where in the operating system the program is stored and what its name is. You can retrieve that information by using the SAS\_EXECFILEPATH and SAS\_EXECFILENAME environmental variables. Program 8.1.1c shows how the name of the executing program is returned by using %SYSGET in a FOOTNOTE statement.

### Program 8.1.1c: Returning the Executing Program Name

```
footnote1 justify=c
  "The executing program is: %sysget(sas_execfilename)";
proc print data=sashelp.class;
run;
```

The environmental variable SAS\_EXECFILEPATH contains both the name of the file and the physical path to that file. The %GRABPATH macro shown in Program 8.1.1d uses these two environmental variables together to return the path without the program name.

### Program 8.1.1d: Returning the Path of the Executing Program

```
%macro grabpath ;
  /* return the path of the currently executing program;
   %qsubstr(%sysget(SAS_EXECFILEPATH), ③
             1, ④
               %length(%sysget(SAS_EXECFILEPATH)) - ⑤
                   %length(%sysget(SAS_EXECFILENAME))
             )
  %mend grabpath;

footnote1 justify=c "The path to the executing program is: %grabpath";
proc print data=sashelp.class;
run;
```

- ③ The %QSUBSTR function is used to grab the path portion.
- ④ The grab starts in the first position and continues until the name of the program.
- ⑤ The length of the whole path (including the program name) less the length of the name of the program yields the width of the path portion of the text contained in the SAS\_EXECFILEPATH environmental variable.

You can use the operating system itself to surface the currently defined environmental variables. Under Windows environmental variables are defined using the SET command. When used without an argument the SET command lists all the currently defined environmental variables. The %SYSEXEC statement can be used to issue the SET command as shown in Program 8.1.1e.

### Program 8.1.1e: Listing Current Environmental Variables and Their Values

```
options nowait;
%sysexec set > c:\temp\environvar.txt; ⑥
```

- ⑥ The SET command is issued without an argument, and the results are written to the specified file.

A portion of the file (C:\temp\environvar.txt ⑥) shows some of the current environmental variable values.

```
SASCFG=C:\Program Files\SASHome2\SASFoundation\9.4\nls\en
SASHOME=C:\Program Files\SASHome2
SASROOT=C:\Program Files\SASHome2\SASFoundation\9.4
SAS_EXECFILENAME=Carpenter_17835TW_Program8.1.1e.sas
```

If you are using SAS Enterprise Guide or SAS Studio, the macro variable &\_SASPROGRAMFILE can be used. This macro variable returns the full path and filename of the SAS program that is currently being run. This macro variable is available only for SAS program files that are saved on the same server on which your SAS Studio code or SAS Enterprise Guide session is running.

**MORE INFORMATION:** The SYMEXIST DATA step function (see Section 8.5.3) can be used to determine if an environmental variable has been defined.

**SEE ALSO:** Levin (2001) and Lund (2001a, 2001b) use the %SYSGET macro function. Carpenter (2008) discusses the %GRABPATH macro in more detail, as well as other ways to access system environmental variables. Pahmer (2014) uses %SYMGET to retrieve the name of the executing program.

## 8.1.2 %SYSMEXECDEPTH and %SYSMEXECNAME

When you have developed a series of nested macros (macros that call other macros), it can sometimes become important to be able to determine which macros are being called and in which order. The nesting depth and the name of the executing macro at each depth can be surfaced using the %SYSMEXECDEPTH and %SYSMEXECNAME functions. These two functions are usually used together, however it is not necessary to do so.

### SYNTAX:

```
%SYSMEXECDEPTH
```

### VALUE RETURNED:

The number of nesting levels (0 for open code)

### SYNTAX:

```
%SYSMEXECNAME (level_number)
```

### VALUE RETURNED:

Name of the called macro at the specified nesting level

The macro %SHOWMACNEST in Program 8.1.2 uses the %SYSMEXECDEPTH and the %SYSMEXECNAME functions to highlight the nesting structure of nested macros.

### Program 8.1.2: Using the %SYSMEXECDEPTH and %SYSMEXECNAME Functions

```
%macro ShowMacNest;
  %local i;
  %do i = 1 %to %sysmexecdepth; ①
    %put Level &i, Macro name is: %sysmexecname(&i); ②
  %end;
%mend showmacnest;
```

- ① The %SYSMEXECDEPTH function returns the total number of nesting levels. Here this value is used as the upper bound for a %DO loop.
- ② The %SYSMEXECNAME function returns the macro name for the specified nesting level (in this case the level is &I).

The use of %SHOWMACNEST is demonstrated in the nested macros shown here. In this example, the macro %ONE calls %TWO, which calls %THREE, which calls %SHOWMACNEST.

```
%macro one;
  %put in one;
  %two
%mend one;
%macro two;
  %put in two;
```

```
%three
%mend two;
%macro three;
  %put in three;
  %showmacnest ③
%mend three;

%put Level 0: %sysmexecname(0); ④
%one
```

- ③ %SHOWMACNEST is called from within the macro %THREE.  
 ④ Nesting level = 0 is used to indicate open code.

The SAS Log shows the various nesting levels:

```
Level 0:OPEN CODE ④
1280 %one
in one
in two
in three
Level 1, Macro name is:ONE ②
Level 2, Macro name is:TWO ②
Level 3, Macro name is:THREE ②
Level 4①, Macro name is:SHOWMACNEST ②
```

**MORE INFORMATION:** The automatic macro variable &SYSMACRONAME, which surfaces the name of the currently executing macro is described in Section 8.3.5.

**SEE ALSO:** Langston (2013) describes a macro that checks for the existence of a specified macro.

### 8.1.3 Assessing Macro Existence and Execution Status with %SYSMACEXEC and %SYSMACEXIST

When you are executing an application that has a series of macros that call other macros, it is not always easy to determine which macro is currently executing or sometime even if a macro definition currently exists. Fortunately, we are not without tools to help us. In Section 8.1.2 the %SYSMEXCDEPTH and %SYSMEXCNAME functions are used to show nesting structure.

The %SYSMACEXEC and %SYSMACEXIST functions can be used to determine if a macro is currently executing or if it has been compiled.

#### SYNTAX:

```
%SYSMACEXIST (macro_name)
```

#### VALUES RETURNED:

- 1 if the macro has been compiled and resides in the WORK.SASMACR catalog
- 0 if the macro definition is not in WORK.SASMACR

#### SYNTAX:

```
%SYSMACEXEC (macro_name)
```

#### VALUES RETURNED:

Determines if the named macro is currently executing

The macro %MACEXEC in Program 8.1.3 checks to see if the specified macro has been compiled and whether it is currently executing.

**Program 8.1.3: Determine If a Macro Has Been Compiled and If It Is Executing**

```

options sasmstore=macro3 mstored; ①

%macro one/store; ②
  %put in one;
  %two
%mend one;
%macro two;
  %put in two;
  %three
%mend two;
%macro three;
  %put in three;
  %macexec(one) ③
  %macexec(three) ④
  %macexec(silly) ⑤
%mend three;
%macro Macexec(macname);
  %if %sysmacexist(&macname) %then
    %put %upcase(&macname) exists in WORK.SASMACR; ⑥
  %else %put %upcase(&macname) does not exist in WORK.SASMACR;
    %if %sysmacexec(&macname) %then
      %put %upcase(&macname) is currently executing; ⑦
    %mend macexec;
%one ⑧

```

- ① Turn on the ability to use stored compiled macros so the interaction with this type of library can be demonstrated.
- ② Store the compiled version of %ONE in the stored compiled macro library.
- ③ %ONE is executing but the compiled macro is not in the WORK catalog.
- ④ %THREE is executing and the compiled macro is in the WORK catalog.
- ⑤ %SILLY does not exist and has not been compiled.
- ⑥ Check to see if the macro has been compiled.
- ⑦ Check to see if the macro is currently executing.
- ⑧ The macro %ONE is called, which in turn will call the other macros.

```

1376 %one ⑧
in one
in two
in three
ONE does not exist in WORK.SASMACR
ONE is currently executing ① ⑦
THREE exists in WORK.SASMACR ④ ⑥
THREE is currently executing ④ ⑦

```

Only the macro %THREE is detected in the WORK.SASMACR catalog by the %SYSMACEEXIST function, while both the %ONE and %THREE macros are detected as executing by the %SYSMACEEXEC function.

**8.1.4 Determining Product Availability Using %SYSPROD**

The %SYSPROD macro function can be used to determine if a particular SAS product has been licensed at your site. The function argument is the name of the product that you want to check for. It will also let you know if you have used it to query for a product that the function does not recognize.

**SYNTAX:**

```
%SYSPROD (product_name)
```

**VALUE RETURNED:**

- 1 if the product is available
- 0 if the product is not available
- 1 if the product name is not recognized

In Program 8.1.4 the macro %CHECKPROD uses the %SYSPROD macro function to check the availability of a specified SAS product.

**Program 8.1.4: Using the %SYSPROD Function**

```
%macro Checkprod(prod=);
  %if %sysprod(&prod)=1 %then %put &prod is available;
  %else %if %sysprod(&prod)=0 %then %put &prod is not available;
  %else %if %sysprod(&prod)=-1 %then %put &prod is unknown;
%mend checkprod;
```

The SAS Log shows that SASGRAPH is not an acceptable code for a SAS product, while GRAPH is:

```
1448 %checkprod(prod=sasgraph)
sasgraph is unknown
1449 %checkprod(prod=graph)
graph is available
1450 %checkprod(prod=gis)
gis is not available
```

One of the disadvantages of this function is that it expects that the SAS products use specific codes, and it is not obvious what those codes are. Worse, the documentation only lists a few of the codes for some of the more common products. Some of the commonly used codes for the %SYSPROD function are as follows:

- AF
- ASSIST
- BASE
- CALC
- CONNECT
- CPE
- EIS
- ETS
- FSP
- GIS
- GRAPH
- IML
- INSIGHT
- LAB
- OR
- PH-CLINICAL
- QC
- SHARE
- STAT
- TOOLKIT

### 8.1.5 Checking Up on Macro Variable Scopes

There are three macro functions that can be used to determine if a macro variable exists and if so, what symbol table it resides in.

#### SYNTAX:

```
%SYMEXIST (macro_variable_name)
%SYMGLOBL (macro_variable_name)
%SYMLOCAL (macro_variable_name)
```

The %SYMEXIST function is used to determine whether a macro variable exists. The %SYMGLOBL and %SYMLOCAL functions are used to determine whether a macro variable resides in either the global or a local table, respectively. Each of these functions returns a true/false (1 or 0). If either %SYMGLOBL or %SYMLOCAL is true %SYMEXIST will necessarily be true as well. The macro %SYMCHKUP in Program 8.1.5 returns a 0 if the macro variable does not exist, 1 if it is global, 2 if it is local, and 3 if there is both a global and local instance of the macro variable.

#### Program 8.1.5: Checking the Scope of a Macro Variable

```
%macro symchkup (mvar);
  %local __rc;
  %let __rc = %eval( %synglobl(&mvar)      ①
                      + %symlocal(&mvar)*2); ②
  &__rc
%mend symchkup;

/* Test;
%put DNE has a rc of %symchkup(DNE); ③
%let silly=global; ④
%put SILLY has a rc of %symchkup(silly); ⑤
```

The SAS Log shows that the %SYMCHKUP macro detects the presence of macro variables of various scopes:

```
172 %mend symchkup;
173 %put DNE has a rc of %symchkup(DNE);
DNE has a rc of 0 ③
174 %let silly=global; ④
175 %put SILLY has a rc of %symchkup(silly);
SILLY has a rc of 1 ⑤
```

- ① %SYMGLOBL will return a 0 or a 1.
- ② %SYMLOCAL will return a 1 if the macro variable exists in any of the existing local tables. This value is multiplied by 2 and the result is added into &RC.
- ③ The macro variable &DNE does not exist and %SYMCHKUP returns a 0.
- ④ &SILLY is defined in the global symbol table, but does not exist in any local table
- ⑤ %SYMCHKUP returns a 1 indicating that the macro variable exists in the global symbol table.

Because %SYMLOCAL detects a macro variable in any local table, a macro variable that exists in multiple local tables will only be detected once.

**MORE INFORMATION:** Additional examples of the use of these functions can be found in Section 9.2.1. Similar functions can be found in the DATA step (see Section 8.5.3).

**SEE ALSO:** Mason (2016) uses %SYMEXIST to check for the existence of macro variables.

## 8.2 Even More Macro Statements

There are a number of macro language statements that have not been introduced in other sections of this book. Most of these are less commonly used, either because they are not needed as often or as you will see, because of author bias. A few others were only briefly introduced elsewhere and are described in more detail in this section.

### 8.2.1 Extending the Use of %SYMDEL

The %SYMDEL statement, which was introduced in Section 2.7, is intended to be used to delete macro variables from the GLOBAL symbol table. The statement accepts a list of macro variables that are referenced directly (without the ampersand).

#### SYNTAX:

```
%SYMDEL list_of_variables </option>;
```

The %SYMDEL statement in Program 8.2.1a removes the macro variables &NADA and &DSN from the GLOBAL symbol table

#### Program 8.2.1a: Deleting Two Macro Variables Using %SYMDEL

```
%symdel nada dsn;
```

By default a warning is issued if an attempt is made to delete a macro variable that does not exist, however the NOWARN option can be used to suppress this warning.

```
%symdel nada dsn/nowarn;
```

As is shown in Program 8.2.1b, you can use indirect references to specify the macro variable or variables that are to be deleted. Program 8.2.1b demonstrates a usage of an indirect list.

#### Program 8.2.1b: Using a Macro Variable to Reference a List

```
%let nada=;
%let dsn=clinics;
%let macvarlist = nada dsn xyz;
%symdel &macvarlist / nowarn;
```

%SYMDEL does not offer a lot of flexibility if you want to delete all the macro variables in the global symbol table. However, by first creating a list of all the macro variables, and then using that list as in Program 8.2.1b, you can indeed do so. The code in Program 8.2.1c enables you to dynamically delete all the macro variables in the global symbol table using %SYMDEL.

#### Program 8.2.1c: Deleting All Macro Variables from the Global Symbol Table

```
proc sql noprint;
  select distinct name
    into :maclist separated by ' '
    from dictionary.macros
    where upcase(SCOPE) eq 'GLOBAL'
      and name ne 'maclist'
/*
      and name ne 'SYS_SQL_IP_ALL'*/
/*
      and name ne 'SYS_SQL_IP_STMT'*/
;
quit;
%put &=maclist;
%symdel &maclist maclist;
%put _global_;
```

The SQL step places a couple of read-only automatic macro variables in the global symbol table. Since they are read-only they cannot be deleted and the attempt will cause an error.

```
ERROR: Attempt to delete automatic macro variable SYS_SQL_IP_ALL.
ERROR: Attempt to delete automatic macro variable SYS_SQL_IP_STMT.
```

You could prevent this error by excluding these macro variables in the WHERE clause in the SQL step (logic commented out in Program 8.2.1c).

Although it seems less of a problem in the current versions of SAS, the use of a macro variable in the %SYMDEL statement may cause an error due to a timing conflict between the compilation and execution of the statement. If the timing problem is encountered, it can be solved in a couple of different ways. The first is to use quoting functions to control what is resolved first. If you do encounter a problem when using a list such as was done in Programs 8.2.1b and 8.2.1c, you can delay the execution by quoting the %SYMDEL statement keyword.

#### **Program 8.2.1d: Using Quoting to Delay Execution**

```
%let nada=;
%let dsn=clinics;
%let maclist = nada dsn;

%unquote(%nrstr(%symdel) &maclist / nowarn);
```

The %NRSTR prevents resolution of the %SYMDEL until after &MACLIST has been resolved. Once &MACLIST has been resolved, the %UNQUOTE removes the quotes and %SYMDEL will be applied to the resolved list of macro variables.

Another solution is to delete the macro variables one at a time by using the CALL EXECUTE routine from within a DATA step. Several variations of this solution have been presented, including ones by SAS Technical Support. The macro %DELVARS shown in Program 8.2.1e, which uses SASHELP.VMACRO and the CALL EXECUTE routine to delete all the macro variables with SCOPE='GLOBAL', is very similar to a macro of the same name, which can be found in SAS Sample 26154.

#### **Program 8.2.1e: Using %SYMDEL with CALL EXECUTE**

```
%macro delvars;
  data vars;
    set sashelp.vmacro;
    where scope='GLOBAL' & substr(name,1,3) ne 'SYS';
    if name ne lag(name) then output vars;
  run;
  data _null_;
    set vars;
    call execute('%symdel '||trim(left(name))||'/nowarn;');
  run;
%mend delvars;

%let nada=;
%let dsn=clinics;
%delvars
%put _global_;
```

Notice that since SASHELP.VMACRO is a VIEW that points back to the symbol table(s), it cannot be used in the same DATA step as the CALL EXECUTE. Again, this is a timing issue—a CALL EXECUTE timing issue this time.

If you try to delete macro variables that are not on the global table (perhaps because the variables do not exist or they exist only on a local table), you will get a warning indicating that the macro variable was not found. This warning is suppressed by using the /NOWARN option.

**MORE INFORMATION:** The %SYMDEL statement was introduced in Section 2.7.

**SEE ALSO:** Watts (2003a) has an example of the PROC SQL step that prepares a list of GLOBAL macro variables for deletion. Similar examples and discussions have appeared on SAS-L by several authors. Discussion of the use of %SYMDEL and variations of the macro %DELVARS can also be found on the SAS Technical Support page under the FAQ section relating to macros.

diTommaso (2003) also discusses the use of %SYMDEL with a CALL EXECUTE.

An alternative to deleting macro variables that has more flexibility can be found on sasCommunity.org: [http://www.sascommunity.org/wiki/Deleting\\_global\\_macro\\_variables](http://www.sascommunity.org/wiki/Deleting_global_macro_variables).

Langston (2015b) demonstrates the use of %SYMDEL.

## 8.2.2 Using the %GOTO and %*label* Statements Appropriately

The %GOTO and %*label* statements are included in this book because you might encounter them someday in someone else's code (warning: subtle author bias may be encountered in this subsection). These statements, like other directed branching statements, enable you to create code that is **very** unstructured. So far (when I have tried hard enough), I have always been able to find better ways of solving a problem (both in coding SAS and in my personal life) other than by using GOTO and %GOTO type statements. My first choice is to use alternative logic, thereby avoiding the use of these statements.

Like the DATA step GOTO statement, %GOTO (or %GO TO) causes a logic branch in the processing. The branch destination will be a macro label (%*label*). Therefore, the argument associated with the %GOTO must resolve to a known %*label*.

### SYNTAX:

```
%GOTO label;  
or  
%GO TO label;  
%LABEL:
```

The *label* associated with the %GOTO statement must resolve to a macro label that you have defined somewhere within the macro using the %*label* statement. The label may be explicitly or implicitly named. In the following example, the label is named explicitly. After execution of the %GOTO statement, the next statement to be executed will be the statement following the %NEXTSTEP: label:

```
%GOTO NEXTSTEP;  
...code not shown...  
  
%nextstep:  
...code not shown...
```

In code that uses %GOTO, it is not unusual for the %GOTO statement to include a label that contains a reference to a macro variable that must be resolved before the %GOTO is executed. In the following example &STEP must resolve to a defined macro label—for example, NEXTSTEP—before the branch can take place. This is often referred to as a *directed* or *computed* %GOTO.

```
%let step = nextstep;  
  
%GOTO &STEP;
```

Because the macro label is preceded by a %, the new user often uses a % with the *label* in the %GOTO statement, as in this statement:

```
%GOTO %NEXTSTEP;
```

Rather than branching to the specified %label, however, a call to execute the macro %NEXTSTEP will be issued before the %GOTO can be executed. Generally, this will result in an error, but it could work if the macro %NEXTSTEP resolves to the name of a macro label.

In the following example, %GOTO is used to determine which of two DATA steps will be executed. The macro labels are explicitly defined in the %GOTO statements. Notice that the %label statement is followed by a colon and *not* a semicolon.

#### Program 8.2.2a: Using %GOTO with Explicit Labels

```
%macro mkwt(dsn);
/* Point directly to the label;
%if &dsn = MALE %then %goto male;
  data wt;
  set female;
  wt = wt*2.2;
  run;
%goto next;
%male:
  data wt;
  set male;
  run;
%next:
%mend mkwt;
```

You can rewrite this example to use implicit labels that reflect the incoming macro variable (&DSN). This makes the use of the %IF unnecessary.

#### Program 8.2.2b: Using %GOTO with Implicit Labels

```
%macro make(dsn);
/* Point indirectly to the label;
%goto &dsn; /* DSN takes on either MALE or FEMALE;
%female:
  data wt;
  set female;
  wt = wt*2.2;
  run;
%goto next;
%male:
  data wt;
  set male;
  run;
%next:
%mend make;
```

Admittedly, this is a rather simplistic case, but you can generally rewrite programs that use %GOTO to avoid the use of the %GOTO altogether.

**Program 8.2.2c: Avoiding the Use of the %GOTO**

```
%macro smart(dsn);
  /*AVOID GOTO WHEN POSSIBLE;
  data wt;
  set &dsn;
  %if &dsn=FEMALE %then wt = wt*2.2;;
  run;
%mend smart;
```

A common use of %GOTO is to avoid execution of portions of a macro by skipping to the macro's %MEND statement. To illustrate the point, Program 8.2.2d is a rather silly example of this technique.

**Program 8.2.2d: Using %GOTO to Skip to the End of a Macro**

```
%macro modfem(dsn);
  /* Execute only for Females;
  %if &dsn ne FEMALE %then %GOTO skip;
  data &dsn;
  set &dsn;
  wt = wt*2.2;;
  run;
  %skip:
%mend modfem;
%modfem(MALE)
```

We could rewrite the %MODFEM macro to avoid the DATA step by using a %DO block just as easily as by skipping to the end of the macro.

When conditions warrant macro termination, rather than skipping to the end of the macro with a %GOTO, the %RETURN statement (see Section 5.4.5) can be used.

**Program 8.2.2e: Using %RETURN to Terminate the Execution of a Macro**

```
%macro modfem(dsn);
  /* Execute only for Females;
  %if &dsn ne FEMALE %then %return;
  data &dsn;
  set &dsn;
  wt = wt*2.2;;
  run;
%mend modfem;
```

**MORE INFORMATION:** The %GOTO statement is used in %TRIM, an autocall macro supplied by SAS, which is discussed in Section 10.6.5.

**SEE ALSO:** The %GOTO statement and %label are used by Wang (2003) in a %WINDOW example. Lund (2003a) uses %GOTO to skip the execution of a macro.

---

### 8.2.3 Using %WINDOW and %DISPLAY

Through the use of the %WINDOW statement, the macro language provides the programmer with a tool that can be used to establish a basic user interface. Similar to the WINDOW statement in the DATA step, %WINDOW can be used to create and display message boxes and to collect information from the user that can then be placed into macro variables.

The %WINDOW statement can be used to do the following:

- display a window
- control window attributes including size and color
- make use of existing key and menu definitions
- display existing macro variable values

- define and assign values to macro variables

Once a window has been defined with the %WINDOW statement, it can then be displayed by using the %DISPLAY statement. The %WINDOW and %DISPLAY statements can be used in open code.

#### SYNTAX:

```
%WINDOW window-name <attributes and display characteristics>;
%DISPLAY window-name <display control options>;
```

Because %WINDOW can be used to create macro variables, it can be useful when having the user specify execution time specific parameters without editing the program. The macro %DSNPROMPT in Program 8.2.3a defines and then displays a macro window, which prompts the user for the name of a data set within the declared library.

#### Program 8.2.3a: Using %WINDOW to Prompt for a Data Set Name

```
%macro dsnprompt(lib=sasuser);
/* prompt user to for data set name;
>window verdsn color=white ①
#2 @5 "Specify the data set of interest" ②
#3 @5 "for the library &lib" ③
#4 @5 'Enter Name: '
      dsn 20 ④ attr=underline required=yes ⑤;

&display verdsn; ⑥

title1 "8.2.3a Print the &lib..&dsn data set";
proc print data=&lib..&dsn;
run;
%mend dsnprompt;

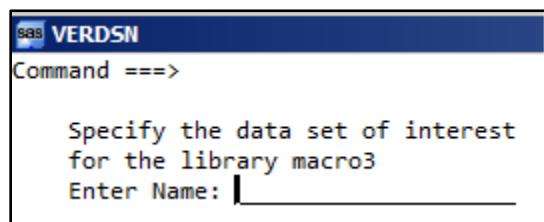
%dsnprompt(lib=macro3)
```

When the %DSNPROMPT macro is executed, the VERDSN macro window will be defined and displayed.

- ① The VERDSN window will have a white background with no specifications for size.
- ② The text (in single or double quotes) is to be displayed at row 2 and column 5 of the window. The same notation for row (#) and column (@) is used as in the PUT and INPUT statements.
- ③ Macro variables can be included in the text (see caveat below).
- ④ The user is prompted for the name of a data set which is placed into &DSN.
- ⑤ Attributes can be assigned to the display of the macro variable.
- ⑥ Although defined by the %WINDOW statement, the VERDSN window is not displayed until the %DISPLAY statement is executed.

The VERDSN window defined and displayed in the %DSNPROMPT macro is shown in Figure 8.2.3a.

**Figure 8.2.3a: Prompting for a Data Set Name**



**CAVEAT:** In Program 8.2.3a the VERDSN window is defined when the macro is executed. It is during this definition phase that &LIB at ❸ is resolved. If &LIB is redefined at a later time (after the %WINDOW statement has executed), then %DISPLAY will still show the original value, not the current value of &LIB. Appendix 2 has additional examples of instances where macro variables are resolved during the compilation phase.

There are a number of attributes that can be associated with a macro window. The window definition shown in Program 8.2.3b specifies the background and foreground colors, the location, and the size of the window.

#### Program 8.2.3b: Prompting for a Two-Digit Year

```
%window getyear
  color=magenta ❷
  icolumn=15 ❸
  irow=10
  columns=30 ❹
  rows=15
  #3 @3 'Enter the two digit year' color=white ❺
  #4 @5 yr 2 color=white attr=underline required=yes ❻
  #5 @3 'Then press "Enter" ' color=white
  ;
%display getyear;
```

- ❷ Color attributes are specified using the COLOR= option for both the background and foreground colors.
- ❸ The upper left corner is located using the ICOLUMN= and IROW= options.
- ❹ The window size is control by using COLUMNS= to specify width and the ROWS= option to specify height.
- ❺ The incoming macro variable (&YR) value is allowed two characters and is required.

Figure 8.2.3b: Prompting for a Two-Digit Year



**SEE ALSO:** Many of the options associated with %WINDOW are introduced and discussed by Alden (2000) and Mace (1997, 1998, 2000, 2002, and 2003). These papers provide very nice overviews as well as detailed (and in most cases more sophisticated) examples of macro windows.

Gau (1999) presents an example of the %WINDOW to manage programs. Ren (1999), Parker (2000), Dynder, Cohen, and Cunningham (2000), Fahmy (2003), Huang (2003), Wang (2003), Parker (2003), and Rhoads and Letourneau (2002) each use a macro window to create user interfaces. Plath (2002) creates and executes a series of macro windows to collect information from the user.

Access control to macros is achieved through the use of the WINDOW statement in an example by Shilling and Kelly (2001).

Glass and Hadden (2016) use the %WINDOW and %DISPLAY statements to collect information from the user of a SAS program.

### 8.2.4 Extending %SYSEXEC with Examples

The %SYSEXEC statement enables you to execute operating system commands and statements from within the macro language. The macro %MAKEDIR in Program 8.2.4 can be used to verify that a directory exists, and, if it does not already exist, to create it. The only parameter used by %MAKEDIR is the directory path to be checked.

#### Program 8.2.4: Verify that a Directory Exists Using %SYSEXEC

```
%macro makedir(newdir);
  %local rc;
  /* Make sure that the directory exists;
  %let rc = %sysfunc(fileexist(&newdir)); ①
  %if &rc=0 %then %do;
    %put Creating directory &newdir;
    /* Make the directory;
    %sysexec md &newdir; ②
  %end;
  %else %put Directory &newdir already exists;
%mend makedir;

options nowait; ③
%makedir(c:\tempzzz)
```

- ① The FILEEXIST function is used to see if the “file,” which in this case is actually a directory, exists. The return code from this function is 0 if the file is not found.
- ② A return code of 0 indicates that the directory does not exist and should be created. %SYSEXEC is used to execute the Windows MD (make directory) command.
- ③ Under Windows the X statement and the %SYSEXEC statement both open a DOS window. Without specifying the NOXWAIT system option you will need to close that window manually.

The advantage of the %SYSEXEC macro statement over the X statement is that you do not need to leave the macro environment before executing the operating system command. By using the %SYSEXEC statement the %MAKEDIR macro mimics a macro function. If an X statement had been used the macro would have had a more limited utility.

When the command issued by %SYSEXEC is executed by the operating system, the success or failure of that OS command is returned to SAS as a code, which is stored in the automatic macro variable &SYSRC. A value of 0 indicates success. In the code that follows, the %MAKEDIR macro attempts to create a directory on the Z: drive, which for this example does not exist.

```
%makedir(z:\tempzzz)
%put Return Code: &sysrc ;
```

The SAS Log shows that &SYSRC will contain a 1 indicating that the directory was not created:

```
62  %makedir(z:\tempzzz)
Creating directory z:\tempzzz
63  %put Return Code: &sysrc ;
Return Code: 1
```

**MORE INFORMATION:** The %SYSEXEC statement is introduced in Section 5.4.3 and is used in Program 8.1.1e.

**SEE ALSO:** Dynder, Cohen, and Cunningham (2000) use a %WINDOW interface to generate a series of directories. The FILEEXIST function is also used by Lund (2003a) to check and establish directories.

Jia(2015) uses %SYSEXEC to create a directory.

## 8.2.5 Deleting Macro Definitions with %SYSMACDELETE

Unless stored compiled macro libraries are being used, the compiled macro is stored in the WORK.SASMACR catalog. As a general rule, it does not matter how many macros are in this catalog or even whether a given macro already exists if it is to be recompiled, however there are instances when it does make a difference (Sun and Carpenter, 2011). Although you can *generally* delete entries from the WORK.SASMACR catalog manually using the Display Manager or other SAS interfaces, this is neither a recommended nor a supported technique.

The %SYSMACDELETE statement is the supported tool for deleting compiled macros from the WORK.SASMACR catalog.

### SYNTAX:

```
%SYSMACDELETE macro_name </NOWARN>;
```

You can only delete one macro for each instance of the %SYSMACDELETE statement, and the NOWARN option can be used to suppress warnings if you try to delete a macro that is either currently executing or does not exist in the WORK.SASMACR catalog.

### Program 8.2.5: Using the %SYSMACDELETE Statement to Delete a Macro Definition

```
%macro silly0;
%* silly0;
%mend silly0;
%macro silly1;
%* silly1;
%mend silly1;
%macro silly2;
%* silly2;
%mend silly2;
%sysmacdelete silly0; ①
%sysmacdelete silly1 silly2; ②
%sysmacdelete silly3; ③
%sysmacdelete silly3 /nowarn; ④
```

After execution of Program 8.2.5 the SAS Log shows the following:

```
113 %sysmacdelete silly0; ①
114 %sysmacdelete silly1 silly2; ②
WARNING: Extraneous argument text on %SYSMDELETE call ignored: SILLY2
115 %sysmacdelete silly3; ③
WARNING: Attempt to delete macro definition for SILLY3 failed. Macro
definition not found.
116 %sysmacdelete silly3 /nowarn; ④
```

- ① The definition for %SILLY0 is deleted.
- ② A warning is issued, because of the second name (%SILLY2). Even with the warning, the definition of the first macro named (%SILLY1) is deleted.
- ③ %SILLY3 does not exist and a warning is issued in the SAS Log.

- ④ %SILLY3 does not exist, but a warning is not issued because of the /NOWARN option.

**SEE ALSO:** Langston (2015b) demonstrates the use of %SYSMACDELETE.

## 8.2.6 Making Macro Variables **READONLY**

In Section 5.4.2 the %GLOBAL and %LOCAL statements are introduced along with the concept of macro variable collisions. These collisions occur when a macro variable assignment inadvertently overwrites the value of another macro variable with the same name in a different symbol table. The **READONLY** options on the %GLOBAL and %LOCAL statements are designed to mitigate some of the issues associated with macro variable collisions. They are not a panacea, however they can be very helpful when you need to protect one or more of your macro variables.

### SYNTAX:

```
%GLOBAL/READONLY varname=value;
%LOCAL/READONLY varname=value;
```

In Program 8.2.6 the macro variable &MYPATH is declared to be global and to be **READONLY**.

#### Program 8.2.6: Declaring a Global Macro Variable **READONLY**

```
%global/readonly mypath = &path; ❶
%put _user_; ❷
%let mypath = abc; ❸
%global/readonly mypath = abc; ❸
%symdel mypath;❸
```

- ❶ The macro variable &MYPATH is declared to be a read-only global macro variable and assigned the value stored in &PATH
- ❷ %PUT is used to show the user-defined macro variables. Notice that the SAS Log does not indicate that &MYPATH has been declared to be **READONLY**.
- ❸ Once declared to be **READONLY** the macro variable &MYPATH cannot be assigned a new value, nor can it be deleted from the global symbol table.

#### Program 8.2.6 (SAS Log): Showing the Usage of the **/READONLY** Option

```
127 %global/readonly mypath = &path; ❶
128 %put _user_; ❷
GLOBAL MYPATH C:\Primary
GLOBAL PATH C:\Primary
129 %let mypath = abc; ❸
ERROR: The variable MYPATH was declared READONLY and cannot be modified or
re-declared.
130 %global/readonly mypath = abc; ❸
ERROR: The variable MYPATH was previously declared as READONLY and cannot
be re-declared.
131 %symdel mypath;❸
ERROR: The variable MYPATH was declared READONLY and cannot be deleted.
```

When using the **READONLY** option on the %GLOBAL or %LOCAL statements, you cannot assign values to more than one macro variable at a time. In the following code an attempt is made to assign values to the macro variables &A, &B, and &C.

```
%global/readonly a=a b=b c=c;
```

In actuality only one macro variable is assigned &A, and %PUT \_USER\_ shows that &A contains the value of a b=b c=c.

**CAVEAT:** In Program 8.2.6 &MYPATH is declared to be a READONLY macro variable in the global symbol table. This declaration also precludes the use of this macro variable name in any local symbol table as well. In fact, the declaration of a READONLY macro variable prevents the use of that name in any other symbol table. READONLY macro variables persist until the end of the SAS session in which they are created.

## 8.3 Even More Automatic Macro Variables

A number of automatic macro variables were introduced in Section 2.6 as well as elsewhere within this book. Depending on how you use SAS and how you use the macro language, these macro variables will have varying utility to you. However, you need to have an understanding of what is available to you so that you can take full advantage of the ones that are actually valuable to you.

You can view the list of currently defined automatic macro variables along with their values by using the %PUT statement and the \_AUTOMATIC\_ option.

```
%put _automatic_;
```

This section describes some of the less commonly used, but no less valuable, automatic macro variables.

### 8.3.1 Passing VALUES into SAS Using &SYSPARM

The value of the SYSPARM system option can be loaded during the SAS initialization phase. Because this option can be used as a SAS initialization option, the value itself can be supplied before SAS is executed. This gives us the ability to pass values into SAS from an outside process or program. The value stored in this system option can be retrieved in a number of ways including the SYSPARM( ) DATA step function and the automatic macro variable &SYSPARM.

Because this option is most useful when its value is loaded when SAS is initially executed, it is most commonly used when SAS is executed in a batch execution environment. The SYSPARM initialization option specifies a character string that can be passed into SAS programs. The maximum length of this macro variable is 32K characters.

In the following example, you would like your programs to automatically direct your data to either a test or production library. To make this switch, assign &SYSPARM the value TST or PROD when you start the SAS session.

Assume that an Open VMS SAS session is initiated with:

```
$ sas/syssparm=tst
```

A typical LIBNAME statement on Open VMS which uses this value might be:

```
libname projdat "usernode:[study03.gdx&syssparm]";
```

The resolved LIBNAME statement becomes:

```
libname projdat "usernode:[study03.gdxtst]";
```

The syntax that you use to load a value into &SYSPARM depends on the operating environment that you are using. See the SAS Companion for your operating environment for more information. When using a shortcut under Windows, -syssparm tst appears on the TARGET LINE in the Properties Window of the shortcut. In JCL, the option is used on the SYSIN line.

The DATA step function SYSPARM() can also be used to retrieve the value of the SYSPARM system option. Depending on how this function is used it might not return the same value as &SYSPARM. The differences between using &SYSPARM directly and the SYSPARM () function are demonstrated in the

following example. Notice in this example that the value of the SYSPARM option contains a macro variable reference.

Initialize SAS using the -SYSPARM option.

```
"C:\... path not shown ...\\9.4\sas.exe" -sysparm &aaa; ①
```

Once initialized, &SYSPARM can be used throughout the session.

#### Program 8.3.1a: Returning &SYSPARM Values

```
%let aaa = AAAAA;
data try2;
a = "&sysparm"; ②
b = sysparm(); ③
put a=;
put b=;
run;
```

The SAS Log shows how the values are assigned to the variables A and B:

```
a=AAAAA; ②
b=&aaa; ③
```

- ① Usually, as in this example, the value for &SYSPARM is set when SAS is first invoked. Since at this point the code has not even been sent to the word scanner, the macro processor is not called, and therefore, no attempt is made to resolve &AAA. As a result, &SYSPARM contains the characters &aaa. If &SYSPARM contains a blank, and therefore more than one word, **double** quotes should be used.
- ② In the data set TRY2 the variable A is a character variable, which has a length of 5, and contains the value “AAAAA”. Before a value can be assigned to the variable A, &SYSPARM is first resolved to &aaa. This is in turn resolved to AAAAA, and it is this value that is then stored in the data set variable A.
- ③ While the variable B is also a character variable, it will, by default, have a length of 200 (this is the default length returned when using the SYSPARM function). Since the SYSPARM() function is executed during the DATA step execution phase, the value “&aaa” will be written directly to the variable B and no attempt will be made to resolve the macro reference.

If &SYSPARM contains more than 200 characters be sure to use the LENGTH statement to set the length of the variable created by the SYSPARM function, otherwise longer values will be truncated.

There is an interesting relationship between the –SYSPARM initialization option, the automatic macro variable &SYSPARM, and the SYSPARM system option. It turns out that updating any one of the three, changes the values of the others. This is good because this means that regardless of which method you use to retrieve the stored value, you will always get the same value. Program 8.3.1b demonstrates this relationship by showing that changing either changes both.

#### Program 8.3.1b (SAS Log): Showing the Relationship between &SYSPARM and the SYSPARM System Option

```
4   %let sysparm=something; ④
5   options sysparm=' ' ; ⑤
6   %put &sysparm;
SYSPARM= ⑥
7   %let sysparm=def; ⑦
8   data a;
9     x = sysparm(); ⑧
10    y = getopt('sysparm'); ⑨
11    z = "&sysparm"; ⑩
12    put x= y= z=;
```

```
13      run;

x=def y=def z=def ❸ ❹ ❺
```

- ❻ We make sure that &SYSPARM has a value using a %LET statement. This value could also have been set using the SAS initialization option (-SYSPARM).
- ❼ Changing the value of the SYSPARM system option also changes the value of &SYSPARM as is shown using a %PUT at ❾.
- ❽ The value of &SYSPARM is written to the SAS Log. The value of &SYSPARM was changed when the system option was updated ❾.
- ❿ The value of &SYSPARM is changed again.
- ❻ The SYSPARM function returns the value stored in the SYSPARM system option. The default length of X is \$200.
- ❻ The GETOPTION function returns the value of the SYSPARM system option. The default length of Y is \$200.
- ❻ The &SYSPARM macro variable is resolved before the assignment is made. The length of Z will be \$3.

**CAVEAT:** Not all operating systems are the same. Consult your SAS Companion for details or limitations on the number of characters that can be passed into &SYSPARM.

**SEE ALSO:** Johnson (2001) has an example that parses several words out of a single &SYSPARM value. Wong (2002) shows examples of both the SYSPARM macro variable and the SYSPARM() function.

### 8.3.2 Learning More about Deciphering Errors

When you encounter problems with the execution of various components of your macro, there are a number of automatic macro variables that you can inspect to try to get a handle on the coding problem.

**SEE ALSO:** Hughes (2016a) uses &SYSERR and &SYSERRORTEXT to examine errors associated with a failed SORT step. Billings (2015) describes a strategy for detecting errors, including the use of &SYSERR and &SQLRC.

#### &SYSERR, &SYSERRORTEXT, and &SYSWARNINGTEXT

The automatic macro variable &SYSERR, introduced in Section 2.6.3, is likely to be one of the first automatic macro variables that you might want to check. Because the codes stored in &SYSERR are cryptic, the automatic macro variables &SYSERRORTEXT and &SYSWARNINGTEXT contain the latest error and warning messages written to the SAS Log.

In a variation of the PROC DATASETS example shown in Section 2.6.3, the macro %COPYALL shown in Program 8.3.2a will check &SYSERR to see if the copy was successful. The resulting error codes are written to the SAS Log.

#### Program 8.3.2a: Checking for PROC Step Errors

```
%macro copyall(inlib=, outlib=);
proc datasets memtype=data;
  copy in=&inlib out=&outlib;
  quit;
%if &syserr>5 %then %do;
  %put ERROR: &=syserrortext; ❶
  %put ERROR: &=syserr; ❷
  %abort;
%end;
%put Copy was successful;
```

```
%mend copyall;
%copyall(inlib=combine, outlib=combtemp)
```

Because the COMBINE *libref* does not exist, the PROC step must fail. The SAS Log shows that &SYSERR takes on a value greater than 5, and &SYSERRORTEXT displays an explanation of this code:

```
158 %copyall(inlib=combine, outlib=combtemp)

ERROR: Libref COMBINE is not assigned.
NOTE: Statements not processed because of errors noted above.
NOTE: The SAS System stopped processing this step because of errors.
NOTE: PROCEDURE DATASETS used (Total process time):
      real time          0.02 seconds
      cpu time          0.03 seconds

ERROR: SYSERRORTEXT=Libref COMBINE is not assigned. ❶
ERROR: SYSERR=1008 ❷
ERROR: Execution terminated by an %ABORT statement.
```

- ❶ The text associated with the error is written to the SAS Log.
- ❷ The return code, which is stored in &SYSERR is written to the SAS Log.

When the %ABORT statement executes the %COPYALL macro terminates, and in this case the %PUT indicating that the copy was successful will not be executed.

Although the DATASETS procedure returns multiple codes (0=success, 1-4 are warnings, and greater than 4 are errors), most steps return a 0/1. This means that you will generally have a %IF statement that checks for &SYSERR values > 0:

```
%if &syserr>0 %then %do;
```

Clearly, using &SYSWARNINGTEXT and &SYSERRORTEXT to parrot back messages to the SAS Log is not particularly helpful. However, if you parse the contents of &SYSERRORTEXT for specific text you can have your macro take specific action. The %IF statement shown here (which is taken from Program 8.3.2b which is not shown), detects that a *libref* has not been established and calls a macro that creates it.

#### Program 8.3.2b (Partial): Checking Error Text to Make Decisions

```
%if %bquote(&syserrortext) =%bquote(Libref %upcase(&inlib) is not
assigned.) %then %makelib(&inlib);
```

**CAVEAT:** Be very careful when making decisions based on the text values that are contained in &SYSERRORTEXT and &SYSWARNINGTEXT. These are READONLY macro variables and they are not reset between step boundaries. Their values only change when a new error or warning is encountered. This means that the value of one of these macro variables could easily have been set in some prior step or even from an earlier program if you are running interactively. This makes the text checking such as was done in Program 8.3.2b somewhat impractical – unless, of course, you are very careful.

**MORE INFORMATION:** The %ABORT statement is introduced in Section 5.4.4. This statement includes options that determine the overall impact of this statement.

**SEE ALSO:** Shtern (2014) uses the CANCEL option on the %ABORT statement to terminate the SAS session.

Failure to copy can occur when a data set is locked. Hughes (2014a) carefully describes various locking situations as well as a macro to detect and avoid failures due to locks. Other descriptions of data set locks can be found in Graham and Osowski (2013) and Galligan (2011).

## &SYSCC

Step condition codes can also be examined using &SYSCC. Unlike &SYSERR, which is a READONLY variable, the value of &SYSCC can be reset by the user. In Program 8.3.2c the %RUNCHECK macro is used as a special batch process RUN statement, which automatically terminates the SAS process if the condition code exceeds the specified value for that step. In this program the %ABORT statement (see Section 5.4.4) includes the use of the ABEND option, which, when running interactively outside of a SAS/AF session, causes the SAS session to end.

### Program 8.3.2c: Checking Condition Codes Using &SYSCC

```
%macro RunCheck(codeval);
run;      /* terminate the step */ ③
%if &syscc > &codeval %then %do; ④
  %put ERROR: Condition Code &syscc exceeds &codeval;
  %put Aborting Process;
  %abort abend;
%end;
%else %if &syscc>0 %then %do; ⑤
  %put WARNING: Condition Code &syscc within limits;
  %let syscc=0;
%end;
%else %do;
  %let syscc=0;
%end;
%mend runcheck;

proc print data=sashelp.class;
  var name ht wt; ⑥
  %runcheck(500)
proc print data=sashelp.class;
  var name height weight;
  %runcheck(0)
```

- ③ Terminate the previous step with a RUN; statement.
- ④ If the value of &SYSCC exceeds the specified tolerance write a message to the SAS Log and terminate the process using a %ABORT statement.
- ⑤ Although &SYSCC exceeds 0, if it is not above the tolerance level for the step, therefore a warning is written to the SAS Log.
- ⑥ The variables HT and WT are not on the data set SASHHELP.CLASS. &SYSCC takes on the value of 3000, which exceeds the tolerance and the SAS session is aborted.

## Function Return Codes and the SYSMSG Function

The success or failure of function calls can also be evaluated within the macro language. In Program 8.3.2d the LIBNAME function is used to assign the libref TEMXX to a location (C:\TEMPXX) which does not exist. The SYSMSG() function returns the text associated with the most recent function call.

### Program 8.3.2d (SAS Log): Showing a Function Return Code and Its Message

```
180 %let rc = %sysfunc(libname(temxx,c:\tempxx)); ⑦
181 %put &rc %sysfunc(sysmsg()); ⑧
-70008 NOTE: Library TEMXX does not exist. ⑨
```

- ⑦ Since the LIBNAME function does not normally return a value, we can instead capture its return code in &RC.
- ⑧ Write the function's return code and its associated message to the SAS Log.
- ⑨ The SYSMSG() function will return the text associated with the call to the LIBNAME function.

**MORE INFORMATION:** The code in Program 8.3.2d is used in Programs 11.2.6a and 12.3.3 where the SYMSG function writes error messages when the LIBNAME function fails. There are two automatic macro variables that will capture the success or failure of LIBNAME and FILENAME statements, see Section 8.3.6.

### Capturing SQL Step Boundary Errors Using &SQLRC

Because PROC SQL executes at the statement level, we may need to be able to evaluate the success or failure of individual statements within a PROC SQL step. To do this we can use the automatic macro variable &SQLRC. This macro variable is reset following the execution of each PROC SQL statement.

#### Program 8.3.2e (SAS Log): Showing Errors at SQL Boundaries

```

226 proc sql ;
227 %put &sqlrc;
SQLRC=0
228 create table class as
229   select *
230     from sashelp.class; ①
ERROR: File SASHELP.CLSS.DATA does not exist.
231 %put &sqlrc;
SQLRC=8 ②
232 create table class as
233   select *
234     from sashelp.class;
NOTE: Table WORK.CLASS created, with 19 rows and 5 columns.

235 %put &sqlrc;
SQLRC=0 ③
236 quit;
NOTE: The SAS System stopped processing this step because of errors. ④

```

- ① The incoming data set name has been misspelled.
- ② &SQLRC contains a nonzero value indicating something other than success.
- ③ The data set is spelled correctly and &SQLRC contains a 0.
- ④ Being able to capture the return code within a step can become important. Notice here that although the NOTE indicates that the step was stopped, it clearly was not as the data set WORK.CLASS was created.

**SEE ALSO:** Additional detail about using automatic macro variable return and completion codes can be found in a very detailed paper by Hughes (2014b).

### Examining Errors Codes Stored in &SYSINFO

While all procedure steps can be checked using the &SYSERR macro variable, some procedures, routines, statements and functions will also provide a return code in the automatic macro variable &SYSINFO. PROC COMPARE is one of those steps.

In the COMPARE step in Program 8.3.2f two very different data sets are compared. &SYSERR detects that warnings were issued, while &SYSINFO has a more specific return code.

#### Program 8.3.2f: Examining &SYSINFO

```

proc compare data=sashelp.shoes comp=sashelp.class;
run;
%put &syserr;
%put &sysinfo;

```

The SAS Log shows the values of &SYSERR and &SYSINFO:

```
271 %put &=syserr;
SYSERR=4
272 %put &=sysinfo;
SYSINFO=3073
```

Possible values for &SYSINFO can be found in the documentation for the procedures that use this macro variable.

**SEE ALSO:** Cheng et. al. (2015) uses &SYSINFO with a PROC COMPARE step.

### 8.3.3 Taking Advantage of the Parameter Buffer

A buffer is a temporary storage location that can be used to store or pass information. Macro parameter buffers enable you to create macros with a variable number of parameters. The PARMBUFF (or PBUFF) switch is used to turn the parameter buffer on, and the automatic macro variable &SYSPBUFF is used to hold the buffer's value.

The /PARMBUFF switch is used on the %MACRO statement to turn on the ability to load the macro variable &SYSPBUFF when the macro is called. The macro %DEMO in Program 8.3.3a demonstrates the process that you will use when taking advantage of &SYSPBUFF.

#### Program 8.3.3a: Demonstrating the Use of the PARMBUFF Switch on the %MACRO Statement

```
%macro demo(a=1, b=2)/parmbuff; ①
  %put buffer holds |&sysbuff|; ②
  %put &=a; ③
  %put &=b; ④
%mend demo;

%demo (a=aa) ⑤

%demo (a=silly, d=unknown) ⑥
```

The macro %DEMO is called twice, and the SAS Log shows the following:

```
35  %demo (a=aa) ⑤
buffer holds |(a=aa)| ②
A=aa ③
B=2 ④
36
37  %demo (a=silly, d=unknown) ⑤
buffer holds |(a=silly, d=unknown)| ②
A=silly ③
B=2 ④
```

- ① The macro statement shows two keyword parameters and the /PARMBUFF switch.
- ② This %PUT writes the contents of &SYSPBUFF to the SAS Log. The value of &SYSPBUFF contains the parameters of the macro call, including the parentheses, just as they are coded. This includes extra spaces, commas, and other characters.
- ③ The value of &A has been passed into the macro as a keyword parameter, and as is usual, it is stored in the macro variable.
- ④ Since &B is not included in the macro call, its value is not included in &SYSPBUFF, and the value of &B remains at its default value.
- ⑤ The macro is called a second time, and the macro parameter list is passed to the macro and stored, including the parentheses, in &SYSPBUFF.

- ⑥ Since the parameter values being passed to the macro are coming in through the buffer, you can call the macro using parameters that do not exist. This call to %DEMO runs without error. However, the parameter &D will not be defined unless you explicitly write code to parse &SYSPBUFF. This enables you to build a macro with a variable number of parameters, and this is done in the macro %IN in Program 8.3.3c.

Although &SYSPBUFF is an automatic macro variable it is stored in the local symbol table. This means that a local symbol table will always exist when using the /PARMBUFF switch. Although on the local table, &SYSPBUFF is an automatic macro variable and can be shown using the %PUT \_AUTOMATIC\_ ; statement rather than the %PUT \_LCOAL\_ ; statement.

Program 8.3.3b shows that &SYSPBUFF is local, but it also highlights inconsistencies when using the %SYMGLOBL and %SYMLOCAL functions. These inconsistencies have been fixed in SAS 9.4M3.

#### Program 8.3.3b: Showing that &SYSPBUFF Is Local

```
%macro test/pbuff;
  %put &syspbuff;
  %put %symexist(syspbuff);
  %put %synglobl(syspbuff);
  %put %symlocal(syspbuff);
%mend test;
%test(abc)
%put &=syspbuff;
```

The SAS Log for %TEST shows that:

```
92  %test(abc)
(abc)
1
1
0
93  %put &=syspbuff;
WARNING: Apparent symbolic reference SYSPBUFF not resolved.
```

The PARMBUFF option is used in Program 8.3.3c to create a macro function that can be used to build a highly flexible IN operator for the DATA step IF statement. It enables you to check a character variable against a list of values of varying lengths. In addition, you can optionally match only the first few characters of the string. It does this by building an IF expression of the following form:

```
if (code eq 'a1' or code eq 'a2' or code eq 'a3') then....
```

The variable that is to be checked (code) is the first parameter in the macro call and the remaining parameters form the values (a1, a2, and a3).

#### Program 8.3.3c: Using PARMBUFF to Build a Flexible IN Operator

```
%macro in() / parmbuff; ①
  %local parms var numparms infunc i thisparm;
  %let parms = %qsubstr(&syspbuff,2,%length(&syspbuff)-2); ②

  %let var = %scan(&parms,1,%str(,)); ③

  %let numparms = ④ %eval(%length(&parms) -
    %length(%sysfunc(compress(&parms,%str(,)))));

  %let infunc = &var eq %scan(&parms,2,%str(,)); ⑤
```

```
%do i = 3 %to (&numparms + 1); ❶
  %let thisparm = %scan(&parms,&i,%str(,)); ❷
  %let infunc = &infunc or &var eq &thisparm; ❸
%end;

(&infunc) ❹
%mend;
```

Source: Pete Lund, Looking Glass Analytics

- ❶ The macro is specified without any parameters. The information needed by the macro will come in through &SYSPBUFF. Although the name %IN is currently not reserved (SAS 9.4), it *will* become a reserved word in the future. It is recommended that %IN not be used as a macro name so as to improve compatibility with future releases of SAS.
- ❷ The parentheses that are automatically included with &SYSPBUFF are stripped off.
- ❸ The first parameter is the variable name that will be checked.
- ❹ Count the number of parameters by counting the number of commas. In this statement the commas are counted by comparing the length of &PARM with its length after the commas have been compressed out. The number of parameters, &NUMPARMS, is one too small because there is one more parameter than there are commas and the first parameter is actually the variable name.

The number of parameters could have also been calculated using the COUNTW function, which was not available when this macro was originally written.

```
%let numparms = %sysfunc(countw(&parms));
```

- ❺ The macro variable &INFUNC will be used to hold the expression that is being built. This statement creates the first expression by equating the name of the variable, &VAR, with the first parameter value (second word in the list).
- ❻ Loop through the remaining parameters (this macro expects at least two comparison parameters).
- ❼ Select the next value from the parameter list.
- ❼ Add this comparison onto the growing list in &INFUNC.
- ❽ Use &INFUNC to pass the list of comparisons back to the IF statement.

In the call to the %IN macro below, an IF statement will be built that will check a variable (CPT) against the following character values '4300', '4301', '44xx', '451x'. Here x represents a wildcard value, which is established by placing the colon operator before those values that include partial strings. Notice that these are character values and the quotes are passed into the macro.

```
If %in(cpt,'4300','4301',:'44',:'451') then...
```

The previous macro call would generate the following code:

```
if (cpt eq '4300' or cpt eq '4301' or cpt eq :'44' or cpt eq :'451') then
...
```

Since &SYSPBUFF is being used to pass the parameters, the macro call can contain any number of comma-separated values.

The macro %ORLIST in Program 8.3.3d is similar to the macro %IN in Program 8.3.3c as it also uses the PARMBUFF switch; however, it parses &SYSPBUFF differently. The %DO %WHILE loop is used to pass through the list of parameter values and one by one the variable/value pairs are added to &ORLIST.

**Program 8.3.3d: Parsing &SYSPBUFF with the %QSCAN Function**

```
%macro ORlist() / pbuff;
  %local datvar i parm orlist;
  %let datvar = %qscan(&sysplib,1,%str(%(,)); ❶
  %let i = 1;
  %do %while(%qscan(&sysplib,&i+1,%str(,%(%))) ne %str());
    %let parm = %qscan(&sysplib,&i+1,%str(,%(%))); ❷
    %if &i=1 %then %let orlist = &datvar=&parm; ❸
    %else %let orlist = &orlist or &datvar=&parm;
    %let i = %eval(&i + 1);
  %end;
  &orlist ❹
%mend orlist;
```

The macro builds a series of logical comparisons separated by the Boolean OR operator. Typical usage would be within an IF statement. Here a %PUT is used to show in the SAS Log what the IF statement would look like after the macro is called:

```
157 %put If %orlist(cpt,'4300','4301',:'44',:'451') then...;
If cpt='4300' or cpt='4301' or cpt=:'44' or cpt=:'451' then...
```

- ❶ The variable name is retrieved as the first word. The %( is used to mark the open parenthesis as a word delimiter along with the comma.
- ❷ %QSCAN is used to separate the values. Notice the use of %( and %), as well as the comma, to designate the open and close parentheses as word delimiters (this prevents them from becoming a part of the first and last words selected by %QSCAN. (See Section 7.1.9 for a discussion of the marking of special characters.)
- ❸ The list of comparisons is temporarily stored in &ORLIST.
- ❹ The resulting list of comparisons is passed back, and replaces the macro call with the resultant list of comparisons.

**SEE ALSO:** Mace (1999) briefly discusses the automatic macro variable &SYSPBUFF. A more detailed discussion of %IN and other user-written macro functions can be found in Lund (1998, 2000a, 2000b, and 2001c). The /PARMBUFF switch and &SYSPBUFF macro variable are used by Lund (2000a) to build a formatted comment for the SAS Log.

**8.3.4 Using &SYSNOBS as an Observation Counter**

When processing a DATA step it is often handy to be able to capture the number of observations written to the new data set. The macro variable &SYSNOBS will contain the number of observations written to the last data set closed by a DATA step. In Program 8.3.4 the macro variable &SYSNOBS will contain the number of observations in WORK.WANT.

**Program 8.3.4: Using &SYSNOBS to Indicate the Number of Observations in a Data Set**

```
data want;
  set sashelp.class (where=(name>'B'));
  run;
%put &sysnobs;
```

Similar to &SQLOBS, which counts observations in SQL steps, this macro variable will be reset by the next DATA step, but it will not be reset by procedure steps, even an SQL step that creates a data table.

If your DATA step creates more than one table, the observation count of only one of the tables (the last one to be closed) is reflected in &SYSNOBS. Generally, this will be the right most table listed in the DATA statement.

**MORE INFORMATION:** The macro %OBSCNT (see Program 11.2.6) will also return the number of observations in a data set.

### 8.3.5 Using &SYSMACRONAME

The automatic macro variable &SYSMACRONAME contains the name of the most local macro that is currently executing. Section 8.1.3 examines the use of four different macro functions that can be used to surface names as well as nesting levels. However, if you only need to know the name of the innermost currently executing macro, then &SYSMACRONAME is available for your use.

Program 8.3.5 demonstrates how the value stored in &SYSMACRONAME changes depending on which macro is executing. When there are nested macro calls only the name of the inner most macro is revealed.

#### Program 8.3.5: Showing the Name of the Currently Executing Macro

```
%macro inner;
%put inner &sysmacroname;
%mend inner;
%macro test;
%put in test before inner: &sysmacroname;
%inner
%put back in test: &sysmacroname;
%mend test;
%test
%put in open code: &sysmacroname;
```

The SAS Log shows that the value of &SYSMACRONAME is updated as the macro being executed changes:

```
246 %test
in test before inner: TEST
inner INNER
back in test: TEST
247 %put in open code: &sysmacroname;
in open code:
```

**MORE INFORMATION:** Macro functions that can be used to determine macro nesting as well as the name of the currently executing macro are described in Section 8.1.3.

**SEE ALSO:** McMullen (2012) uses &SYSMACRONAME in a macro that tests data assertions.

### 8.3.6 Using &SYSLIBRC and &SYSFILRC

Whenever you attempt to create a *libref* or a *fileref* a return code is generated. You can view the success or failure of the operation by examining this return code, which is stored in either &SYSLIBRC or &SYSFILRC. Success is indicated by the return of a 0. A nonzero integer is returned when the LIBNAME or FILENAME statement is not successful.

Program 8.3.6 demonstrates various aspects of the use of the &SYSLIBRC macro variable (usage of &SYSFILRC is similar).

**Program 8.3.6: Checking the Success of a LIBNAME Statement**

```
* This library location does not exist;
libname mytemp "c:\temploc";
%put Zero is success: &syslibrc;
```

The SAS Log shows that in this usage the location 'c:\temploc' does not exist and that a nonzero value (-70008) is returned:

```
38 libname mytemp "c:\temploc";
NOTE: Library MYTEMP does not exist.
39 %put Zero is success: &syslibrc;
Zero is success: -70008
```

The LIBNAME and FILENAME functions (see Program 8.3.2d) also have a return code, however these functions do not update the corresponding automatic macro variables. These are updated only by the LIBNAME and FILENAME statements. This includes when these functions are executed using %SYSFUNC.

You can change or reset the values of these macro variables directly, however they can only be reset to integers. Fractional values are truncated, and you will generate an error if you try to insert a value that cannot be converted to a number. Interestingly, scientific notation does not generate an error, nor does it convert to the correct value.

## 8.4 Even More System Options

The macro programmer should at least be aware that there are a number of less commonly used system options that affect the operation and performance of the macro language. You can list the system options that apply to the macro language by using the GROUP= option on the PROC OPTIONS statement.

**Program 8.4: Displaying System Options Related to the Macro Language**

```
proc options group=macro;
run;
```

**MORE INFORMATION:** Some of the primary system options used with the macro language were introduced in Section 3.3. Additional system options that can be used with autocall macro libraries are discussed in Section 10.4.2.

### 8.4.1 Memory Control Options

Typically, macro symbol tables, and therefore the values of macro variables are stored in memory. When the memory required to store the value of a macro variable is not available, SAS will instead write the macro variable to a catalog (under Windows the catalog is named WORK.SAS0ST0). In this catalog each macro variable is a separate entry (that is, it has an entry type of MSYMTAB).

#### MVARSIZE

MVARSIZE specifies the maximum size that an individual macro variable can take on before it is written to disk. The default size for SAS9.4 under Windows is 64K bytes, which is also the same as the maximum size of a macro variable.

## **MSYMTABMAX**

MSYMTABMAX specifies the maximum memory that is available for all symbol tables. When this value is exceeded the macro variables are written to disk. The default size for Windows and UNIX is about 4 megabytes (one megabyte for z/OS). To improve performance increase this limit if you have either a large number of variables or if the variables themselves are large.

By adjusting the values of these options, you can control where macro variables and symbol tables will be written. Usually, these options are not of general concern, but they can be useful if you have either large symbol tables or large macro variables and you are limited either in available memory or available disk space.

**SEE ALSO:** Dilorio (1999) uses the MVARSIZE option to force macro variables into a catalog where they can be removed.

---

### **8.4.2 Preventing New Macro Definitions with NOMCOMPILE**

The MCOMPILE option should almost always be left on (its default value). When NOMCOMPILE is specified you will not be able to compile new macros. The only time I have found this option to be helpful was with an application that was being executed in a controlled environment and user-defined macros were highly discouraged.

**SEE ALSO:** Sun and Carpenter (2011) discuss the use of this option along with others when attempting to develop a controlled environment.

---

## **8.5 Even More DATA Step Functions and Statements**

The DATA step has a number of ways to interface with the macro language. Often you will use the macro language to write DATA step code; however, there are a number of DATA step tools that can be used to create macro variables and to execute macro code. The CALL SYMPUTX routine (introduced in Section 6.1) and the CALL EXECUTE routine (introduced in Section 6.5) are prime examples of DATA step routines that work with macro language elements that write to symbol tables. This section describes some other DATA step functions that you, as a macro programmer, should know.

---

### **8.5.1 DOSUBL Function**

In Section 6.5 the CALL EXECUTE routine is introduced and discussed. Of special interest are the timing issues associated with that routine. CALL EXECUTE gives us the ability to immediately execute macro statements from within the DATA step. However, because of the timing of events when using this function, the results can be ‘different’ from what you might otherwise expect (see Section 6.5.3 for more detail on timing issues).

The DOSUBL function is similar to the CALL EXECUTE routine in that it can be used to immediately submit and execute macro code from within a DATA step. However, many of the timing issues associated with the CALL EXECUTE routine are eliminated by this function. DOSUBL is not a replacement for CALL EXECUTE; rather, it is a different way of solving the problem of the execution of code from within the DATA step.

#### **SYNTAX:**

```
rc = DOSUBL(argument);
```

Program 6.5.3b was used to illustrate the timing differences between macro language elements and non-macro language elements in code submitted through CALL EXECUTE. When a macro is called through CALL EXECUTE macro code is executed immediately (for the entire macro) while non-macro code (including masked macro code) is placed in a stack for later execution. Because this dichotomy is step

independent, the behavior is very different than all other macro/non-macro executions which respect the step boundary. When using the DOSUBL function to execute a macro, the step boundaries are respected.

Program 8.5.1 repeats the example that was used for CALL EXECUTE in Program 6.5.3b, except that DOSUBL is used instead of CALL EXECUTE.

#### Program 8.5.1: Event Timing Associated with the DOSUBL Function

```
%macro test;
data _null_; ③
  put 'Calling SYMPUTX';
  call symputx('x3',100);
  run;

%put Ready to compile DATA step for NEW; ④
data new; ⑦
  %put Compiling NEW; ⑤
  put 'Executing NEW';
  y = &x3; ⑥
  run;
title 'Data NEW'; ⑧
proc print data=new;
  run;
%mend test;

data _null_; ①
  rc= dosubl('%test'); ②
  put rc=; ⑨
  run;
```

- ➊ During DATA step execution the DOSUBL function calls the macro %TEST. The calling DATA step is suspended and %TEST is immediately executed.
- ➋ The DOSUBL function contains a call to the %TEST macro. Notice that %TEST is enclosed in single quotes.
- ➌ The DATA step in %TEST is immediately executed and the macro variable &X3 is defined and given the value of 100. If %TEST had been called using CALL EXECUTE, this DATA step would have been placed in a stack for execution after the calling DATA step had completed execution, and &X3 would remain undefined until then.
- ➍ The %PUT is executed after the DATA step completes. If a CALL EXECUTE had been used, this %PUT would have executed before the preceding DATA step.
- ➎ This %PUT is executed as the DATA step (⑦) is being compiled.
- ➏ The macro variable &X3 is resolved during the compilation of the DATA step.
- ➐ After compilation the DATA step is executed.
- ➑ The title is defined and the PROC PRINT executes.
- ➒ The calling DATA step resumes execution and the PUT executes. RC=0 indicates that the DOSUBL executed successfully.

```
Calling SYMPUTX ③
NOTE: DATA statement used (Total process time):
      real time          0.03 seconds
      cpu time          0.01 seconds

Ready to compile DATA step for NEW ④
Compiling NEW ⑤
Executing NEW
NOTE: The data set WORK.NEW has 1 observations and 1 variables. ⑦
NOTE: DATA statement used (Total process time):
      real time          0.01 seconds
```

```

cpu time          0.01 seconds

NOTE: Writing HTML Body file: sashtml.htm
NOTE: There were 1 observations read from the data set WORK.NEW. ❸
NOTE: PROCEDURE PRINT used (Total process time):
      real time          0.46 seconds
      cpu time           0.29 seconds

rc=0 ❹
NOTE: DATA statement used (Total process time): ❺
      real time          0.73 seconds
      cpu time           0.40 seconds

```

If you want to execute a macro from within a DATA step and the macro is not specifically designed to execute with CALL EXECUTE, consider using DOSUBL.

**SEE ALSO:** The documentation for DOSUBL has a nice example that shows how one of the limitations of CALL EXECUTE can be overcome by using DOSUBL. Henderson (2014) and Parker (2015) both use the DOSUBL function to generate code for PROC STREAM.

### 8.5.2 Deleting Macro Variables with CALL SYMDEL

When executing within the DATA step it is possible to delete macro variables from the global symbol table through the use of the CALL SYMDEL routine. Much like the %SYMDEL macro statement (see Sections 2.7 and 8.2.1), this routine only deletes macro variables from the global symbol table.

#### SYNTAX:

```
CALL SYMDEL(macrovariablename<, NOWARN>);
```

The use of the CALL SYMDEL routine is shown in Program 8.5.2, which creates both local and global macro variables and then attempts to delete them using CALL SYMDEL. A warning is issued when an attempt is made to delete a macro variable that does not exist. The optional second argument can be set to NOWARN, which eliminates this warning.

#### Program 8.5.2: Using the CALL SYMDEL Routine

```

%global City;
%let city = Los Angeles; ❶
%macro test;
%local city; ❷
%let city=Anchorage;
%let state=Alaska;
data _null_;
  put 'Delete Global &CITY';
  call symdel("city"); ❸
  put 'There is no Global &CITY to delete';
  call symdel('city'); ❹
  put '&state does not exist in the global table';
  call symdel('state','nowarn'); ❺
  run;
/* show that the local version of &CITY still exists;
%put Within TEST &=city; ❻
%mend test;
%test
/* is there a global version of &CITY?;
%put &=city; ❾

```

- ❶ The macro variable &CITY is established in the global symbol table.
- ❷ A local version of &CITY is also established.
- ❸ The global version of &CITY is deleted.
- ❹ The second attempt to delete &CITY will result in a warning, since &CITY no longer exists on the global table. The local version of &CITY is ignored.
- ❺ &STATE only exists on the local table, but the NOWARN option prevents the warning from being issued in the SAS Log.
- ❻ Show that the local version of &CITY remains unaffected.
- ❼ Show that the global version of &CITY has been eliminated

#### Program 8.5.2 (SAS Log): Showing Results of the Use of CALL SYMDEL

```

Delete Global &CITY ❸
There is no Global &CITY to delete
WARNING: Attempt to delete macro variable CITY failed. Variable not found.

❹
&state does not exist in the global table ❺
NOTE: DATA statement used (Total process time):
      real time          0.01 seconds
      cpu time           0.00 seconds

Within TEST CITY=Anchorage ❻
WARNING: Apparent symbolic reference CITY not resolved. ❼
114  %* is there a global version of &CITY?;
115  %put &=city;
      city

```

**MORE INFORMATION:** The %SYMDEL macro statement is introduced in Section 2.7.

### 8.5.3 Using SYMEXIST, SYMGLOBL, and SYMLOCAL

If your DATA step is going to create a macro variable, it could be important to know if that macro variable already exists in either a local or global symbol table. Each of these functions can assist with that determination.

#### SYNTAX:

```

SYMEXIST(macrovariablename)
SYMLOCAL(macrovariablename)
SYMGLOBL(macrovariablename)

```

#### VALUES RETURNED:

Each of these functions returns either a

1	macro variable is found
0	macro variable is not found

The SYMEXIST function only will tell you whether the macro variable exists in some scope. SYMGLOBL (note the spelling of this function) checks only the global symbol table, while SYMLOCAL checks each of the local symbol tables.

**Program 8.5.3: Detecting Macro Variables and Their Scope**

```
%global City state; ①
%let city = Los Angeles;
%let state= CA;
%macro test;
%local city; ②
%let city=Anchorage;
data _null_;
  var='state'; ③
  if symexist("city") then do;
    if symlocal('city') then put 'macro variable city exists locally'; ④
    if symglobl('city') then put 'macro variable city exists globally'; ⑤
    if symglobl(var) then put 'macro variable ' var ' exists globally'; ⑥
  end;
  run;
%mend test;
%test
```

In the DATA step in %TEST a check is made to determine if a macro variable exists in some table and if it does secondary checks are used to determine if it is a local or global table. In this case &CITY is in both symbol tables. The SAS Log shows that all three IF statement expressions are true.

```
214 %test

macro variable city exists locally ④
macro variable city exists globally ⑤
macro variable state   exists globally ⑥
```

- ① The macro variables &CITY and &STATE are added to the global symbol table.
- ② The macro variable &CITY is also added to the local table.
- ③ The DATA step variable VAR is created with the value of 'state'.
- ④ %SYMLOCAL shows that there is a local version of the macro variable &CITY.
- ⑤ %SYMGLOBL shows that there is a global version of the macro variable &CITY.
- ⑥ A variable name (VAR) that resolves to the name of a macro variable (STATE) is used to show that &STATE exists on the global table.

Remember that these are DATA step functions and as such they work with strings that take on the names of macro variables or variables that resolve to the name of a macro variable.

**MORE INFORMATION:** Section 8.1.5 discusses the macro versions of these functions. SYMEXIST is used in Program 9.3a to check for the existence of a macro variable before retrieving it using SYMGET.

# **Chapter 9: Exploring Some Less Common Intermediate Topics**

<b>9.1 Building Macro Calls .....</b>	<b>231</b>
9.1.1 Building Macro Calls %&name .....	231
9.1.2 Calling Macros from the Display Manager .....	233
<b>9.2 Working with Macro Variables .....</b>	<b>236</b>
9.2.1 Determining Macro Variable Existence and Scope .....	236
9.2.2 Creating a Large Number of Macro Variables.....	238
<b>9.3 Using the Macro Language to Form Simple Hash Tables.....</b>	<b>241</b>
<b>9.4 Using the Macro Language for Formatted Table LookUps .....</b>	<b>243</b>
<b>9.5 Making Comparisons to Null Values—Some Considerations .....</b>	<b>244</b>
<b>9.6 Evaluating Expressions Stored in a Data Set.....</b>	<b>245</b>
<b>9.7 Using Macro Language Elements on Remote Servers .....</b>	<b>246</b>
<b>9.8 Working with Macro Variables That Contain Special Characters .....</b>	<b>248</b>
9.8.1 Quoting Review.....	248
9.8.2 The Problem with Quotes.....	248
9.8.3 Ampersands and Percent Signs .....	249
9.8.4 Lists and Nested Functions—The Comma Problem.....	251

Although less commonly encountered, there are a number of topics related to the macro language that can cause the unsuspecting macro programmer a fair amount of grief when they unexpectedly either show up in a program that you are working on or when you need to deal with one of these situations.

This chapter deals with some of these topics and also introduces a few topics that a strong macro programmer needs to know.

Most of the example macros in this chapter have been simplified to show techniques, and are generally not practical in and of themselves.

For the examples in this chapter and indeed for all of the code examples throughout the book, if you want to execute these sample programs, then be sure to follow the setup instructions. Remember that all of the data sets and programs are available for download, so you do not need to retype either the code or the data. For instructions on accessing and setting up the programs and data, see the “Example Code and Data” section within this edition’s “About This Book” front matter.

---

## **9.1 Building Macro Calls**

Usually, a macro call is made from within a SAS program, and the call is coded directly as in most of the examples in this book. There are times, however, when the macro call is not that straightforward.

---

### **9.1.1 Building Macro Calls %&name**

Generally, when you are coding you will know the name of the macro that you want to have executed. However, occasionally the name of the macro to be called will depend on information that is not known until the execution of your program.

Although this is not a usual coding solution, the macro call itself can contain the name of a macro variable, which stores all or part of the macro name. One scenario occurs when there are multiple versions of a macro available for use, and they need to be distinguished. The macro call in Program 9.1.1a could be used to invoke the %DEBUGNEW macro.

#### **Program 9.1.1a: Building a Macro Call with a Macro Variable**

```
%let version = new;
%debug&version
```

Fortunately, the macro variable &VERSION is resolved before the macro is called, resulting in the macro call %DEBUGNEW.

In some versions of SAS, the macro processor will try to execute the %DEBUG macro before resolving the macro variable &VERSION. If you encounter this problem, one potential solution is to immediately follow the macro call with a semicolon. Another solution is use quoting functions to force the resolution of the macro variable first. %NRSTR can be used to mask the %DEBUG until after the &VERSION has resolved.

```
%unquote(%nrstr(%debug) &version)
```

Some macro programmers feel that this resolution sequence is backwards and they would prefer that the %DEBUG execute first so that they can append the resolved value of &VERSION onto the resultant code of %DEBUG. This can also be accomplished with macro quoting functions. This time %NRSTR is used to mask the macro variable until after the macro has been called.

```
%debug%unquote(%nrstr(&version))
```

In the example shown in Program 9.1.1b the name of the macros to be called depend on values in a data set variable. Each of the distinct values of the variable ORIGIN correspond to a macro of the same name. The macro call (%&&ORIG&I) will not be finalized until the macro variable reference (&&ORIG&I) is first resolved.

#### **Program 9.1.1b: Naming a Macro Call Using Variable Values**

```
%macro regions;
%local i;
proc sql noprint;
  select distinct origin
    into :orig1-
      from sashelp.cars;
quit;

%do i = 1 %to &sqllobs;
  %&&orig&i
%end;
%mend regions;
%regions
```

For the SASHELP.CARS data set, the macro calls will be:

- %ASIA
- %EUROPE
- %USA

**MORE INFORMATION:** This technique is used in Program 12.4.8a to make automated data corrections.

### 9.1.2 Calling Macros from the Display Manager

When you find yourself executing macros interactively on a regular basis, you may find that you can automate the macro calls through either the Display Manager or SAS Enterprise Guide. The following examples are for the Display Manager, however the steps are similar for SAS Enterprise Guide.

#### Adding a Macro Call to a Function Key

Most of the function keys can be assigned a Display Manager command, macro statement, or macro call. Many of the function keys come predefined, however you can either add a command to a key that is without an assignment or you can overwrite an existing command for any of the keys.

The KEYS window is by default displayed with the F9 key. In the key definitions shown in Figure 9.1.2a, the Shift-F11 key has been assigned a macro %PUT statement and the Shift-F12 key has been assigned a macro call to the %BESTEVER macro (see Program 9.1.2a).

**Figure 9.1.2a: Adding a Macro Call to a Function Key**

Key	Definition
F1	help
F2	reshow
F3	end; /*gsubmit but
F4	recall
F5	wpgm
F6	log
F7	output
F8	zoom off;submit
F9	keys
F11	command focus
F12	
SHF F1	subtop
SHF F2	
SHF F6	
SHF F7	left
SHF F8	right
SHF F9	
SHF F10	wpopup
SHF F11	%put _user_;
SHF F12	%bestever

Once assigned the user can execute the %BESTEVER macro simply by selecting Shift-F12.

#### Calling a Macro with an Argument

The calling of a macro through the use of a function key, like was done for the %BESTEVER macro, is handy as long as your macro does not have any arguments. The %SHOWRPT macro shown in Program 9.1.2b uses the LIST option in a PROC REPORT step to display the default code used in the PROC REPORT step (this can save a fair amount of typing if the code is then copied from the SAS Log into the Program Editor).

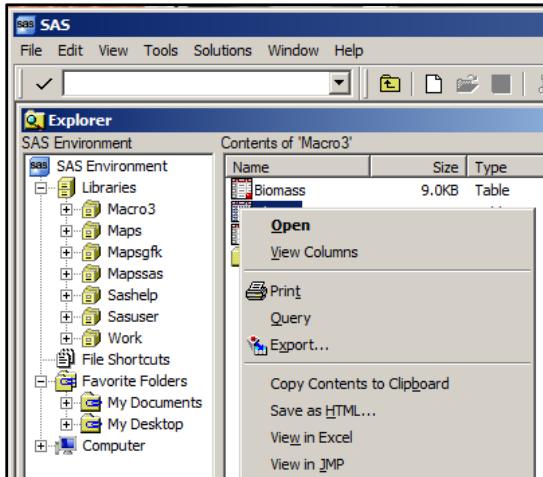
**Program 9.1.2b: Displaying PROC REPORT Code**

```
%macro showRPT(dsn=sashelp.class);
proc report data=&dsn list;
run;
%mend showrpt;
```

We would like to have a call to the %SHOWRPT macro available for any given SAS data set from within the Display Manager. We do this by adding the macro call to one of the DM's pop-up menus.

The SAS Explorer is used to find and select SAS controlled items, including data sets. If you right-click on a data set, a pop-up menu (see Figure 9.1.2b) is displayed with the options that can be applied to that data set. Notice that these options include printing and exporting. In the example that follows, this menu will be modified to include a macro call.

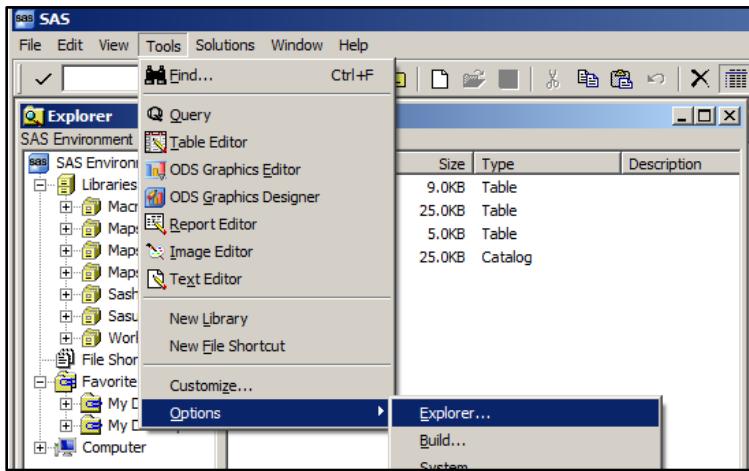
**Figure 9.1.2b: SAS Explorer Pop-Up Menu without Modifications**



Fortunately, it is quite easy to not only access the code that builds this menu, but it is also fairly easy to modify the menu elements. In this example we are going to add a call to the %SHOWRPT macro. First we need to access the code behind the menu.

We do this by bringing up the options that are associated with the SAS Explorer. With the SAS Explorer the active window, select Tools, Options, and Explorer are shown in Figure 9.1.2c.

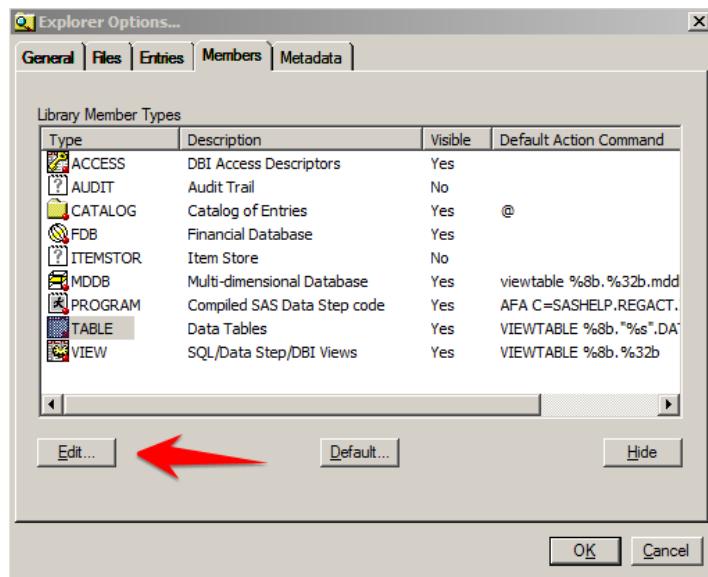
**Figure 9.1.2c: Accessing the SAS Explorer's Options**



This gives us access to types of SAS controlled entities that can be controlled by the SAS Explorer.

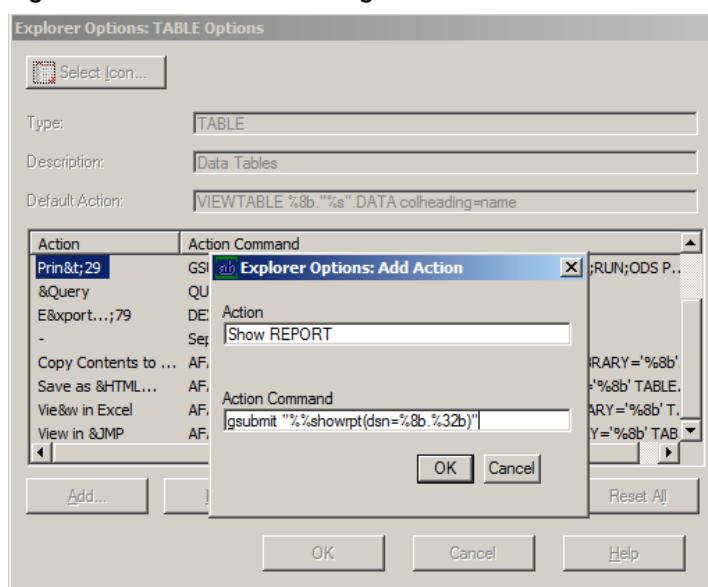
The members tab (Figure 9.1.2d) surfaces the types of entities that you can select from the SAS Explorer. We are interested in SAS data sets (tables), so the TABLE entry is selected. Once a selection is made the “Edit” button can be selected next.

**Figure 9.1.2d: SAS Explorer Options**



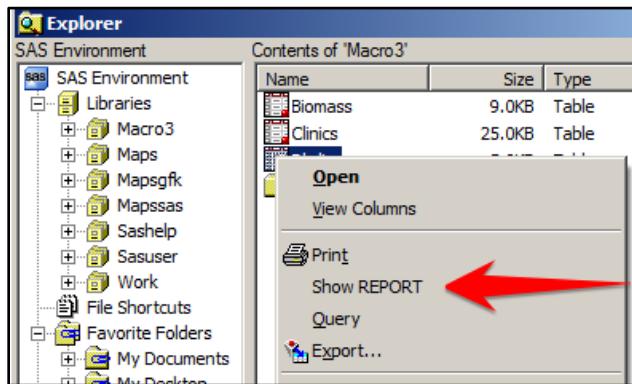
The “Edit” button brings up the definitions of the script that sits behind the pop-up menus. We want to add a new menu instruction below the PRINT, so we highlight PRINT and select ADD, a ‘Add Action’ dialog box appears (see Figure 9.1.2e). In this dialog box we add the display text (Show REPORT) and the action command, which in our case is the macro call that we want executed (gsubmit “%%showrpt (dsn=%8b.%32b)”). Notice that the data set name uses a scripting code; where the libref = %8b, and the member name=%32b. A double percent sign is used in front of the macro call to delay its execution until after the data set name scripts have been resolved.

**Figure 9.1.2e: Add Action Dialog Box**



After entering the script on the ‘Action Command’ line, we press OK, and work our way back out of the menus. Now when we right-click on a data set our pop-up menu will include a selection for showing the report (see Figure 9.1.2f). The selected data set is passed to the %SHOWRPT macro for execution.

**Figure 9.1.2f: Executing the %SHOWRPT Macro from the Pop-Up Menu**



Assuming that the macro definition is available, perhaps in the autocall macro library, the PROC REPORT step shown in Program 9.1.2b will be executed and the expanded PROC REPORT step code will be displayed in the SAS Log for Program 9.1.2b.

**Program 9.1.2b (SAS Log): Execution of Program 9.1.2b on the MACRO3.DBDIR Data Set**

```
8 proc report data=Macro3.Dbdir list; run;
NOTE: Writing HTML Body file: sashtml.htm

PROC REPORT DATA=MACRO3.DBDIR LS=95 PS=53 SPLIT="/" CENTER ;
COLUMN dsn Keyvar;

DEFINE dsn / DISPLAY FORMAT= $8. WIDTH=8 SPACING=2 LEFT "dsn" ;
DEFINE Keyvar / DISPLAY FORMAT= $20. WIDTH=20 SPACING=2 LEFT "Keyvar"
;
RUN;
```

**SEE ALSO:** See Section 14.4.6 in *Carpenter's Guide to Innovative SAS® Techniques* (2012) for more detail on adding macro calls to Display Manager pull-down menus.

## 9.2 Working with Macro Variables

Throughout this book we have been using macro variables in a variety of different ways, but of course this is SAS, so there are a number of other ways to create and use our macro variables. This section highlights a few of these alternatives.

### 9.2.1 Determining Macro Variable Existence and Scope

Sometimes you need to be able to determine whether a macro variable has been created. In addition, you might also need to know in which symbol table the macro variable resides. The %SYMEXIST, %SYMGLOBL, and %SYMLOCAL functions, which are introduced in Section 8.1.5, can be used to help answer these types of questions

The argument for each of these functions is either a macro variable name (without the ampersand), or a text expression that resolves to a macro variable name. All three of these functions return a 1 (true) if the macro

variable exists and a 0 (false) if it does not exist. Program 9.2.1a is a test program to demonstrate the use of these three functions.

#### Program 9.2.1a: Determining Existence and Scope of a Macro Variable

```
%let inglobal = abc; ❶
%macro test;
  %let inlocal = def; ❷
  %let lvar = inlocal;
  %if %symexist(notavar)=0 %then %put NOTAVAR does not exist; ❸
  %if %sympglobl(inglobal) %then %put INGLOBAL in global; ❹
  %if %symlocal(inlocal) %then %put INLOCAL in local; ❺
  %if %symlocal(&lvar) %then %put INLOCAL in local (using LVAR);
  %if %sympglobl(inlocal)=0 %then %put INLOCAL does not exist in global;
  %inner
  %if %symlocal(inner)=0 %then %put INNER is not local; ❻
%mend test;
%macro inner;
  %let inner=in_inner;
  %if %symlocal(inlocal) %then %put INLOCAL in INNER; ❼
%mend inner;
%test
```

- ❶ &INGLOBAL is added to the global symbol table
- ❷ &INLOCAL and &LVAR are defined and expected to be in the local symbol table
- ❸ This macro variable is unknown.
- ❹ &INGLOBAL is global in scope as we had expected
- ❺ &INLOCAL is in the local scope as we had expected
- ❼ %SYMLOCAL even detects a macro variable in a higher local scope
- ❻ Once a local table is closed, %SYMLOCAL successfully determines that the macro variable is no longer in the local scope. At this point %SYMEXIST would also fail to find this macro variable.

#### Program 9.2.1a (SAS Log): Execution Results of Program 9.2.1a

```
124  %test
NOTAVAR does not exist ❸
INGLOBAL in global ❹
INLOCAL in local ❺
INLOCAL in local (using LVAR)
INLOCAL does not exist in global
INLOCAL in INNER ❼
INNER is not local ❻
```

The %SYMLOCAL function detects a macro variable in any local symbol table, but does not tell you if a macro variable is in a specific symbol table. The %SYMCHECK macro in Program 9.2.1b enables you to check for a specific macro variable in a specific local (or global) table. This macro, which acts like a function, returns true (1) or false (0) depending on whether the macro variable exists in a specific symbol table.

#### Program 9.2.1b: Checking for a Macro Variable in a Specific Symbol Table

```
%macro symcheck(mscope,mvname);
  %local fetchrc dsnid rc;
  %let rc = 0;
  %let dsnid = %sysfunc(open(sashelp.vmacro ❸
                            (where=(scope=%upcase("&mscope") and
                                   name=%upcase("&mvname"))),i));
  %let fetchrc = %sysfunc(fetch(&dsnid,noget)); ❹
```

```
%if &fetchrc eq 0 %then %let rc=1; ⑩
%let dsnid = %sysfunc(close(&dsnid)); ⑪
&rc
%mend symcheck;
```

- ⑧ The OPEN function enables the user to work with a data set outside of the DATA step. Once opened, the data set, its data, and its metadata can be accessed through a series of additional functions. Each of these functions, such as the FETCH ⑨ and CLOSE ⑩ functions used in this example, use the data set ID value returned by the OPEN function (stored in &DSNID in this example).  
The SASHELP.VMACRO view is opened with a specific WHERE clause. Typically, this view has one row per macro variable across all symbol tables. However, macro variables containing more than 200 bytes are displayed on multiple rows in 200-byte chunks, with the variable OFFSET used as an indicator that the macro variable's value was split up in the view.
- ⑨ The FETCH function attempts to read a row from the view, given the WHERE clause.  
⑩ The read can only be successful (FETCHRC=0) if the selected macro variable and scope exists.  
⑪ The CLOSE function closes an opened data set. If you do not close an opened data set, it will not be available for other operations.

A typical usage of %SYMCHECK appears below, where a check is made on the existence of &PATH in the global symbol table.

```
%if %symcheck(global,path) %then
  %put The macro variable PATH exists in the Global symbol table;
```

**MORE INFORMATION:** The SET routine and the FETCH functions are also used in Program 9.2.2d.

**SEE ALSO:** Vijayarangan (2016) uses a number of DATA step functions to open and read from SAS data sets.

## 9.2.2 Creating a Large Number of Macro Variables

Macro variable list processing is discussed in a number of places in this book. In those examples the list of values are assigned to a series of macro variables with a common root name. Sometimes we need to create a list of macro variables; however, their names must take on specific values.

The techniques shown in this section can be used as the starting point for the development of simple hash tables (Section 9.3) and formatted lookups (Section 9.4).

The simplest form is of course a list of %LET statements. In Program 9.2.2a five %LET statements are used to create five macro variables.

### Program 9.2.2a: Creating a Named List Using %LET Statements

```
%let Name    = Alfred;
%let Sex     = M;
%let Age     = 14;
%let Height  = 69;
%let Weight  = 112.5;
```

As long as the list is fairly short this is an easy solution, but what if the list is very long or perhaps the full list of variables is not known, then this solution becomes impractical.

### Building the List from a Data Set

When the list of macro variables and their values reside in a data set, reading the list into a series of macro variables is straightforward. In Program 9.2.2b the same list as was used in Program 9.2.2a is contained in the data set WORK.VALUES. This example simulates whenever a data set contains two variables, where one column contains the macro variable name and the other contains the corresponding macro variable value.

**Program 9.2.2b: Building a List of Macro Variables from a Data Set**

```

data values;
input mvar $ value $; ①
datalines;
Name    Alfred
Sex     M
Age     14
Height  69
Weight  112.5
run;
%macro mvarlist;
data _null_;
  set values;
  call symputx(mvar,value,'l'); ②
  run;
* The macro variables are available for use here;
%put _local;
%mend mvarlist;
%mvarlist

```

- ① The name of the macro variable is stored in the variable MVAR, while the variable VALUE contains the value for that macro variable.
- ② The SYMPUTX routine is used to create the macro variables. Notice that the first argument (MVAR) is not quoted. Two DATA step variables will be used to create each macro variable. Because we do not know the names of the macro variables when writing the macro, we cannot use a %LOCAL statement to force the new macro variables onto the local symbol tables. Instead, this is accomplished by specifying an 'L' in the third argument of the SYMPUTX.

One macro variable is created for each observation in the data set WORK.VALUES.

**Using DATA Step Variable Names as Macro Variables**

In the previous example the name of the macro variable was stored in a DATA step variable. The other common scenario occurs when our macro variables take on the same name of the variables in our DATA step, and to complicate things just a bit let's make each a list. Effectively we are going to store the entire data set in a series of macro variables.

In Program 9.2.2c, we are going to assume that we do not know the names of the variables in the data set, but we want to create a series of macro variables for each column using the name of the data set variable as the root of the macro variable list.

**Program 9.2.2c: Using DATA Step Variable Names**

```

%macro M_all_data(dsn=);
%local nobs;
data _null_;
  set &dsn;

  * Variable Arrays;
  array _nums {*} _numeric_; ③
  array _char {*} $ _character_;

  if dim(_nums) do i = 1 to dim(_nums);
    call symputx(catt(vname(_nums{i})④),_n_⑤,_nums{i}⑥,'l');
  end;
  if dim(_char) do i = 1 to dim(_char);
    call symputx(catt(vname(_char{i})),_n_,_char{i},'l');
  end;
run;
%let NObs = &sysnobs; ⑦

* Your process which utilizes these macro variables goes here;

```

```
%put _local_; ③
%mend m_all_data;
%m_all_data(dsn=sashelp.class)
```

- ③ Two arrays are created, one for numeric and one for character variables. Since we do not know the names of the variables the variable naming shortcuts \_NUMERIC\_ and \_CHARACTER\_ are used. This solution does not assume that there is at least one numeric and one character variable in the data set of interest, however warnings will be issued if either array has zero elements.
- ④ The physical name of the variable can be retrieved from the PDV through the use of the VNAME function.
- ⑤ The observation number is appended onto the macro variable name, using the CATT function, which automatically converts the numeric value of \_N\_ to a character value (without generating a note in the SAS Log).
- ⑥ The value of the variable is retrieved from the array and used as the second argument in the SYMPUTX routine.
- ⑦ The total number of observations in the data set is saved in &NOBS.
- ⑧ Your user process (in this example it is simulated with a %PUT) goes here. At this point the entire data set is available in macro variables.

### Loading the Macro Variable Lists Directly Using a Macro Function

The solution used in Program 9.2.2c uses a DATA step and as such the macro %M\_ALL\_DATA can't be used as a macro function. Fortunately, the macro language also has the ability to pull information not only from the metadata of a SAS data set, but from the data itself. In Program 9.2.1b the FETCH function is used to read information contained in a VIEW. In the solution shown in Program 9.2.2d the SET routine and the FETCHOBS function will be used to create a series of macro variables for each observation in a data set.

#### Program 9.2.2d: Auto Generating a List of Macro Variables Based on Data Set Variables

```
%macro M_all_data(dsn=);
%local dsid i nobs;
%let dsid = %sysfunc(open(&dsn)); ①

%let nobs= %sysfunc(attrn(&dsid,nlobs)); ②
%syscall set(dsid); ③
%do i = 1 %to &nobs; ④
    %let rc=%sysfunc(fetchobs(&dsid,&i)); ⑤
    /* Local process goes here; ⑥
    %put ***** Observation &i ****;;
    %put _local_;
%end;
%let dsid = %sysfunc(close(&dsid)); ⑦
%mend m_all_data;

%m_all_data(dsn=sashelp.class)
```

- ① The data set of interest is opened and assigned an identification number, which is stored in &DSID.
- ② The ATTRN function returns numeric attributes of the data set. In this instance we are requesting the number of non-deleted observations by specifying NLOBS. The number of observations is stored in &NOBS.
- ③ The SET routine sets up an association between the variables in the data set with macro variables of the same name. The macro variables are not populated at this point.
- ④ A %DO loop is used to step through the observations of the data set.

- ➅ The FETCHOBS function reads the observation associated with the number stored in &I (the %DO loop index).
- ➆ This macro does not create a list of values for each observation. Instead, the series of macro variables will be replaced for each observation, and the user-defined process that uses these macro variables (probably a called macro), will take place within the confines of this %DO loop, which is defined at ➄. In this example, the local process is simulated with %PUT statements.
- ➇ When the macro has completed accessing the data set, the data set is closed so that it can be used by another process.

**MORE INFORMATION:** Examples that use these types of macro variable lists can be found in Chapters 11 and 12. Program 9.2.1b uses the FETCH function.

**SEE ALSO:** Murphy (2007) expands on examples similar to those shown here. Parker (2014) uses the ATTRN function along with OPEN and CLOSE functions to retrieve metadata information from a data set. Michelsen (2014) uses the FETCHOBS function to read data from a data set.

### 9.3 Using the Macro Language to Form Simple Hash Tables

When we think of an array, we generally envision a list of values that are associated by some name and that are indexed. In DATA step arrays, the list is often a set of variables, with an array name declared on the ARRAY statement, and we can then step through this list using a numeric index. A hash table is similar to an array in that it is used to collect a series of values, however it can be indexed using nonnumeric values.

In Section 9.2.2 a series of macro variables that contain values taken from the data, and the macro variables themselves are named based on the data as well. We can expand this concept to form a simple hash table, which can be used as mechanisms for very efficient memory resident retrievals. The examples shown in this section are trivial; however, even searches in the hundreds of thousands of items would be very fast.

The simplest example of these techniques can be found in Program 6.4.2c. There the macro variable name and its value are both based on information stored in the data table. In Program 9.3a this concept is expanded to store a series of values based on a single ‘index’ value (the student name).

#### Program 9.3a: Single Index Value Hash Storing Multiple Values

```
data _null_;
  set sashelp.class;
  call symputx(name ①,catx(' ', age, sex, height, weight②));
run;
```

- ➊ The name of the macro variable will be the student’s name.
- ➋ The macro variable will contain a series of ‘|’ separated values associated with that student.

The SAS Log shows the following:

```
138  %put &=alice;
ALICE=13|F|56.5|84
139  %let alice_wt = %scan(&alice ③,4,%str());
140  %put &=alice_wt;
ALICE_WT=84
```

- ➌ In this example the student’s name becomes the index to the information stored in memory about that student.

We can of course accomplish this same information retrieval system without using macros at all. In the DATA step we could use arrays or a hash object, however arrays and hash objects do not survive the

DATA step boundary. One advantage of using macro variables in this method is that once it is loaded into memory, the information can be retrieved any number of times. Even across step boundaries.

In a later step, if we need a student's age, sex, height, or weight we merely need to extract it from our memory resident table. In the following DATA step, three student names are used to retrieve height information from the appropriate macro variables.

#### Program 9.3a (Continued): Retrieving Information from the Memory Resident Table

```
data student_ht;
input student $;
if symexist(student)④ then height = scan(symget(student), 3, '|')⑤;
else height=' ';
datalines;
alice
fred
barbara
run;
```

- ④ For each student name that we check for a corresponding macro variable that would have been created during an earlier step ①.
- ⑤ When the macro variable exists, we retrieve the information that it contains using the SYMGET and SCAN functions. We must use SYMGET to reference the macro variable because we do not know the name of the macro variable until DATA step execution and the value of STUDENT is determined.

In Program 9.3a each name has one macro variable associated with it and that macro variable contains a list of values that must be parsed. Although I find it to be less practical, you could also create a list of macro variables for each name. In Program 9.3b one macro variable is created for each combination of student name and item of interest (AGE, SEX, HEIGHT, and WEIGHT).

#### Program 9.3b: Named Macro Variables Lists with One Value per Macro Variable

```
data _null_;
set sashelp.class;
call symputx(catx('_',name,'age'),age,'l');
call symputx(catx('_',name,'sex'),sex,'l');
call symputx(catx('_',name,'height'),height,'l');
call symputx(catx('_',name,'weight'),weight,'l');
run;
```

The macro variables created in this DATA step are in the form of '*name\_variable*'. We can show a couple of these values in the SAS Log by using the %PUT.

```
171 %put &=alice_height;
ALICE_HEIGHT=56.5
172 %put &=alfred_sex;
ALFRED_SEX=M
```

Using this technique you can instantly retrieve the value associated with any of the saved variables for any student.

In the two previous examples the list of values has essentially a single index variable (student name). In Program 9.3c, both first and last name are used together to form a compound index using the same technique as was applied in Program 9.3b. The first and last names of the patient are concatenated with an underscore to name the macro variable (*last\_first*) that will contain the various values that are to be stored.

#### Program 9.3c: List of Macro Variables Indexed by First and Last Names

```
data _null_;
set macro3.clinics;
call symputx(⑥catx('_',lname, fname), catx('|', sex, dob, ht, wt⑦));
run;
```

- ⑥ The name of the macro variable is constructed from the last and first names of the patient.
- ⑦ The value consists of the concatenated values of the DATA step variables of interest.

For any given patient, we can retrieve the values of the variables that we have saved. The information for Robert Pope is retrieved from the macro variable &POPE\_ROBERT.

```
173 %put &=pope_robert;
POPE_ROBERT=M|-1060|72|158 ❸
174 %put Birthdate for Robert Pope:
%sysfunc(putn(%scan(&pope_robert,2,%str()),date9.)); ❹
Birthdate for Robert Pope: 05FEB1957
```

- ❸ Robert Pope's date of birth is the second item in the list of values stored.
- ❹ The date (-1060) is retrieved and converted to a calendar date using the PUTN function and the DATE9. format.

**MORE INFORMATION:** Program 6.4.2c introduces the topic of creating a simple hash by generating a list of named macro variables that are retrieved using the SYMGET function.

## 9.4 Using the Macro Language for Formatted Table Lookups

A table lookup is performed when one value is used to determine a second value. One common form of lookup is through the use %IF-%THEN/%ELSE statements; however, the use of formats to perform lookup operations are more efficient. Although less commonly used, it is possible to use formatted lookups within the context of the macro language.

Program 9.4a is taken from a larger macro, and in this portion a list of %IF-%THEN/%ELSE statements are used to match a city name (&SITE) to a city code (&SCODE).

### Program 9.4a: Using %IF-%THEN/%ELSE to Perform a Lookup

```
%macro findsite(site=Ketchikan);
%LET SITE=%UPCASE(&SITE);
%if      &SITE EQ KETCHIKAN %then %let SCode=A_KTN_;
%else %if &SITE EQ POW %then %let SCode=B_POW_;
%else %if &SITE EQ PETERSBURG %then %let SCode=C_PTG_;
%else %if &SITE EQ WRANGELL %then %let SCode=C_WRG_;
%else %if &SITE EQ SITKA %then %let SCode=D_SIT_;
%else %if &SITE EQ JUNEAU %then %let SCode=E_JNU_;
%else %if &SITE EQ GUSTAVUS %then %let SCode=F_GUS_;
%else %if &SITE EQ ELFIN COVE %then %let SCode=G_ELF_;
%else %if &SITE EQ YAKUTAT %then %let SCode=H_YAK_;
%else %let SCode=UNK_;
%put &=site &=scode;
%mend findsite;
%findsite(site=Yakutat)
```

Although they execute quickly, code that includes long lists of these types of %IF-%THEN/%ELSE statements can be difficult to maintain and potentially difficult to debug. When performing these kinds of lookup operations, formats can be used to not only improve performance, but also to make code that is easier to write and maintain. The user-defined formats are created using PROC FORMAT just as you would for formats used in formatted DATA step lookups. Actually, the same formats can be used for lookups in both the macro language as well as more traditional uses in Base SAS.

The format is created using PROC FORMAT and then used as an argument in the PUTC function in Program 9.4b.

**Program 9.4b: A User-Defined Format Performs a Formatted Lookup in the Macro Language**

```

proc format;
  value $sitecode
    "KETCHIKAN" = 'A_KTN_'
    "POW"       = 'B_POW_'
    "PETERSBURG" = 'C_PTG_'
    "WRANGELL"   = 'C_WRG_'
    "SITKA"      = 'D_SIT_'
    "JUNEAU"     = 'E_JNU_'
    "GUSTAVUS"   = 'F_GUS_'
    "ELFIN COVE" = 'G_ELF_'
    "YAKUTAT"    = 'H_YAK_'
    other        = 'UNK_';
  run;

%macro findsite(site=Ketchikan);
%LET SITE=%UPCASE(&SITE);
%let scode = %sysfunc(putc(&site,$sitecode.));
%put &=site &=scode;
%mend findsite;
%findsite(site=Yakutat)

```

The PUT and INPUT functions cannot be used in the macro language, even with the %SYSFUNC function, because they are compilation time functions. Their execution time counterparts (INPUTN, INPUTC, PUTN, and PUTC) can be used with %SYSFUNC. The PUTC function looks up a character (text) value and returns another character value.

**MORE INFORMATION:** A coded table lookup is used to avoid a series of the IF-THEN/ELSE statements in a DATA step in Program 6.3.2b. Program 6.4.2c uses a simple hash to perform a lookup.

---

## 9.5 Making Comparisons to Null Values—Some Considerations

Unlike data set variables, a macro variable can take on a null value; that is, the macro variable can store nothing. This is generally not possible for variables on a data set.

When working with null macro variables the syntax may at first look odd to the DATA step programmer. In Section 5.2.1 an %IF statement, which compares &CITY to a null value (shown here) was introduced.

```
%if &city= %then %do;
```

This is generally considered to be standard syntax for making this type of comparison. Notice that there is nothing between the equal sign (comparison operator) and the %THEN. Since DATA step comparisons must have something on the right of the comparison operator, this statement form often makes newer macro programmers uneasy.

An acceptable alternative form of this same comparison incorporates the use of the %LENGTH function. When a macro variable contains a null value, the %LENGTH will return a zero.

```
%if %length(&city) = 0 %then %do;
```

Although it generally works correctly, using quotes to satisfy the need to have ‘something on the right side of the comparison operator’ is not considered good programming practice. The quotes are parsing characters in the DATA step, but not so in the macro language.

```
%if "&city"="" %then %do;
```

To do something similar without stepping out of the macro world, you could use one of the macro quoting functions to put ‘something to the right of the equal sign.’ Here, the %STR function is used (with nothing between the parentheses).

```
%if &city = %str() %then %do;
```

Although commonly used, Chung and King (2009) showed that each of the previous comparisons can fail under various circumstances. Their tests show that the most robust test of a null value uses a combination of the %SUPERQ and %SYSEVALF functions.

```
%if %sysevalf(%superq(city)=,boolean) %then %do;
```

## 9.6 Evaluating Expressions Stored in a Data Set

In a discussion in the forums on communities.sas.com, (<https://communities.sas.com/thread/48498>), the poster wanted to store an equation in the same data set that was to use the data. Because assignment statement expressions (like arithmetic equations) are evaluated during the DATA step’s compilation phase and the data are not read until the execution phase, it is not straightforward to move the equation from the data set into code.

In the first DATA step in Program 9.6 the equation is stored as a part of the data set in the variable CALC\_EXPRSN. For demonstration purposes this step has been separated from the step that actually integrates and uses the equation.

### Program 9.6: Data Set Containing an Equation

```
data have ;
format calc_exprsn $50. ;
input input1 input2 input3 input4 calc_exprsn $ &;
datalines;
4 6 7 9  INPUT1 * INPUT2 + 100
8 8 6 2  INPUT3 / INPUT2
8 3 90 11 INPUT1 - INPUT3
run;
```

A clever solution, which I would not have thought of, was offered by contributor @Fugue. The equation and the data are converted to macro variables and then reintroduced to the DATA step through the use of the RESOLVE function.

### Program 9.6 (Continued): Executing a Macro Expression during DATA Step Execution

```
data want;
length name $ 8;
set have ;
drop i name;
macro_exprsn = tranwrd(calc_exprsn, 'INPUT' , '&INPUT'); ①
array nums {*} input1-input4; ②
do i = 1 to dim(nums); ③
  call vname(nums{i}, name); ④
  call symputx(name, nums{i}); ⑤
end;
calc_rslt = resolve('%sysevalf('||macro_exprsn||')'); ⑥
calc_rslt2 = compbl(resolve(macro_exprsn)); ⑦
run;
```

- ① The data variable CALC\_EXPRSN contains an arithmetic equation, which uses one or more variables whose names begin with INPUT. We are going to use the macro language to perform the math, so these are changed to macro variables of the same name by adding ampersands using the TRANWRD function.

- ❷ An array is defined for the DATA variables that contain numeric values that might be used in the equation.
  - ❸ An iterative DO loop will be used to step through the array one element at a time.
  - ❹ Through the use of the VNAME routine, the name of the variable in the *i*th position in the array is temporarily stored in the variable NAME, and in ❺ a macro variable is created using this same name.
  - ❻ The SYMPUTX routine is used to create a macro variable with the same name as the variable being referenced in the array, and both the macro variable and the DATA step variable of the same name will hold the same value.
- When the DO loop completes execution, for each of the four variables (INPUT1 – INPUT4) there will be a corresponding macro variable that contains the same value.
- ❼ Several successive operations must take place before an assignment can be made to the variable CALC\_RSLT:
    1. The data variable, MACRO\_EXPRSN, which contains the expression, must be resolved. For the first observation MACRO\_EXPRSN becomes &INPUT1 \* &INPUT2 + 100
    2. The single quotes surrounding the %SYSEVALF prevent it from being seen by the macro scanner. This enables us to execute the %SYSEVALF during the execution of the DATA step. Before the %SYSEVALF can execute, the macro variables must be resolved. The %SYSEVALF becomes: %SYSEVALF(4 \* 6 + 100).
    3. The RESOLVE function enables us to resolve/execute macro language elements during the execution phase of the DATA step. When the RESOLVE function executes it causes the %SYSEVALF to execute, which performs the arithmetic in the equation, and returns the result.
    4. The result (124) is written into the variable CALC\_RSLT.  - ❽ Unlike the expression in ❼, this assignment statement does not contain a %SYSEVALF. As a result, the equation is not executed and CALC\_RSLT2 just receives the character string. This variable is only being included in this example so that you can see the resolved equation. The COMPBL removes multiple blanks to make the equation easier to read.

## 9.7 Using Macro Language Elements on Remote Servers

Using the macro language through a remote SAS session on a remote server is basically the same as on a local system, however there are a couple of additional things that you should be aware of. Primarily, these have to do with the transfer of macro variables between the two SAS sessions.

Two statements are available that specifically make these transfers possible.

### %SYSRPUT

Creates a local macro variable based on a value in the remote session

### %SYSLPUT

Creates a macro variable in the remote server session based on a local session

Before your two sessions can communicate, you must have established a connection between the local and remote session using SAS/CONNECT.

#### **SYNTAX:**

```
%SYSRPUT localname = remotevalue;
%SYSLPUT remotename = localvalue </options>;
```

The %SYSRPUT statement will be executed in the remote session and will create a macro variable in the local environment. While %SYSLPUT is executed in the local environment and will create a macro variable in the remote environment. In Program 9.7a two macro variables local to the %RBUILD macro are being transferred into a remote process through the use of %SYSLPUT. The remote macro variables are

then used inside the remote session. When the remote session is complete a notification is passed back to the local session through the use of the %SYSRPUT statement.

#### Program 9.7a: Using %SYSLPUT and %SYSRPUT to Transfer Values to and from a Remote Session

```
%macro RBuild(dsn=,reg=);

%syslput rindat = &dsn; ①
%syslput rinreg = &reg; ①
rsubmit; ②
  data buildbig;
    set &rindat(where=(region="&rinreg")); ③
    run;
  %nrstr(%sysrput bigbuild = done;); ④
endrsubmit;

%put NOTE: BUILDBIG status is: &bigbuild; ⑤
%mend rbuild;
%rbuild(dsn=sashelp.shoes ,reg=Africa)
```

- ① The two local parameter values are pushed to the remote environment using %SYSLPUT.
- ② The DATA step is submitted to the remote environment for execution.
- ③ In the remote environment, we must use the remote macro variables, not the local ones.
- ④ The %SYSRPUT creates a local macro variable based on values in the remote environment. %NRSTR is used to hide the statement during macro compilation so that the %SYSRPUT statement is treated as text and moved to the remote session. Without %NRSTR, the %SYSRPUT statement would execute on the local host.
- ⑤ The local macro variable &BIGBUILD, which was created in the remote environment, is now available in the local environment.

The %SYSLPUT statement supports a number of options that can substantially increase its utility. These include the ability to transfer multiple macro variables and the use of wildcards to selectively transfer multiple macro variables into the remote environment.

In Program 9.7b the %SYSLPUT uses the \_LOCAL\_ macro variable designator. This syntax became available in SAS 9.4, and it copies all the macro variables from the most local scope into the most local scope in the remote environment. The new remote macro variables will have the same names and values as their counterparts in the local environment.

#### Program 9.7b (Partial): Using \_LOCAL\_ on the %SYSLPUT Statement

```
%macro RBuild(dsn=,reg=);

%syslput _local_;
rsubmit;
  data buildbig;
    set &dsn(where=(region="&reg"));
  . . . portions of the program not shown . . .
```

In addition to the \_LOCAL\_ keyword shown in Program 9.7b, you can use \_ALL\_, \_GLOBAL\_, and \_AUTOMATIC\_ to copy macro variables in their respective scopes. You can also use the LIKE option to copy selected macro variables that match some criteria. In the following %SYSLPUT statement, all local macro variables that start with RC (the asterisk is seen as a wildcard character) will be copied to the remote session.

```
%syslput _local_/like='rc*';
```

**SEE ALSO:** Moriak (2005) describes a number of remote processing tips, including the assignment of macro variables across sessions. Liang and Xie (2016) use remote servers to perform parallel processing.

## 9.8 Working with Macro Variables That Contain Special Characters

It is generally preferable to avoid storing any special characters within a macro variable. However, we do not always have that luxury. When a macro variable contains a special character, such as a comma, a quote, macro trigger, or an unmatched open or closed parenthesis, syntax issues might arise when the macro variable is resolved. In Rosenbloom and Carpenter (2011) a number of different scenarios are discussed along with potential quoting solutions. To simulate the need for the use of the %LEFT function the macro variable &STRING has been given some leading blanks in each of the examples in this section.

**MORE INFORMATION:** The general topic of macro quoting and macro functions that provide quoting were introduced in Section 7.1.

**SEE ALSO:** Rosenbloom and Carpenter (2011) cover many of the items in this section in more detail.

### 9.8.1 Quoting Review

Macro quoting functions are used to mask or hide certain characters during various stages of the macro scanning, parsing, compilation, and execution process. Often these special characters are masked to prevent them from being misinterpreted. The interpretation and use of a given special character might be perfectly appropriate at one point in the process and cause errors at another. Whitlock (1991a) provides an excellent discussion of these issues. Tables 7.1.8a and 7.1.8b describe which special characters are masked by each of the various quoting functions.

For the examples in Section 9.8, assume that you want to use a macro variable within a WHERE statement. Program 9.8.1 shows what this step might look like.

#### Program 9.8.1: Using a Macro Variable within a WHERE Clause

```
%let string = %str(      MEDHIS);
%put where trim(dsn)="%left(&string)";
proc print data=macro3.dbdir;
  where trim(dsn)=:"%left(&string)";
run;
```

The %PUT is used to show how the resultant WHERE clause will appear to the parser.

### 9.8.2 The Problem with Quotes

When the macro variable contains a quote mark, as it does in Program 9.8.2, some quoting functions can be helpful if you are attempting to mask the quote mark during macro execution. However, they will generally not be too helpful if you want to mask the quote mark *after* the macro facility has ‘written’ the code, and it is seen outside of the macro language.

#### Program 9.8.2: Resolved Double Quotes within Double Quotes

```
%let string = %str(      MEDHIS "aka G"); ①
/* The resolved text has multiple double quotes;
%put where trim(dsn)=:"%left(&string)"; ②
/* %BQUOTE will not help;
%put where trim(dsn)=:"%left(%bquote(&string))"; ③
/* Using single quotes on the outside masks the macro triggers;
```

```
%put where trim(dsn)=:'%left(&string)'; ④
/* delay the parsing of the outer single quotes;
%put where trim(dsn)=:%unquote(%bquote(')%left(&string)%bquote(')); ⑤
proc print data=macro3.dbdir;
  where trim(dsn)=:%unquote(%bquote(')%left(&string)%bquote('));
  run;

/* Using the SAS9.4 TSLIT autocall macro function;
%put where trim(dsn)=:%tslit(%left(&string)); ⑥
```

- ① The macro variable &STRING contains both leading blanks as well as double quotes. We will assume that the double quotes cannot be changed to single quotes.
- ② Using &STRING as a quoted constant within a WHERE clause causes syntax problems. The SAS Log shows that there are too many double quotes.

```
where trim(dsn)=:"MEDHIS "aka G""
```

- ③ The %BQUOTE function will mask the resolved double quotes during macro execution, but not when the WHERE is being parsed.
- ④ The use of single quotes on the outside eliminates the double double quotes problem, however they also prevent the resolution and execution of the macro language elements. The SAS Log shows that the unexecuted macro elements remain as a part of the string.

```
where trim(dsn)=:'%left(&string)'
```

- ⑤ Using the single quotes is the right idea; however, we need to mask them until after the %LEFT function has been able to operate on the resolved value of &STRING. We mask the single quote using the %BQUOTE function (it could have been masked using %STR (%')) instead). The %UNQUOTE function causes the whole string to be passed to the macro processor before any of it (this means the leading quote) is seen by the parser. The resolved value is passed back without any macro quoting. The SAS Log shows both the single quotes and the resolved value for &STRING.

```
where trim(dsn)=:'MEDHIS "aka G"'
```

- ⑥ Starting in SAS 9.4 you can use the %TSLIT autocall macro function to enclose resolved macro references with single quotes. Developed for use in PROC DS2, this function is also available outside of DS2. The SAS Log shows:

```
292 %put where trim(dsn)=:%tslit(%left(&string));
where trim(dsn)=:'MEDHIS "aka G'"
```

**SEE ALSO:** A further discussion, including alternate methodologies, of how to deal with quotes within quotes can be found in Carpenter (2014a). More information on the problems associated with embedded special characters can be found in Rosenbloom and Carpenter (2011).

### 9.8.3 Ampersands and Percent Signs

Because ampersands and percent signs are macro language triggers, extra consideration must be taken when the incoming string contains these symbols. This is especially true when they are not intended to be macro triggers. In Program 9.8.3 &STRING contains an ampersand that is not intended to be a macro variable reference.

**Program 9.8.3: The Resolved Value Contains an Ampersand That Should Not Be a Macro Trigger**

```
%let string = %str(      MEDHIS H&J); ①

/* When &STRING is resolved, &J is seen as a macro variable reference;
%put where trim(dsn)=:"%left(&string)"; ②

/* Mask with %SUPERQ;
%put where trim(dsn)=:"%left(%superq(string))"; ③

/* Mask with %SUPERQ and %QLEFT;
%put where trim(dsn)=:"%qleft(%superq(string))"; ④
```

- ① The &J is not intended to be a macro reference. Because the macro variable &J is currently not defined, a warning is issued in the SAS Log.

```
82  %let string = MEDHIS H&J;
WARNING: Apparent symbolic reference J not resolved.
```

Since &J cannot be resolved, &STRING will contain an ampersand.

In this particular example, we could have used the %NRSTR function to mask the & when the macro variable was created. However, that is not a practical solution when the value is coming from other, usually uncontrollable, sources (like data or user-supplied values).

- ② Each time &STRING is resolved or used, another attempt will be made to resolve &J. Clearly we need to mask the ampersand. Since the %LEFT function in turn calls the %VERIFY function, a series of warnings appear in the SAS Log.

```
84  %put where trim(dsn)=:"%left(&string)";
WARNING: Apparent symbolic reference J not resolved.
where trim(dsn)=:"Site H&J"
```

- ③ The %SUPERQ function assumes that the argument is a macro variable that is to be resolved, however the resolved value is immediately masked. Notice that since the %SUPERQ function assumes that its argument is a macro variable, the argument is written without the &. The SAS Log now shows only a single warning.

```
86  %put where trim(dsn)=:"%left(%superq(string))";
WARNING: Apparent symbolic reference J not resolved.
where trim(dsn)=:"Site H&J"
```

This warning is generated when the %PUT is executed, and is a result of the fact that the %LEFT function always returns an unquoted value, regardless of whether the incoming argument was quoted.

- ④ By using the %QLEFT function along with the %SUPERQ function all of the warnings can be eliminated and the meaning will be removed from the &J.

### 9.8.4 Lists and Nested Functions—The Comma Problem

Another common special character problem is often encountered when working with lists, specifically comma-separated lists. Program 9.8.4 demonstrates the problem and suggests a solution.

#### Program 9.8.4: Working with Comma-Separated Lists

```
%let string = %str(      )MEDHIS K, L, and M;

/* When &STRING is resolved, the commas are seen as parameter separators;
%put where trim(dsn)=%left(&string); ①

/* Mask commas with %BQUOTE;
%put where trim(dsn)=%left(%bquote(&string)); ②

/* Mask commas with %SUPERQ;
%put where trim(dsn)=%left(%superq(string)); ③
```

- ① The commas in the resolved value of &STRING cause %LEFT to fail. The SAS Log shows that the commas are interpreted as parameter separators in the call to the %LEFT macro.

```
104 %put where trim(dsn)=%left(&string);
ERROR: More positional parameters found than defined.
where trim(dsn)=""
```

- ② The %BQUOTE function will mask the commas so that the resolved value of &STRING will be correctly seen as a single parameter.

```
107 %put where trim(dsn)=%left(%bquote(&string));
where trim(dsn)='MEDHIS K, L, and M'
```

- ③ Instead of using %BQUOTE the %SUPERQ function could also be used to mask the commas.

Remember that regardless of whether you use %QBQUOTE or %SUPERQ, the %LEFT function will automatically unquote the text that it returns. If you need to have the commas to remain masked be sure to use the %QLEFT function.

**SEE ALSO:** Whitlock (2003a) gives a very nice introduction to macro quoting.

# **Chapter 10: Building and Using Macro Libraries**

<b>10.1 Establishing Macro Libraries .....</b>	<b>254</b>
<b>10.2 Using %INCLUDE as a Macro Library .....</b>	<b>254</b>
<b>10.3 Using Stored Compiled Macro Libraries .....</b>	<b>256</b>
10.3.1 Stored Compiled Macro Library Overview .....	256
10.3.2 Defining and Using a Stored Compiled Macro Library .....	256
10.3.3 Storing and Retrieving the Source Code for Compiled Macros .....	258
10.3.4 Recovering Compiled Macro Source Code .....	260
10.3.5 Using the %SYSMACDELETE Statement.....	260
10.3.6 Changing the SASMSTORE= <i>libref</i> Location.....	260
<b>10.4 Using the Autocall Facility .....</b>	<b>261</b>
10.4.1 Autocall Library Review.....	262
10.4.2 Tracking Autocall Macro Locations .....	262
10.4.3 Options Used with Macro Libraries .....	265
<b>10.5 Macro Library Essentials .....</b>	<b>265</b>
10.5.1 The Macro Library Search Order.....	265
10.5.2 Establishing a Macro Library Structure and Strategy .....	266
10.5.3 Interactive Macro Development .....	266
10.5.4 Modifying the SASAUTOS System Variable .....	267
<b>10.6 Autocall Macros Supplied by SAS .....</b>	<b>268</b>
10.6.1 %VERIFY and %KVERIFY .....	270
10.6.2 %LEFT and %QLEFT .....	270
10.6.3 %CMPRES and %QCMPRES .....	271
10.6.4 %LOWCASE and %QLOWCASE .....	272
10.6.5 %TRIM and %QTRIM .....	273
10.6.6 %DATATYP .....	273
10.6.7 %COMPSTOR .....	274
10.6.8 Autocall Macros That Assist with Color Conversions .....	275
10.6.9 Surfacing Other Autocall Macros Supplied by SAS .....	277

The management of large numbers of macros can be problematic. Names and locations must be remembered, changes must be managed, duplication and variations of individual macros must be monitored, and efficiency issues must be taken into consideration. These problems come even more to the forefront when macros are shared among programmers—especially in a networked environment.

Macro libraries enable you to control and manage the proliferation of your macros. They also enable the macro developer to store, maintain, and manage large numbers of macros. These libraries can be established through the use of:

- %INCLUDE statements: Included macro definitions can mimic a macro library
- Autocall facility: group of programs, which define macros, that can be automatically called by macro name
- Stored compiled macros: collections of macros that were previously compiled and then stored permanently

Macro libraries foster an environment that promotes good macro programming practices by making it easier to avoid the use of duplicate macro definitions. As a result of using these libraries, you will find that it is easier to track and maintain your macros, and macro programming will become even more fun.

For the examples in this chapter and indeed for all of the code examples throughout the book, if you want to execute these sample programs, then be sure to follow the setup instructions. Remember that all of the data sets and programs are available for download, so you do not need to retype either the code or the data. For instructions on accessing and setting up the programs and data, see the “Example Code and Data” section within this edition’s “About This Book” front matter.

**MORE INFORMATION:** Sections 3.3.4 introduces the system options used to establish and control the use of autocall macro libraries.

**SEE ALSO:** First (2001a) discusses various aspects of macro libraries. Additional detail on macro libraries can be found in Carpenter (2001), and a more general discussion on strategies for building automated systems, including the use of macro libraries, can be found in Carpenter and Smith (2001).

Burlew (2014, Chapter 10) discusses various aspects of macro libraries. A good summary of the use of macro libraries can also be found in Chapter 9 of *SAS 9.4 Macro Language Reference, Fourth Edition*.

Muller (2016) specifies a number of considerations when working with macro libraries.

## 10.1 Establishing Macro Libraries

The use of macros is essential to an automated and flexible system. This implies that the control of the macro code is very important to the maintenance of the application. All too often macro definitions become buried within the programs that use them. The result is usually a proliferation of multiple versions of similar macros in multiple programs, each with its own definition of the macro. Parallel code, multiple programs or macros that do essentially the same thing, is an especially difficult problem in large applications that are maintained by multiple programmers. Even when there is only one programmer, there is a tendency to clone a program (“with slight modifications”). Will each version be updated each time a bug is found? Who makes sure that the update happens? Are the various versions of the macro documented? These problems are minimized by placing the definition in a macro library.

Macro libraries are used to avoid the problem of macro cloning by providing a single location for all macro definitions. Rather than cloning the macro, it is adapted (generalized) to fit each of its calling programs and the new generalized macro is stored in one of the libraries. Once in a library, there will only be *one* macro definition to maintain, and it can be used by as many programs as is needed. Obviously this requires documentation as well as diligence. Part of the solution is to place *all* macro definitions in a common library (this could be multiple physical locations), which is accessible to all programs in the application. This way no macro definition will be buried within a program.

There are three types of macro libraries: %INCLUDE, stored compiled macros, and autocall macros. Each has its advantages and disadvantages, and best of all they can be used in conjunction with each other!

It is important to understand the behavior of these libraries, how they are set up, and how they interact with each other.

## 10.2 Using %INCLUDE as a Macro Library

The least sophisticated approach to setting up a macro library [warning: this is an obvious bias on the part of the author] is obtained through the use of %INCLUDE files that store macro definitions. Strictly speaking, the use of %INCLUDE does not actually set up a macro library. The %INCLUDE statement points to a file, and when the statement is executed, the indicated file (be it a full program, macro definition, or a statement fragment) is inserted into the calling program at the location of the call. When

using the %INCLUDE statement to build a macro library, the included file will usually contain one or more macro definitions.

Although you can identify the file to be included directly from within the %INCLUDE statement, it is usually done indirectly through the use of a *fileref*.

```
filename mymacdef  'c:\mymacros\macdef1.sas';
.....
* Bring in the %MACRO to %MEND statements for the MACDEF1 macro;
%include mymacdef;
.....
%macdef1
.....
```

One way to build a library or collection of files that are to be included, is to place them in one central location. This way they will be easier to find and maintain. Under Windows some users of %INCLUDE libraries will indicate that these files are to be included, and that they, therefore, might not be complete programs by using an extension of INC or some other extension other than SAS. While using an extension of INC in no way hampers the file's use by SAS, Microsoft Windows users should be aware that the file will not necessarily be marked by a SAS registered icon.

As was mentioned above, the included file can contain any snippet of SAS code. Within the context of this section, however, we are interested in building macro libraries, and to do this, the included file would contain a macro definition—for example, %MACRO and %MEND statements. There are a couple of efficiency issues that the user of %INCLUDE libraries should remember.

Generally, a given file should not contain more than one macro definition, and the macro being called should not be called from within the included code. The first is important because if the file contains multiple macro definitions, then all the definitions must be loaded and compiled just to use one of the macros. Of course, if all the macros will eventually be used within that SAS session, then this does not really matter. Second, if the macro call is always placed within the included code, the code will need to be re-included (and the macro recompiled) each time that particular macro call is to be used. Remember that a macro only needs to be compiled once during a SAS session.

The greatest disadvantage of using %INCLUDE in a large application is tracking and maintaining the *filerefs* that point to the individual files. While this is less of a problem in static systems, it can still be non-trivial. This issue virtually goes away with the use of the other macro library alternatives.

When you have several macro definitions in a specific location you can use a FILENAME statement that does not fully identify down to the file level. The file reference is then completed when the %INCLUDE is issued:

```
* Identify down to the directory level;
filename maclib  'c:\mymacros';
.....
* Complete the fileref explicitly;
%include maclib(macdef1.sas);
.....
%macdef1
.....
```

When the %INCLUDE inserts code into your program, that code is compiled and then executed. If the inserted code is a macro definition, the %MACRO through %MEND statements will be compiled and the macro will be ready to be called.

## 10.3 Using Stored Compiled Macro Libraries

Macros are always compiled before they are executed, and it is possible to store the compiled code for future use. Compiled macros are stored in a catalog named SASMACR, and by default this catalog is stored in the WORK library. Each compiled macro is stored with the entry type of MACRO and with an entry name corresponding to the name of the macro. When a macro is called, SAS automatically searches this catalog for the compiled version of the macro that was called. Permanently stored compiled macros are also written to a SASMACR catalog, but this catalog is stored in a different (permanent) library instead of the WORK library.

### 10.3.1 Stored Compiled Macro Library Overview

The ability to store and make use of stored compiled macros is by default turned off, and the ability to use them is turned on with the system option MSTORED (the default NOMSTORED turns it off). The library that is to be used to store the permanent SASMACR catalog is specified by using the SASMSTORE= option. Using the stored compiled macro facility enables you to compile the macro one time and then store the compiled version. When called, the macro is already compiled so that you save time and resources, although this is generally not a particularly significant savings.

#### MSTORED

turns on the ability to use the facility. Most sites have this turned off (NOMSTORED) by default.

#### SASMSTORE=

specifies the *libref* (not a *fileref* like in SASAUTOS=) that contains a SAS catalog named SASMACR. This catalog is analogous to WORK.SASMACR, which temporarily stores macros compiled in the current session.

Remember that during macro compilation only macro statements are compiled; simple macro references and non-macro text are not evaluated until macro execution.

During normal macro operations, you may notice that SAS temporarily compiles macros (known as *session compiled macros*) and places them in WORK.SASMACR. This means that the *libref* that you use with SASMSTORE= cannot be WORK.

**CAVEAT:** Although the macro is compiled, the source code is by default not saved and cannot generally be reconstructed from the compiled version of the macro. It is always a good idea to save the source code or use the SOURCE option (see Section 10.3.3).

**MORE INFORMATION:** The option SOURCE, which can be used to save source code when compiling macros, is discussed in Section 10.3.3.

**SEE ALSO:** You can find an easy-to-read and thorough description of stored compiled macros in O'Connor (1992).

Brief introductions to stored compiled macros can be found in Carey and Carey (1996, p. 223), Carpenter and Smith (2001), and Carpenter (2001).

Davis (1997) includes a description and example of the use of stored compiled macros.

### 10.3.2 Defining and Using a Stored Compiled Macro Library

The OPTIONS statement in Program 10.3.2a turns on the use of stored compiled macros (MSTORED) and designates the PROJSTOR *libref* as the location for the catalog PROJSTOR.SASMACR using the SASMSTORE= system option.

**Program 10.3.2a: Defining a Compiled Macro Library**

```
libname projstor 'c:\myprojA\sasmacros';
options mstored sasmstore=projstor;
```

The *libref* must, of course, point to a location that already exists, as the LIBNAME statement will not create a location. Regardless of whether the LIBNAME statement is successful the value of SASMSTORE= will be changed.

You cannot use more than one location reference (*libref*) with the SASMSTORE option. Therefore, if you do want to search multiple catalogs, you need to use either a concatenated *libref* or a concatenated catalog.

Program 10.3.2b creates a concatenated location (MULTI) from the two *librefs* PROJSTOR and ALLMSTOR. Now both locations will be searched when SAS looks for the SASMACR catalog. If the compiled macro is in more than one catalog, the definition found first will be used (read left to right).

**Program 10.3.2b: Using Multiple Compiled Macro Libraries**

```
libname projstor 'c:\myprojA\sasmacros';
libname allmstor 'f:\GroupProjA\saspgrms\macros';
libname multi (projstor, allmstor);

options mstored sasmstore=multi;
```

By default the compiled macro is stored in WORK.SASMACR. You redirect this location at compile time through the use of the /STORE option on the %MACRO statement. For the SASMSTORE definition shown in Program 10.3.2b, the macro AERPT in Program 10.3.2c, will be stored in the PROJSTOR.SASMACR catalog.

**Program 10.3.2c: Storing a Compiled Macro in a Permanent Catalog**

```
%macro aerpt(dsn, stdate, aelist) / store;
```

The use of stored compiled macros is not without problems. The developer must make sure that a macro does not exist in more than one catalog, or if it does exist in multiple catalogs, that the search order of the catalogs is correct. Also, since it is not always possible to reconstruct the code used to create the compiled macro, the developer must make sure that the code is correctly maintained independently from the SASMACR catalog. Fortunately, there are options and statements that are designed to assist with the recovery of the code used to compile macro definitions.

A second potential problem for stored compiled macro libraries is that while compiled macros are all in one location, in the SASMACR catalog, the code itself could be anywhere. Usually, developers that use stored compiled macro libraries will also have a central location to store the source code. A logical location is in the same directory as the SASMACR catalog.

In a multiuser environment users may encounter a third problem relative to the sharing of the macro catalog. The first user to access the macro can lock the catalog, preventing others from using it. To get around this, the SASMACR catalog must be set to READ-ONLY so that multiple users can use the compiled macro at the same time. The ACCESS= option is used on the LIBNAME statement in Program 10.3.2d to create a READ-ONLY catalog.

**Program 10.3.2d: Specifying a READ-ONLY Macro Catalog**

```
libname projmac 'c:\myprojA\sasmacros' access=readonly;
options mstored sasmstore=projmac;
```

### 10.3.3 Storing and Retrieving the Source Code for Compiled Macros

The problem of retrieval of the source code is solved to some extent by using the /SOURCE option that can be used with /STORE on the %MACRO statement. When the SOURCE or SRC option is included, SAS will save the source code as part of the macro entry in the SASMACR catalog. This entry will contain all the code making up the macro definition, including the %MACRO and %MEND statements. To store the source code for %AERPT, the %MACRO statement includes the use of the /SOURCE option shown in Program 10.3.3a.

#### Program 10.3.3a: Using the /SOURCE Option

```
%macro aerpt(dsn, stdate, aelist) / store source;
```

You cannot store the source code when there are nested macro definitions; however, as was discussed in Program 5.1.3a, it is rarely, *if ever*, a good idea to nest macro definitions anyway. If the macro definition for %AERPT contained a nested macro definition, its source code would be stored in the catalog as a part of the definition of %AERPT and you would not see a catalog SOURCE entry for the nested macro.

Once you have source code stored in the SASMACR catalog, the %COPY statement can be used to copy it from the catalog to an external file or *fileref*.

#### SYNTAX:

```
%COPY macro_name </options>;
```

Options for the %COPY statement include:

- LIB=<libref>  
library (other than WORK) that contains the catalog with the SOURCE code.
- OUTfile=<fileref| external\_file>  
output destination of the %COPY statement (defaults to LOG).
- SOURCE or SRC  
specifies that the source code is to be copied.

The source code for %AERPT stored in the SASMACR catalog is copied to an external file using the %COPY statement in Program 10.3.3b.

#### Program 10.3.3b: Retrieving Source Code from a Compiled Macro

```
%copy aerpt / lib=macro3
            out='c:\temp\aeprt.sas'
            src;
```

If the OUT= option is not specified the source code for the macro is written to the SAS Log. Be careful when using the OUT= option. If the specified file already exists %COPY will NOT overwrite it and there will be NO warning, note, or error or other indication that the copy was unsuccessful.

The macro %CHKSRCOPY in Program 10.3.3c performs a check prior to the copy.

**Program 10.3.3c: Check to See If the Resultant File Exists Prior to Generating the Source**

```
%macro chksrccopy(macname=, srcchk=, action=error, age=_);
*****  

/* macname  name of the macro whose source code is to be retrieved  

/* srcchk   path to the file to be checked prior to the copy  

/* action    resultant action if the file exists  

/*          error      copy is terminated with an error  

/*          This is the default  

/*          warning   copy proceeds with a warning  

/*          the previous file is deleted.  

/*          saveold  previous version is saved use AGE character  

/*          age      aging character appended onto the previous file -  

/*          no checking is done  

*****;  

%local xwait;  

%let xwait = %sysfunc(getoption(xwait)); ①  

/* set xwait to noxwait;  

options noxwait; ②

%if %sysfunc(fileexist(&srcchk)) %then %do; ③
  /* The file exists. Take appropriate action;
  %if %upcase(&action) = WARNING %then %do;
    %sysexec del &srcchk; ④
    %put WARNING:|&srcchk| was deleted prior to source retrieval;
  %end;
  %else %if %upcase(&action) = SAVEOLD %then %do;
    %sysexec copy &srcchk &srcchk.&age; ⑤
    %put NOTE:|&srcchk| was aged to |&srcchk.&age| prior to retrieval;
  %end;
  %else %do; ⑥
    %put ERROR: |&srcchk| exists - source generation terminated;
    %return;
  %end;
  /* retrieve source;
  %copy &macname / out="&srcchk" source; ⑦
%end;

/* Restore xwait system option;
options &xwait; ⑧
%mend chksrccopy;
%chksrccopy(macname=aerpt, srcchk=c:\temp\aeprt2.sas)
```

- ① Retrieve the current setting of the XWAIT system option, so that we can return it to its original setting at the end of the macro ③.
- ② Set the system option to NOXWAIT.
- ③ The FILEEXIST function is used to determine whether the file in &SRCCHK already exists. The %DO block is entered if it does exist.
- ④ Issue a warning and delete the existing file prior to copying from the macro catalog.
- ⑤ If the old version is to be saved, copy the existing file to a new version and rename it using the &AGE character(s). Stated this way, the DOS COPY command does not check to see if the new name already is in use. If it is, the file is replaced.
- ⑥ The %COPY is not performed and an ERROR message is generated.
- ⑦ The %COPY takes place.
- ⑧ The XWAIT system option is restored to its original setting, which was captured at ①.

### 10.3.4 Recovering Compiled Macro Source Code

If you do need to surface the source code in a compiled stored macro for which the /SOURCE option was not used, you might be able to apply a technique attributed to Ian Whitlock, which uses the %PUT and a macro quoting function, as shown in Program 10.3.4a. Some, but not all, of the inner workings of the macro will be written to the SAS Log. This is not a technique that is supported by SAS, and it will not surface macro language statements.

#### Program 10.3.4a: Recovering Compiled Macro Code

```
%put %quote (%doboth() );
```

When you are concerned about preventing others from reengineering your macro or to make sure that the inner workings are not surfaced, you can use the /SECURE option to prevent the %COPY statement or other techniques like MPRINT from revealing your code. In Program 10.3.4b the macro statement is adapted to include the use of the SECURE option. The STORE option is not required in order to use the SECURE option.

#### Program 10.3.4b: Using the SECURE Option

```
%macro dobboth(indata=,vlist=,cnt=10) / secure;
```

If you use the SECURE option, be sure to store your code separately because it will be impossible to reconstruct it from the compiled macro even when the STORE option has been applied.

**SEE ALSO:** Sun and Carpenter (2011) describe a number of security and control techniques that can be used with stored compiled macro libraries.

### 10.3.5 Using the %SYSMACDELETE Statement

The %SYSMACDELETE statement can be used to delete a macro definition from the WORK.SASMACR catalog. This statement accepts the name of a single session compiled macro, so a list of statements is needed if more than one macro is to be removed from the WORK catalog. In Program 10.3.5 the macro definitions for %DOBOTH and %AERPT are to be removed.

#### Program 10.3.5: Using %SYSMACDELETE to Remove Compiled Macros from WORK.SASMACR

```
%sysmacdelete dobboth;
%sysmacdelete aerpt;
```

A warning is issued if an attempt is made to delete a macro that does not exist in the WORK.SASMACR catalog.

Under Windows you can use the SAS Explorer to delete macro entries from the WORK.SASMACR catalog. Although this technique seems to work, it is *not* supported by SAS, and is *not* recommended as there may be memory elements of the macro that are not cleared. Play it safe; use %SYSMACDELETE.

**MORE INFORMATION:** Initial discussion of %SYSMACDELETE can be found in Section 8.2.5.

### 10.3.6 Changing the SASMSTORE= *libref* Location

Generally, once a *libref* has been assigned to the SASMSTORE= system option it will not need to be changed. However, if you need to change the location referred to by that *libref*, you may find that the *libref* is locked by the SASMACR catalog, thus preventing you from using that *libref* with a new location. The %SYSMSTORECLEAR statement can be used to close the SASMACR catalog, which releases the lock, and enables you to assign a new location to the SASMSTORE= *libref*.

In Program 10.3.6 the *libref* MACRO3 is used with the SASMSTORE= option. Once a macro is read into or out of the MACRO3.SASMACR catalog, the catalog is opened, and until it is closed the location for the MACRO3 *libref* cannot be reassigned.

#### Program 10.3.6: Using the %SYSMSTORECLEAR Statement

```

libname macro3 "&path\data"; ❶
options mstored sasmstore=macro3;

%macro doboth(indata=,vlist=,cnt=10) /store; ❷
  %if &vlist ne %then %sortit(dset=&indata, bylist=&vlist);
  %else %put SORTIT was not called;
  %look(dsn=&indata,obs=&cnt)
%mend doboth;

%sysmstoreclear; ❸

libname macro3 'c:\temp'; ❹
%macro aerpt(dsn, stdate, aelist) / store; ❺
  %put testing AERPT;
%mend aerpt;

```

- ❶ The MACRO3 *libref* is established and used with the SASMSTORE= system option to establish a stored compiled macro library.
- ❷ The MACRO3.SASMACR catalog is opened when either a macro is read from it or, as in this case, is written to it. Unless it is otherwise closed, once the catalog is opened, it remains open for the duration of that job or session. The open catalog prevents the use of the MACRO3 *libref* in another LIBNAME statement ❹.
- ❸ The %SYSMSTORECLEAR statement closes the SASMACR catalog so that the catalog itself will not prevent the *libref* from being reassigned.
- ❹ The location associated with the MACRO3 *libref* can now be changed. Although it is a new location, the SASMSTORE= option does not need to be changed as it is still pointing to the MACRO3 *libref*.
- ❺ The compiled version of the macro %AERTP will be stored in the SASMACR catalog in the new location.

## 10.4 Using the Autocall Facility

When a requested macro is not found in a SASMACR catalog, the autocall library is searched. Much like an %INCLUDE file, this library contains the macro definitions in the form of SAS code. When a macro is called, SAS searches for a file in the specified autocall location with the *same* name as the name of the macro. The code in the corresponding file is then submitted for processing. Since this file contains the macro definition (the %MACRO to %MEND statements), the macro is then compiled and made available for execution. Under Windows this library location is a folder and each macro definition is an individual file. Under MVS a partitioned data set (PDS) is used with each member corresponding to a macro.

By default the ability to make use of the autocall facility is turned on (MAUTOSOURCE). However, since the ability to make use of the autocall facility is essential for virtually every program that I write, I always explicitly turn it on, just to be sure, regardless of whether or not I think that it should already be on.

**MORE INFORMATION:** Autocall libraries were first introduced in Section 3.3.4. Although Section 10.4.1 is a brief review, most of the remainder of the material in Section 10.4 assumes that you understand the material in Section 3.3.4.

**SEE ALSO:** Agbenyegah (2015) highlights autocall macros supplied by SAS that are used with market research. Wang (2015) uses several annotate autocall macros.

### 10.4.1 Autocall Library Review

As you create autocall libraries be sure that you use *filerefs*, *not librefs*, to specify the locations of the autocall programs. You can also replace the *fileref* in the SASAUTOS= option with a direct location reference; however, this approach is less flexible and is not as “clean.”

Notice that the SASAUTOS= option in Program 10.4.1a contains a reference to the SASAUTOS *fileref*. This automatic *fileref* is defined by SAS during the SAS initialization process. It points to the location of a series of macros supplied by SAS Institute (Section 10.6 discusses these autocall macros supplied by SAS in more detail). When defining an autocall library be sure to include the SASAUTOS *fileref* or you will lose the ability to call these autocall macros supplied by SAS.

#### Program 10.4.1a: Establishing an Autocall Library with the Default Settings

```
options mautosource sasautos=sasautos;
```

Usually, your user-defined macro libraries will be stored in a variety of locations. Program 10.4.1b makes sure that the autocall facility is available (MAUTOSOURCE), and it specifies the *filerefs* of the locations (SASAUTOS=) that contain the SAS programs with the macro definitions of interest.

#### Program 10.4.1b: Creating a Multilevel Autocall Library

```
filename grp5mac 'z:\group5\macros';
filename prj5Amac 'g:\group5\prjA\macros';
options mautosource sasautos=(prj5amac, grp5mac, sasautos);
```

Very often the library location will depend on the usage of the macro. In the scenario depicted in Program 10.4.1b, project-level macros are stored in the first library to be searched (PRJ5AMAC), while the more general macros used by the entire group are stored in GRP5MAC. Very often for my work I will also first list a development level library on a drive (which only the development team has access to). When a macro is ready to go into production it merely needs to be moved from the development library to one of the production libraries, where it can be accessed by all of the appropriate users.

Notice that the SASAUTOS *fileref* is listed last. This enables the user to create macros that will override the default definitions of the macros supplied with SAS. Some macro users feel that this is a bad practice, and advocate that SASAUTOS be listed first. They argue that with SASAUTOS listed first, if someone ‘accidentally’ creates a macro with the same name as one of the SAS autocall macros, then the SAS version of the autocall macro will be used.

### 10.4.2 Tracking Autocall Macro Locations

When you have a series of autocall libraries it is sometimes difficult to determine from which location a given macro was drawn. This can be especially true if different versions of a macro reside in different libraries. There are three system options that can be used with autocall libraries that can assist you with determining where a given macro definition resides:

#### **MAUTOCOMPLOC.**

gives the macro location at the time of macro compilation

#### **MAUTOLOCDISPLAY**

displays the macro library location when a macro is called

#### **MAUTOLOCINDEXES**

adds the full location to the SASMACR catalog at macro compilation

#### **MAUTOCOMPLOC**

Typically, when an autocall macro is first called it is compiled and the compiled macro is stored in the catalog WORK.SASMACR. When turned on, the MAUTOCOMPLOC system writes the physical location of the macro code to the SAS Log when it is first compiled. The SAS Log for Program 10.4.2a shows that the %LEFT autocall macro has been invoked for the first time in a SAS session.

**Program 10.4.2a (SAS Log): Using the MAUTOCOMPLOC System Option**

```

9   option mautocomploc;
10  %put |%left(    leading blanks)|;
MAUTOCOMPLOC: The autocall macro LEFT is compiling using the autocall
source file C:\Program
      Files\SASHome2\SASFoundation\9.4\core\sasmacro\left.sas. ①
MAUTOCOMPLOC: The autocall macro VERIFY is compiling using the autocall
source file C:\Program
      Files\SASHome2\SASFoundation\9.4\core\sasmacro\verify.sas. ②
|leading blanks|
11
12  %put %left( another left);
another left ③

```

- ① The %LEFT macro is an autocall macro supplied by SAS. When it is first compiled in a session its physical location is surfaced in the SAS Log. This location is included in the autocall search by including the SASAUTOS fileref in the SASAUTOS= system option (see Programs 10.4.1a and 10.4.1b).
- ② The %LEFT macro calls a second autocall macro, %VERIFY. This macro's location is also revealed.
- ③ A second invocation of %LEFT does not require the macro to be recompiled so that the location message is not regenerated by MAUTOCOMPLOC.

**MAUTOLOCDISPLAY**

While the MAUTOCOMPLOC only writes the source location once, the MAUTOLOCDISPLAY writes this information each time the macro is called.

**Program 10.4.2b (SAS Log): Using the MAUTOLOCDISPLAY System Option**

```

16  option mautolocdisplay;
17  %put |%left(    leading blanks)|;
MAUTOLOCDISPLAY(LEFT): This macro was compiled from the autocall file
C:\Program ④
      Files\SASHome2\SASFoundation\9.4\core\sasmacro\left.sas
MAUTOLOCDISPLAY(VERIFY): This macro was compiled from the autocall file
C:\Program
      Files\SASHome2\SASFoundation\9.4\core\sasmacro\verify.sas
|leading blanks|

```

- ④ Although %LEFT has already been called in this session (Program 10.4.2a was executed first in this case), MAUTOLOCDISPLAY causes the full location of the macro definition to be written to the SAS Log.

Because MAUTOLOCDISPLAY writes the location of a macro each time it is called, it can be very useful when debugging a series of nested autocall macros.

**MAUTOLOCINDEXES**

Although both MAUTOCOMPLOC and MAUTOLOCDISPLAY will show you the physical location in the SAS Log, this is not very useful when you need to harvest this path, perhaps when building documentation. The MAUTOLOCINDEXES option writes the path information into the SASMACR catalog where it can be harvested automatically.

By default the description information for a macro entry in the SASMACR catalog is blank. When MAUTOLOCINDEXES is turned on and an autocall macro is compiled, the physical path information is added to the description field for the catalog entry associated with that macro. Program 10.4.2c prints the catalog entries for those autocall macros that have been compiled in the current SAS session.

**Program 10.4.2c: Using the MAUTOLOCINDEXES System Option**

```
options mautolocindexes; ⑤
```

```
%put %lowcase(ALL UPPER);
title 'Program 10.4.2c Using MAUTOLOCINDES';
proc print data=sashelp.vcatalog
  (where=(libname='WORK' & memname='SASMACR')) ;
  var libname memname objname objdesc;
run;
```

- ⑤ The MAUTOLOCINDES system option is specified and the %LOWCASE autocall macro is called. Assuming that the %LEFT macro was called earlier, before the MAUTOLOCINDES option was in effect (see Program 10.4.2b), the path for the autocall macro %LOWCASE can be surfaced in the catalog entry description (OBJDESC) using PROC PRINT.

**Figure 10.4.2: Using the MAUTOLOCINDES System Option**

Program 10.4.2c Using MAUTOLOCINDES				
Obs	libname	memname	objname	objdesc
29748	WORK	SASMACR	LEFT	
29749	WORK	SASMACR	LOWCASE	⑥ 'Program Files\SASHome2\SASFoundation\9.4\core\sasmacro\lowcase.sas'
29750	WORK	SASMACR	VERIFY	

- ⑥ Only the path information for %LOWCASE is included in OBJDESC as it is the only macro to be compiled subsequent to the issue of the MAUTOLOCINDES system option.

It is common to want to retrieve the path description automatically. Program 10.4.2d uses the %GETAUTOPATH macro to retrieve the path of an autocall macro.

#### Program 10.4.2d: Retrieving the Path Location of an Autocall Macro

```
%macro getautopath(macname=left);
proc sql noprint;
select objdesc ⑧
  into :macpath ⑨
  from dictionary.catalogs
  where libname='WORK' &
    memname='SASMACR' &
    objname="%upcase(&macname)";
quit;
%mend getautopath;

%let macpath = ; ⑦
%put |%qtrim(string )|;
%getautopath(macname=qtrim)
%put qtrim &macpath; ⑩
```

- ⑦ The macro variable &MACPATH is added to the most current local symbol table, and %QTRIM is used. This loads the path (this assumes that MAUTOLOCINDES has been turned on) into the entry in the SASMACR catalog.
- ⑧ The description for this entry (OBJDESC) is read from DICTIONARY.CATALOGS.
- ⑨ The description, which contains the path information, is written to the &MACPATH macro variable. Because it already exists in a higher table ⑦ the macro variable is not written to the local table.
- ⑩ &MACPATH now contains the path information.

**MORE INFORMATION:** Autocall macros were introduced in Section 3.3.4.

**SEE ALSO:** A short discussion of the use of the autocall facility can be found in Jensen and Greathouse (2000).

---

### 10.4.3 Options Used with Macro Libraries

There are several system options that can be used with autocall libraries that can on occasion be helpful. Three of these (MAUTOSOURCE, SASAUTOS=, and MRECALL) were introduced in Section 3.3.4.

#### MPRINTNEST and MLOGICNEST

When you are using nested macros (macros that call macros), MPRINT and MLOGIC only show the innermost level, and it is difficult to discern the hierarchy of calling macros. MPRINTNEST and MLOGICNEST overcome this limitation by listing each of the calling macros in order. MPRINTNEST and MLOGICNEST, respectively, require the use of the MPRINT and MLOGIC system options, which must also be turned on.

#### MCOMPILENOTE

This option can be used to instruct the macro facility to write a note to the SAS Log each time you compile a macro. There are three levels for this option.

**SYNTAX:**

```
OPTIONS MCOMPILENOTE=<NONE | NOAUTOCALL | ALL>;
```

**NONE**

prevents any compilation notes from going to the SAS Log (this is the default).

**NOAUTOCALL**

prevents notes associated with the compilation of autocall macros.

**ALL**

writes a note to the SAS Log for each compiled macro.

Macros do not have to compile successfully in order to generate a NOTE. In fact, this is one of the advantages of this option because macros that result in compile errors will have a specific note written to the SAS Log.

---

## 10.5 Macro Library Essentials

Although establishing and using macro libraries is fairly straightforward, there are a number of issues that arise that tend to make the process more difficult. Some of the issues that you might want to know more about before creating or using macro libraries include the following:

- how SAS searches for a macro definition
- how to establish a macro library structure
- how to develop macros in an interactive environment
- how to modify the SASAUTOS system variable

---

### 10.5.1 The Macro Library Search Order

As these libraries of macros are established it is important for both the developer and the user to understand the relationship between them. One of the more important aspects of this relationship is the order of locations that SAS searches for a macro definition.

When a macro is called, the macro facility must first find the macro definition before it can be compiled and executed. Since the macro definition can be stored in any of several locations, the developer needs to be able to control the search. The search for the macro definition uses the following order:

1. WORK.SASMACR
2. Stored compiled macros (when they are turned on)
3. Autocall macros

The first time that a macro is called it will not be found in any of the catalogs of compiled macros, but once it is called, it will be compiled and the compiled code will be stored (in WORK.SASMACR unless otherwise specified). On successive calls to that macro, its compiled definition will be found and the search will not extend to the autocall library a second time. This process minimizes the compilation of the macro.

This is a major advantage over the use of %INCLUDE as a macro library. When a macro definition is brought into the program through %INCLUDE, it will be compiled for each %INCLUDE. Of course, if the programmer is careful, this is not such a bad thing. If the %INCLUDE appears only once, the macro will be compiled at that time and added to the WORK.SASMACR catalog, and the compiled macro definition will be used from then on (as long as it is not included another time).

Macros in the autocall libraries are never inspected by the SAS Macro Facility until the macro is called. The macro is then compiled (and its compiled version is loaded into WORK.SASMACR) and executed.

### **10.5.2 Establishing a Macro Library Structure and Strategy**

Placing your macros in a library will help to organize your programs. However, in larger groups, projects, or in organizations with multiple programmers sharing macros, it becomes necessary to organize the libraries themselves. In the designations of the locations of both the autocall library (Program 10.4.1b) and the stored compiled macros (Program 10.3.2b), there are two locations. By specifying multiple locations for the libraries it becomes possible to organize them into collections of macros. Typically, for the work that I do, I like to arrange the collections according to how the macros are to be used. Macros that are specific to a task or project will be placed in the location that will be searched first. Then the macros that are more generalized or are useful to multiple projects are placed in a location that will be available to users of all projects. This arrangement enables the developer to create general tools that are available to everyone as well as specific macros that apply to only one project or task.

The question then is not *if* you should set up a library, but which macro library setup to use.

The %INCLUDE library and the autocall library can be set up in a very similar fashion. The primary advantage of the autocall library is that the developer does not need to manage or work with the individual *fileref*s, as these are controlled automatically through the SASAUTOS= system option.

Unless a macro is unusually large or complex, it generally takes very little time to locate and compile a macro stored in the autocall library. This suggests that there is not a substantial time savings in using the stored compiled macro library. Since both libraries require the developer to store and maintain the source code, there does not seem to be a compelling reason to adopt one library type over the other. For this reason and because almost all SAS sites already use the autocall facility (if for no other reason than to get access to the autocall macros supplied by SAS—see Section 10.6), the autocall library seems to be used much more frequently than the stored compiled macros library.

One strategy that has worked well in limited situations combines these two types of libraries. This combined library approach specifies an autocall library (SASAUTOS=). It also turns on the stored compiled macro library (MSTORED) and then points the location (SASMSTORE=) to the **same** location as the autocall library. Since the SASAUTOS=*fileref* and the SASMSTORE=*libref* both point to the same directory, the SASMACR catalog will reside in the same location as the source code that defines the macro. Once this is done all the macros in the autocall directory will **also** have the /STORE option. Now we have the best of both worlds: one source program and a compiled macro, both stored in one easy-to-find location.

The autocall macro %COMPSTOR supplied by SAS (see Section 10.6.7) can be used to permanently store a compiled version of selected autocall macros supplied by SAS.

---

### **10.5.3 Interactive Macro Development**

When developing macros in the interactive environment, special care must be taken to make sure that the correct macro definitions are compiled and stored. If the developer is not careful it is possible to call a macro without using the latest, most up-to-date version of the macro definition. This is not a good thing.

As was discussed earlier, after a macro definition is submitted, the compiled code is stored in the appropriate SASMACR catalog. This catalog will be in the WORK library unless stored compiled macros are being used and the /STORE option is present on the %MACRO statement. When the macro is called, SAS looks for the macro definition in one or more of these catalogs, and only if the definition is **not** found is the autocall library searched.

It is important to remember the order in which macros are stored, compiled, and re-executed. If you use the display manager during the debugging process, a change to a macro definition stored in a program in the autocall library (SASAUTOS=) will not change the compiled version in WORK.SASMACR. Re-execution of the macro will not implement changes unless the compiled version of the macro has been eliminated from the WORK.SASMACR catalog (see %SYSMACDELETE in Section 8.2.5) or you recompile the macro so that the WORK.SASMACR catalog is updated using the version in the autocall library.

Suppose the developer is debugging a macro whose definition resides in the autocall library. She calls the macro for execution, but is not satisfied with the results. If she edits the program, saves it back into the autocall library, and then re-executes it, the results will be the same! Her changes will be ignored! Her changes are not implemented because the macro has not been recompiled! She has updated the code, but because the macro has already been compiled, and because it already resides in the SASMACR catalog, SAS will never look for the new definition in the autocall library. After making the change in the macro definition, she needs to do one of the following:

- Submit the macro definition (%MACRO to %MEND). This places the latest definition in the SASMACR catalog. This can be done directly through the display manager or by naming the macro definition in a %INCLUDE statement.
- Delete the compiled macro entry from the appropriate SASMACR catalog. The next execution of the macro will cause SAS to seek out the autocall definition, which will then be compiled and re-stored in the SASMACR catalog. The %SYSMACDELETE statement (see Section 10.3.5) can be used if you need to delete the entry from WORK.SASMACR.
- Exit and reenter SAS.

Each of these solutions solves the same problem. When your SAS environment includes macro libraries, you must remember that as you edit your macro definitions, you must also update the appropriate SASMACR catalogs as well. It is not enough to just change the macro definition.

The above example illustrates a situation that is usually not an issue when you use a %INCLUDE library to hold macro definitions. When the %INCLUDE statement brings in a macro definition, the definition itself is usually inserted directly into the code. It is then submitted and compiled—all in the same operation. Although this might initially sound like an advantage, it is not because the macro must be recompiled each time that the %INCLUDE is executed. This method does not take full advantage of the ability to save the compiled macro in the SASMACR catalog.

**CAVEAT:** It has been this author's experience that manually deleting the compiled version of the macro from the WORK.SASMACR catalog (the second method in the list above) successfully forces the recompilation of the macro. However, this technique is *not* one that is either recommended or supported by SAS because this approach and its various ramifications have not been fully tested and could, under certain unforeseen circumstances, cause problems. Instead, the %SYSMACDELETE statement should be used.

#### 10.5.4 Modifying the SASAUTOS System Variable

The default value of SASAUTOS is established as a system variable in the configuration file (the name and location of the configuration file varies by operating system and version. Since this file is automatically executed when SAS is initiated, it becomes a useful way to establish customized library locations without using system options.

To modify the SASAUTOS path simply edit your configuration file. Look for the -SET SASAUTOS statement and add one or more paths to the list. In Program 10.5.4, the directory "f:\GroupProjA\saspgms\macros" has been added as the first directory in the concatenated set of autocall directories.

#### **Program 10.5.4: Modifying the SASAUTOS Environmental Variable**

```
/* Setup the SAS System autocall library definition */
-SET SASAUTOS (
    "f:\GroupProjA\saspgms\macros"
    "!SASROOT\core\sasmacro"
    "!SASROOT\aacomp\sasmacro"
    "!SASROOT\accelmva\sasmacro"
    "!SASROOT\assist\sasmacro"
    "!SASROOT\dmscore\sasmacro"
    "!SASROOT\dquality\sasmacro"
    "!SASROOT\ets\sasmacro"
    "!SASROOT\graph\sasmacro"
    "!SASROOT\hps\sasmacro"
    "!SASROOT\inttech\sasmacro"
    "!SASROOT\or\sasmacro"
    "!SASROOT\qc\sasmacro"
    "!SASROOT\stat\sasmacro"
)
```

Once edited, the configuration file can be saved in another location. The appropriate configuration file is selected at SAS start-up with the -CONFIG initialization option.

The SASAUTOS environmental variable is not available under all operating systems, and its specification varies. Consult the SAS Companion appropriate for your version of SAS and your operating system.

## **10.6 Autocall Macros Supplied by SAS**

In addition to any macros that you might write and add to your own autocall library, SAS comes supplied with a large number of macros in its own autocall library. These can be found in your default SASAUTOS location; under Windows this might be !sasroot\core\sasmacro. Depending on your operating environment and what SAS products are licensed, additional autocall macro locations may be available. Consult the SAS Companion for your operating environment for more information on the location of the autocall libraries. The SASAUTOS environmental variable (see Section 10.5.4) contains a list of the locations of autocall macros supplied by SAS. Although there are literally hundreds of autocall macros in these libraries, most are for internal use by SAS, and only a few are actually designed to be used by SAS users.

Not only are these macros useful in and of themselves, but because the code is available for your inspection. You can modify these macros for your own purposes or use them as patterns for new macros. One of the really big advantages of looking at these programs is that you will get to see some macro code that was written by the developers of portions of SAS. Very often this macro code includes interesting techniques and can be instructive for someone who learns by example.

Several autocall macros that are supplied by SAS behave as macro functions, and looking at their code provides insight into writing your own macro functions (see Section 7.5). Some of these macros are noted briefly here and several are discussed in detail throughout this book.

**Table 10.6: Selected SAS Autocall Macros**

Autocall Macro	Described in Section	Function Description
%CMPRES and %QCMPRES	10.6.3	Compresses a text string by removing multiple (repeated) blanks as well as leading and trailing blanks.
%COMPSTOR	10.6.7	Permanently stores a compiled version of selected autocall macros supplied by SAS.
%DATATYP	10.6.6	Determines whether the argument is numeric.
%LEFT and %QLEFT	7.2.6 10.6.2	Left-justifies the text argument by removing leading blanks.
%LOWCASE and %QLOWCASE	7.2.7 10.6.4	Converts the text string to lowercase (this is the functional opposite of the %UPCASE macro function).
%TRIM and %QTRIM	7.2.8 10.6.5	Trims trailing blanks from text.
%VERIFY %KVERIFY	10.6.1	Locates the first character not in a string—the opposite of the %INDEX macro function.

For the most part, these macros closely mimic DATA step functions in name, syntax, and result. None are difficult to use. Indeed, the hardest part about using them is to know of their existence.

Many of these macros also have a “Q” version (for example, %QLEFT and %QTRIM) for use when the returned value contains symbols or characters whose meaning should remain hidden. See Section 7.1 for more on macro quoting and when it may be needed.

Because these are macros (although they sometimes act like functions), the text string or argument cannot contain embedded commas. When the argument does contain a comma, the macro parser interprets the comma as a parameter delimiter and syntax errors can result. Often when this happens, you can quote the argument so as to hide the embedded comma.

To make matters a bit more complicated, some of these macros (such as %LOWCASE) also call macro functions that use commas to delimit arguments. For these macros, commas in the argument can also cause problems in the interior macro or macro function. Quoting may be of less utility in these cases, although using the Q version of the macro will usually help.

The following sections show how to use some of these macros and in most instances show a portion of the actual macro definition. When you want to examine the code used to create these autocall macros that are supplied by SAS, you can search for the %MACRO to %MEND statements contained in your SAS installation or you can use the DATA step such as the one shown in Program 10.6.

#### Program 10.6: Surfacing the Program Statements in an Autocall Macro

```
data _null_;
  infile sasautos('ds2csv.sas'); ①
  input; ②
  put _infile_; ③
run;
```

- ① The sasautos fileref is completed with the filename. Because autocall macros and their associated file share the same name, simply supply the name of the macro that you are interested in.
- ② Each line of the program is read into the input buffer.
- ③ The contents of the input buffer are written to the SAS Log. A FILE statement could be used to reroute the code directly to a file.

**SEE ALSO:** *SAS 9.4 Macro Language Reference, Fourth Edition*, describes these macros in Chapter 13, “Autocall Macros.”

### 10.6.1 %VERIFY and %KVERIFY

Functionally the opposite of the INDEXC function, this macro enables you to search a text string for the first character that is not in the target list. %VERIFY then returns the position of that character. %KVERIFY does the same thing, but is designed to support National Language character sets.

#### SYNTAX:

```
%VERIFY (sourcecharacters, exclusioncharacters)
%KVERIFY (sourcecharacters, exclusioncharacters)
```

#### VALUE RETURNED:

The position of the first character in the source list that is not in the exclusion list.

In Program 10.6.1 the macro variable &YRCODE contains a string that starts with some unknown number of digits. We want to strip off the leading numbers and create a new macro variable (&CODE) that starts with the first character that is not a number.

#### Program 10.6.1 (SAS Log): Using the %VERIFY Autocall Function

```
30  %let yrancode = 2016CAdwz25;
31  %let code = %substr(&yrancode,%verify(&yrancode,1234567890));
32
33  %put &=yrancode &=code;
YRCODE=2016CAdwz25 CODE=CAdwz25
```

The first argument to %VERIFY is the string that is to be searched, and the second contains a list of one or more characters that we want to avoid (in this case all the numbers). Because the letter C, which is in the fifth position, is the first non-digit, the %VERIFY function will return the number 5. The %SUBSTR function uses this number to determine the starting position of the subset of characters that is to be retrieved.

**SEE ALSO:** The %VERIFY macro is used by Hirabayashi (2001) in a macro that generalizes the output from PROC FREQ.

### 10.6.2 %LEFT and %QLEFT

The syntax and operation of the %LEFT and %QLEFT functions is described in Section 7.2.6; however, it is interesting to actually take a look at the code associated with these macro functions. The entire %LEFT macro, *sans* internal documentation, is shown here:

```
%macro left(text);
%local i;
%if %length(&text)=0 %then %let text=%str( );
%let i=%verify(&text,%str( )); ①
%if &i ② %then %substr(&text,&i); ③
%mend;
```

- ① The %VERIFY macro (see Section 10.6.1) is used to return the position of the first non-blank character in &TEXT.
- ② Assuming that there is a leading blank (&i>0), this position is then used by the %SUBSTR function (%QSUBSTR is used by %QLEFT) to subset the incoming text starting with the first non-blank character.

- ❸ The %SUBSTR function resolves to the amended text, and this value is passed back out of the macro.

Because this is a macro function, the macro call is replaced with the left-justified string.

### 10.6.3 %CMPRES and %QCMPRES

The %CMPRES macro mimics the COMPBL DATA step function by removing multiple blanks from the text string. In effect, it also calls %LEFT and %TRIM because it also removes all leading and trailing blanks.

#### SYNTAX:

```
%CMPRES (argument)
%QCMPRES (argument)
```

#### VALUE RETURNED:

The value of the *argument* without leading or trailing blanks and without any repeated internal blanks is returned.

In the example in Section 6.2.1, SQL is used to create a macro variable containing a numeric value. As part of the process, the numeric value is converted to character prior to being assigned to the macro variable, and as a part of the conversion process the result is right-justified (there are a number of leading blanks). Program 10.6.3 creates the macro variable directly from the numeric variable and then uses %CMPRES to remove the leading blanks. In this example, the vertical bars ( | ) are used to help show the position of the blanks.

#### Program 10.6.3: Using the %CMPRES Autocall Function

```
%let text = %str( The quick brown fox jumped over the lazy dog
);
%put text= |&text|;
%put text= |%cmpres(&text)|;
```

The SAS Log shows that multiple internal blanks, as well as leading and trailing blanks, have been removed:

```
62  %put text= |&text|;
text= | The quick brown fox jumped over the lazy dog |
63  %put text= |%cmpres(&text)|;
text= |The quick brown fox jumped over the lazy dog|
```

The code for the %CMPRES macro shows how logic and a %DO %WHILE loop is used to step through the incoming text string:

```
%macro cmpres(text);
%local i;
%let i=%index(&text,%str( )); ❶
%do %while(&i ne 0);
  %let text=%qsubstr(&text,1,&i)%qleft(%qsubstr(&text,&i+1)); ❷
  %let i=%index(&text,%str( )); ❶
%end;
%left(%qtrim(&text)) ❸
%mend;
```

- ❶ The %INDEX function is used to find any occurrences of a double blank %STR( ). If found, the starting position is noted in &I, which is used in the %DO %WHILE loop.
- ❷ The %LET is used to rewrite the incoming text string &TEXT. Notice that two macro functions are used in the same statement. The %QSUBSTR function returns the first &I characters in the string (this is where the first of the two blanks is retained). The %QLEFT autocall macro then left-justifies the remainder of the string. Because the function and macro are resolved and executed before the %LET,

the two resulting text strings are effectively concatenated into a new string, which is placed back into &TEXT. Using %QLEFT eliminates the need to step iteratively through a long string of blanks.

- ③ The %LEFT macro will be called last and will resolve to a text string. Effectively, this is the text that is passed back from the call to %CMPRES. Actually, nothing is really passed back. Rather, this macro resolves to a potentially modified version of &TEXT, and that string is what is seen as the resolved text in the code that called %CMPRES. Because this is a macro call, a semicolon is not wanted and could cause errors when the macro is called. %QCMPRES uses a %QLEFT at this point to return quoted text.

You could easily modify this macro to remove any combination of characters, not just double blanks.

#### 10.6.4 %LOWCASE and %QLOWCASE

The %LOWCASE autocall macro function, which was introduced in Section 7.2.7, translates text to all lowercase.

The logic used in this macro has changed in recent versions of SAS. The current version takes advantage of the LOWCASE DATA step function, which had not been available in earlier versions of SAS.

```
%macro lowercase(string);
%sysfunc(lowercase(%nrbcquote(&string)))
%mend;
```

Although neither as efficient nor as straightforward, the code for earlier versions of this macro is interesting because of the way it finds uppercase characters and then translates them to lowercase.

```
%macro lowercase(string);
%local i length c index result;
%let length = %length(&string);
%do i = 1 %to &length;
    %let c = %substr(&string,&i,1);❶
    %if &c eq %then %let c = %str( );
    %else %do;
        %let index = %index(ABCDEFGHIJKLMNPQRSTUVWXYZ,&c); ❷
        %if &index gt 0 %then ❸
            %let c = %substr(abcdefghijklmnoprstuvwxyz,&index,1); ❹
        %end;
        %let result = &result.&c; ❺
    %end;
    &result ❻
%mend;
```

For discussion purposes, assume that %LOWCASE is called with the following:

```
%LOWCASE (SAS Macro Language)
```

- ❶ The  $i^{\text{th}}$  character in the string of interest (&STRING) is temporarily stored in the macro variable &C. For this example &C will contain M when &I=5.
- ❷ The %INDEX function is used to see if &C contains an uppercase letter. If an uppercase letter is found, its position is stored in &INDEX. For &C=M, the %INDEX function returns a 13 (since M is the 13th letter of the alphabet).
- ❸ If an uppercase letter is found (&INDEX > 0), it is converted using the %SUBSTR function ❹.
- ❺ In the %SUBSTR function, &INDEX becomes the column indicator for the matching lowercase letter. For &INDEX=13, the lowercase m is selected and assigned to &C. The value of the macro variable &C has now been converted to lowercase.
- ❻ &C is then appended (converted or not) onto &RESULT, which continues to grow until the entire string has been checked.

- ❶ The macro variable &RESULT contains the converted string, and this will be the final resolved value of the macro call.

**SEE ALSO:** Sylvia Tze (2000) discusses a macro that, although not a macro function, will convert all characters except the first character to lowercase.

## 10.6.5 %TRIM and %QTRIM

The %TRIM macro function (introduced in Section 7.2.8) removes trailing blanks from the argument and returns the result. This macro uses a %DO with a negative index value to step backwards to the last non-blank character.

```
%macro trim(value);
%local i;
%do i=%length(&value) %to 1 %by -1; ❶
  %if %qsubstr(&value,&i,1)^=%str( ) %then %goto trimmed; ❷
  %end;
  %trimmed: %if &i>0 %then %substr(&value,1,&i); ❸
%mend;
```

- ❶ The macro uses a %DO loop to step through the argument from the last character forward until the first nonblank character is encountered. Notice that the %LENGTH function is used to determine the last character. Usually, the value returned by the %LENGTH function will be the last nonblank character, and the %DO would only iterate once.
- ❷ %QSUBSTR is used to examine the  $i^{\text{th}}$  character and if it is not a blank then our search is over.
- ❸ The %SUBSTR function is then used to select all the characters in the argument up to and including the last character (the  $i^{\text{th}}$  character) examined by the %QSUBSTR ❷.

## 10.6.6 %DATATYP

Since macro variables are not designated as numeric or character, it may become necessary to determine whether a given macro variable contains a value that could be used as a number. This can be important if arithmetic operations (including numeric comparisons) are to be performed.

### SYNTAX:

```
%DATATYP(argument)
```

### VALUE RETURNED:

NUMERIC	The value could be interpreted as a number
CHAR	The value cannot be interpreted as a number

The macro %DATATYP returns whether the value of the parameter is potentially numeric or character. Although it does not detect numbers written in other bases such as hexadecimal, the macro is not limited to integers and will even detect numbers written in scientific notation. The single parameter is the value that you want to have checked and the macro returns either NUMERIC or CHAR.

### Program 10.6.6 (SAS Log): Showing the Use of the %DATATYP Macro

```
224 %let d1 = 2.54;
225 %let typd1=%datatyp(&d1);
226 %put &=d1 &=typd1;
D1=2.54 TYPD1=NUMERIC ❶
227
228 %let d2 = 4.3e3;
229 %let typd2=%datatyp(&d2);
230 %put &=d2 &=typd2;
D2=4.3e3 TYPD2=NUMERIC ❷
231
```

```

232 %let d3 = abcd3;
233 %let typd3=%datatyp(&d3);
234 %put &=d3 &=typd3;
D3=abcd3 TYPD3=CHAR ③

```

- ① The number does not need to be an integer.
- ② Scientific notion is seen as numeric.
- ③ Values that cannot be interpreted as numeric cause a CHAR to be returned.

## 10.6.7 %COMPSTOR

For those sites that make use of stored compiled macros, or that have an existing catalog of compiled macros (see Section 10.3.2), or that want to optimize the access to the autocall macros supplied by SAS, the %COMPSTOR macro can be used to compile and permanently store selected macros in the SAS autocall library. Once these macros have been compiled, it is no longer necessary to include the SASAUTOS *fileref* in the SASAUTOS system option for these macros.

### SYNTAX:

```
%COMPSTOR (PATHNAME=unquoted_path)
```

### VALUE RETURNED:

Compiled autocall macros

When the %COMPSTOR macro is executed, it creates a *libref* named SASMACR that points to the location (pathname=) passed into the macro. A SASMACR catalog is then created in that directory, the compiled macros are stored in the catalog, and a SASMSTORE= system option is executed that points to this directory (*libref*= SASMACR).

### Program 10.6.7: Compiling Autocall Macros Using %COMPSTOR

```
%compstor (pathname=c:\PERMMACS)
```

The %COMPSTOR macro call in Program 10.6.7 establishes the following:

- The *libref* SASMACR pointing to the C:\ PERMMACS directory
- The setting of the SASMSTORE= system option to SASMACR
- The SASMACR catalog in this library (SASMACR.SASMACR)

Compiled versions of selected autocall macros supplied by SAS

The current version of %COMPSTOR will compile these autocall macros:

- %CMPRES, %QCMPRES
- %COPYRTE
- %DATATYP
- %LEFT, %QLEFT
- %MDARRAY
- %TRIM, %QTRIM
- %VERIFY

Two of these macros, %COPYRTE and %MDARRAY, are compiled but not mentioned in the SAS 9.4 documentation.

There are a couple of caveats and even a couple of side benefits that you should be aware of before using %COMPSTOR:

- %COMPSTOR is itself an autocall macro, so SAS will need to have access to its own autocall library when %COMPSTOR is executed.
- %COMSTOR issues its own SASMSTORE= option. This macro will clear any existing SASMSTORE definition (not the library, just the SASMSTORE= connection). Be sure to reestablish your own option settings after running %COMPSTOR (see the %STOREOPT macro—Program 12.3.1 and the %HOLDOPT macro—Program 11.2.4). Of course, when you do reestablish the options, be sure to include the location of the new SASMACR catalog.
- You cannot use a *libref* as the argument to %COMPSTOR, so you must know the physical path to the directory of interest. If you do need to use a *libref* or a *fileref* you can do so through the use of the %SYSFUNC macro function. Assuming that the *libref* MYMACS points to C:\PERMMACS, %SYSFUNC can call the PATHNAME function to retrieve the physical location.

#### Program 10.6.7 (Continued): Using PATHNAME to Retrieve a Physical Path

```
%compstor(pathname=%sysfunc(pathname(mymacs)))
```

You will now have two *librefs* pointing to the same location (MYMACS and SASMACR).

A side benefit of using %COMPSTOR is that there are a couple of undocumented macros in %COMPSTOR that will be added to the SASMACR catalog.

If you already have a catalog containing compiled macros of your own, you can use %COMPSTOR to add the autocall macros to your existing SASMACR catalog. In the following example, the *libref* MYMACS points to a SASMACR catalog in which you have a number of your own macros. %COMPSTOR adds the autocall macros.

#### Program 10.6.7 (Continued): Adding Compiled Autocall Macros to a Stored Compiled Library

```
options mstored sasmstore=mymacs;
%compstor(pathname=%sysfunc(pathname(mymacs)))
```

You should be aware that if your catalog already contains a macro with the same name as one of the autocall macros, your version will be overwritten when %COMPSTOR is executed.

**SEE ALSO:** Thorton (2014) uses the PATHNAME function to determine a physical location.

## 10.6.8 Autocall Macros That Assist with Color Conversions

There are a series of autocall macros that can be used to assist with the conversion of colors from one color designation to another. These color conversions can be especially useful in SAS/GPGRAPH when specific colors are requested, but not in the color system being applied in your program.

**Table 10.6.8: Color Conversion Macros**

Macro Name	Macro Purpose
%COLORMAC	Compiles the macros noted in %HELPCLR
%HELPCLR	Provides help and some documentation on the color conversion macros
%CMY	Creates an RGB color name from CMY components
%CMYK	Creates a CMYK color name from CMYK components
%CNS	Creates an HLS color name from Color Naming System
%HLS	Creates an HLS color name from HLS components
%HSV	Creates an HLS color name from HSV components

Macro Name	Macro Purpose
%RGB	Creates an RGB color name from RGB components
%HLS2RGB	Converts an HLS color name to an RGB color name
%RGB2HLS	Converts an RGB color name to an HLS color name

The execution of the %COLORMAC macro defines and compiles the other macros shown in Table 10.6.8. %HELPCLR provides some usage notes on each of these macros. You can use %HELPCLR to see the usage notes on the primary macros generated by %COLORMAC. Program 10.6.8 demonstrates the use of some of these macros.

#### Program 10.6.8: Using the Color Conversion Macros

```
%helpclr() ❶
%helpclr(all) ❷
%helpclr(rgb) ❸
%colormac() ❹
%put %nrstr(%rgb(100,100,0) = ) %rgb(100,100,0); ❺
%put %nrstr(%RGB2HLS(CXFFFF00) = ) %RGB2HLS(CXFFFF00); ❻
%put %nrstr(%HLS2RGB(H0B480FF) = ) %HLS2RGB(H0B480FF); ❼
```

The SAS Log for program 10.6.8 shows the result of these macro calls.

#### Program 10.6.8 (SAS Log): Showing Usage Notes for %HELPCLR and %COLORMAC

```
72  * Program 10.6.8.sas
73  * Using the %HELPCLR and %COLORMAC macros to convert colors;
74  %helpclr() ❶
          Color Utility Macros Help

HELP is currently available for the following macros

      CMY          CMYK          CNS          HLS
      HVS          RGB           HLS2RGB     RGB2HLS

Enter %HELPCLR(macroname) for details on each macro,
or %HELPCLR(ALL) for details on all macros.

75  %helpclr(all) ❷
          Color Utility Macros

The Color Utility macros are designed to lessen the
burden of coding SAS/GPGRAPH colors. They can be used
wherever SAS/GPGRAPH colors may be used. Each of the
available macros is described briefly below.

%CMY(cyan, magenta, yellow)
Creates RGB Color Name from CMY components.
cyan    = 0 to 100 Percent Cyan
magenta = 0 to 100 Percent Magenta
yellow  = 0 to 100 Percent Yellow

Example: %CMY(100,0,100) is CX00FF00 (green).

. . . . portions of the SAS Log are not included . . . .

76  %helpclr(rgb) ❸

%RGB(red, green, blue)
Creates RGB Color Name from RGB components.
```

```

red    = 0 to 100 Percent Red
green = 0 to 100 Percent Green
blue   = 0 to 100 Percent Blue

Example: %RGB(100,100,0) is CXFFFF00 (yellow).

77  %colormac() ④

*** Color Utility macros are now available ***

For further information on the Color Utility macros
enter:
  %HELPCLR(macroname) - Specific macro information
  %HELPCLR(ALL)       - All macro information
  %HELPCLR            - List of macro names
78  %put %nrstr(%rgb(100,100,0) = ) %rgb(100,100,0); ⑤
%rgb(100,100,0) = CXFFFF00
79  %put %nrstr(%RGB2HLS(CXFFFF00) = ) %RGB2HLS(CXFFFF00); ⑥
%RGB2HLS(CXFFFF00) = H0B480FF
80  %put %nrstr(%HLS2RGB(H0B480FF) = ) %HLS2RGB(H0B480FF); ⑦
%HLS2RGB(H0B480FF) = CXFFFF00

```

- ➊ The macro call without arguments shows which macros are described by %HELPCLR. Unlike the other macros shown in this section, this macro is not established by %COLORMAC.
- ➋ Show complete documentation for all the macros described by %HELPCLR.
- ➌ Show the documentation for the macro %RGB.
- ➍ %COLORMAC compiles the other macros and makes them available.
- ➎ The decimal code of 100,100,0 converts to an RGB code of CXFFFF00 using the %RGB macro.
- ➏ The %RGB2HLS macro converts CXFFFF00 to H0B480FF in HLS.
- ➐ %HLS2RGB converts the HLS code back to RGB.

### 10.6.9 Surfacing Other Autocall Macros Supplied by SAS

The autocall macros mentioned in the earlier sections of 10.6 are only a small fraction of the available autocall macros that are supplied by SAS. Many are designed to only be used internally, however some can be quite useful.

The DATA step in Program 10.6.9 reads an incoming *fileref* and lists the names of all the \*.SAS files and the names of the macro definitions that it contains. Because it is not currently possible to directly read a series of files (SAS programs) across members of a concatenated library, the individual members of the library are processed one at a time.

#### Program 10.6.9: Listing of SAS programs and Macro Definitions

```

%macro listsas(f_ref=);
%local cdloc loccnt i;
%let cdloc = %sysfunc(pathname(&f_ref)); ①
%let cdloc = %sysfunc(translate(&cdloc,%str( ),%str(%),
                                %str( ),%str(%)),
                                %str( ),%str(%'))); ②
%let cdloc = %sysfunc(tranwrd(&cdloc,%str(o  C:),%str(o__C:))); ③
%let loccnt = %sysfunc(count(&cdloc,%str(c:),i)); ④

data ListSAS(keep=f_macname macname);
  length fname $250 f_macname macname $32;
  %do i = 1 %to &loccnt; ④
    done=0;
    length filein&i $250;
    fname=' ';
    do until(done);

```

```

infile "%qscan(&cdloc,&i,%str(_))\*.sas" ⑤
      filename=filein&i truncover
      end=done lrecl= 1000 dlm= ' (/)% ; ';
input @; ⑥
fname=filein&i; ⑦
f_macname = scan(fname,-2,'\.'); ⑧
pos = find(_infile_,'%macro ', 'i') ; ⑨
if pos;
  input @(pos+6) macName $; ⑩
  output ListsAS; ⑪
end;
%end;
stop;
run;
%mend listsas;
%listsas(f_ref=sasautos) ⑫

```

- ① The PATHNAME function is used to return the physical path for this *fileref* ⑫.
- ② Various character strings are removed that would otherwise prevent the parsing of the list of individual files.
- ③ The number of files in the concatenated *fileref* are counted. Notice in this solution that only files on the C: drive will be counted.
- ④ A %DO loop is used to step through each incoming file individually.
- ⑤ A %QSCAN is used to retrieve the physical name/location of the file that is to be read.
- ⑥ The input buffer is refreshed. This causes the name of the file to be reloaded ⑦.
- ⑦ The name of the current file is recovered from the temporary variable FILEIN&I, which was established through the use of the FILENAME= option on the INFILE statement.
- ⑧ The full name of the incoming file is parsed to recover just the name of the SAS program.
- ⑨ Each row of each program is searched for the %MACRO keyword. The assumption is then made that the next word will be the name of the macro in the %MACRO statement.
- ⑩ The macro name is then retrieved and stored in the variable MACNAME.
- ⑪ The name of the detected macro and the file that it resides in is written to the data set LISTSAS.
- ⑫ When used on the automatic *fileref* SASAUTOS, the %LISTSAS macro will surface all the macros stored in the SAS autocall library. A portion of the data set WORK.ListSAS is shown in Figure 10.6.9.

**Figure 10.6.9: A Portion of the Data Set Created by Program 10.6.9.**

VIEWTABLE: Work.Listsas		
	f_macname	macname
1	aarfm	aaRFM
2	aarfm	EM_RFMs_CONTROL
3	aarfm	EM_RFMs_TRAN_2_CUST
4	aarfm	EM_RFMs_SET_SCORE
5	aarfm	EM_RFMs_NESTED_RANK
6	aarfm	EM_RFMs_INDEPENDENT_RANK
7	af	af
8	angle	angle
9	annomac	ANNOMAC
10	annomac	ARROW
11	annomac	sequenc
12	annomac	sequence

This basic approach to detect and surface autocall macro definitions was first proposed by Peter Crawford (Crawford, 2016).

**MORE INFORMATION:** Program 13.1.3 also reads SAS programs as an input stream.

**SEE ALSO:** Lund (2016) uses two of the SAS/GRAFH annotate autocall macros.

## **Part 3 Dynamic Macro Coding Techniques**

<b>Chapter 11 Writing dynamic Programs.....</b>	<b>281</b>
<b>Chapter 12 Examples of Dynamic Programs .....</b>	<b>333</b>

A dynamic program is one, which by necessity, is not completely defined prior to execution. Many, if not most, SAS programs are static and the programmer determines through the use of DATA step and PROC step statements the order and logic of execution. In static programs, when the programmer *knows* about data exceptions and special cases, he or she must *hardcode* logic to handle them. Dynamic programs, on the other hand, may use the analysis data itself, control data, or even information gathered from the operating environment to determine the path and logic of execution. The programmer who writes dynamic programs has the ability to create generalized programs that will execute correctly on a wider variety of data, and in a wider variety of situations.

The examples in Section 6.4.1 use a data set to control the processing of the macro. This type of control frees the macro programmer from hard coding data-specific information into the macro. In the %REGIONRPT example (Program 6.4.1b) the macro will work for any number of regions. With minor changes to the macro (see Chapter 6 questions 2 and 3) even further data dependencies can be removed from the macro itself.

Programs and macros that write themselves or adjust to the data are dynamic in nature, and it is a huge programming advantage to be able to write macros general enough to be data independent. Programming successfully at this level is much more difficult. The %REGIONRPT macro mentioned above assumes that the name of the variable used to classify the groups will be REGION and that it is a character variable. The macro would have been much more dynamic if it could have accepted a list of one or more variables to use and to be able to determine whether these variables were character or numeric. The most general macros will have the fewest data dependencies.

In this part of the book two chapters are dedicated to dynamic programming techniques. Chapter 11, discusses the characteristics and programming fundamentals of dynamic macros in more detail, while Chapter 12 illustrates these principles with a number of example macros.

# Chapter 11: Writing Dynamic Programs

<b>11.1 Dynamic Programming Introduction and Design Elements.....</b>	<b>282</b>
11.1.1 A Short Macro Language Review from the Perspective of a Dynamic Programmer .....	282
11.1.2 Elements of a Dynamic Program .....	287
11.1.3 Creating Data Independence.....	289
11.1.4 Elements for Making a Program Dynamic.....	289
11.1.5 Controlling the Program with Data .....	290
11.1.6 List Processing Basics.....	291
11.1.7 Iterative Step Execution.....	291
11.1.8 Building Statements .....	291
<b>11.2 Information Sources .....</b>	<b>293</b>
11.2.1 Using SASHELP Views .....	293
11.2.2 Using SQL DICTIONARY Tables.....	296
11.2.3 Automatic Macro Variables .....	297
11.2.4 %SYSFUNC and DATA Step Functions .....	297
11.2.5 Retrieving Operating System Information.....	300
11.2.6 Using Data Set Metadata.....	300
11.2.7 Using Data Tables to Control a Process .....	303
11.2.8 Creating and Using Control Files .....	304
11.2.9 Using SET Statement Options .....	306
<b>11.3 Using &amp;&amp;VAR&amp;I Constructs as Vertical Macro Arrays .....</b>	<b>307</b>
11.3.1 Creating the List of Macro Variables .....	308
11.3.2 Resolving &&VAR&i .....	308
11.3.3 Stepping through a List of Data Sets.....	309
<b>11.4 Horizontal Lists .....</b>	<b>309</b>
11.4.1 Creating Horizontal Lists .....	310
11.4.2 Resolving Horizontal Lists .....	310
11.4.3 Stepping through the Horizontal List.....	311
11.4.4 Counting the Items in a List.....	312
<b>11.5 Using CALL EXECUTE .....</b>	<b>313</b>
<b>11.6 Writing %INCLUDE Programs .....</b>	<b>315</b>
<b>11.7 Writing Applications without Hardcoded Data Dependencies.....</b>	<b>317</b>
11.7.1 Generalized and Controlled Repeatability.....	318
11.7.2 Setting Up Project Control Files.....	319
11.7.3 Using Control Files to Build Macro Variable Lists .....	321
11.7.4 Using Control Files to Create Empty Data Sets.....	322
11.7.5 Using Control Files to Create Data Validation Checks Dynamically.....	324
<b>11.8 Building SAS Statements Dynamically .....</b>	<b>327</b>
<b>11.9 More Than Just the Macro Coding .....</b>	<b>328</b>
11.9.1 Naming Conventions .....	328
11.9.2 Directory Structure.....	330
11.9.3 Using the AUTOEXEC File.....	333
11.9.4 Unifying <i>fileref</i> and <i>libref</i> Definitions .....	334

Because dynamic programming techniques are such an integral part of the macro language, many of the examples in other sections of this book also demonstrate the topics that are outlined in the following sections of this chapter. The topic of dynamic programming is first introduced in Section 5.2.2, where several basic issues relating to the topic are covered. Then a somewhat more complex dynamic programming example is described in Program 6.4.1b. This chapter expands on those basic introductory examples.

It is usually not easy to begin writing this type of macro. Most macro programmers write a great many macros that are, to varying degrees, dynamic before this type of programming becomes second nature. This chapter illustrates those characteristics or elements that are common to many macros that dynamically build code. Learn to watch for and to program these elements and the entire process becomes easier.

In dynamic programming, the control and flow of the program is driven by the data or by some other external source of control information. This information is transferred to macro variables, which are in turn used to drive the dynamic process. Key to this process is the ability to convert values stored in data sets into macro variables. Once stored in macro variables, these values can be used by the macro language in subsequent steps. Section 6.1 describes the use of the SYMPUTX routine in a DATA step, and Section 6.2 shows you how to use PROC SQL's INTO clause to build macro variables in an SQL step.

Data sets are often created for the sole purpose of directing and controlling the flow of the dynamic application. These control data sets can be used to store any project, data set, or variable-specific information that will be needed by the programs in the application. This information can include the names of data sets, the variables within those data sets, variable attributes, and field-check information. These control data sets are then used to create a series of macro variables. Application programs that require project-specific information, such as the names of data sets and variables, use these macro variable lists to dynamically build the SAS code needed for the project, data set, or variable of interest.

Dynamic programming is an advanced macro topic. Creating the SAS programs and macros that can take advantage of dynamic techniques, such as &&VAR&I macro variables, is not initially easy. You might need to practice and work with the techniques discussed in this chapter for a while before dynamic programming techniques become second nature for you.

For the examples in this chapter and indeed for all of the code examples throughout the book, if you want to execute these sample programs, then be sure to follow the setup instructions. Remember that all of the data sets and programs are available for download, so you do not need to retype either the code or the data. For instructions on accessing and setting up the programs and data, see the "Example Code and Data" section within this edition's "About This Book" front matter.

**SEE ALSO:** Several examples of dynamic programming can be found in Blood (1992), Carpenter and Callahan (1988), and Carpenter (1997). Carpenter and Smith (2000) discuss the design aspects of a dynamic application. Muller (2014 and 2016) discusses library structure considerations. A number of considerations when generalizing programs and structures are presented by Holland (2016).

---

## 11.1 Dynamic Programming Introduction and Design Elements

As been stated elsewhere, writing dynamic SAS macro applications is not initially intuitive for most macro programmers. The learning curve can be both steep and long. The rewards however are substantial. This section provides an overview of the basic concepts needed to start the process of learning. Each of these topics is covered in more detail later in this chapter.

---

### 11.1.1 A Short Macro Language Review from the Perspective of a Dynamic Programmer

Many of the dynamic programming techniques require a basic understanding of macro language elements introduced in earlier sections of this book. A few of the more critical ones are reviewed in this section.

## Timing Is Everything

Take a few moments to review Section 1.4, especially the discussion centered on Figure 1.4d. Hopefully, if you are starting to tackle the learning of dynamic programming techniques, you tend to be intuitively aware of this diagram and the consequences of not understanding its implications. If you are going to successfully use dynamic programming techniques, it is essential that you have a deep understanding of the timing of the resolution and execution of macro language elements.

Remember that macro language elements are resolved and executed first. If you think about it, you are going to be using the macro language to write SAS code. The code has to be written before it can be used (parsed, compiled, and executed); therefore the macro language elements have to come first. Programmers that don't keep this execution order in mind will not be successful when attempting to write dynamic applications.

## Processing Macro Language Tokens

The macro scanner searches for macro language elements by looking for the & and % characters, which are known as macro language triggers. These triggers are used to identify macro language elements. These macro elements are broken up into tokens or words, which are the smallest meaningful unit of code. These words are used to form chunks (not a technical term) of macro language code. It is these chunks that form the macro language elements that are passed to the macro facility for processing. Notice that I carefully did not say that the “chunk” of macro code was a statement. The chunk or macro language element might be a single macro variable reference (&DSN), a macro function call (%sysfunc(exist(&dsn))), a complete macro statement (%LET DSN=CLINICS;), a macro call (%ABC), or even a macro definition (%MACRO to %MEND statements). Understanding how SAS processes these macro language elements can be very important when using advanced programming techniques.

**SEE ALSO:** Burlew (2014, Chapter 2 pp 22-24) goes into nice detail on macro tokens and how they are handled by the macro processor.

## Macro Variable Lists and the Resolution of Macro Variables

Most dynamic applications involve the use of techniques that are collectively known as list processing techniques (Fehd and Carpenter, 2007). In list processing the elements of the ‘list’ are used to drive a process, usually once for each element in the list. The list might be stored as a series of items in a single macro variable (horizontal list) or as a series of macro variables each containing a single item (vertical list).

While the macro language does not support arrays *per se*, these vertical and horizontal lists can effectively function as arrays. In terms of the computer, an array is a list of items that can be addressed by name and index. Taken together, the name of the list and its index form the call to the array. Both horizontal and vertical macro lists have these properties. Section 14.2 discusses some alternatives to the vertical and horizontal macro variable lists.

### Vertical Lists (&&VAR&I)

The vertical list consists of a series of macro variables that are indexed and are named with a common root word. The list is addressed using three ampersands, generally in the form of &&VAR&I, which is sometimes referred to as the “amper amper var amper i construct”. This is an indirect reference—it is a macro variable that holds the name of a macro variable, which holds the value of interest. In Program 11.1.1a a list of three macro variables is created. Each is named with the root name DSN followed by the index number.

#### Program 11.1.1a: Naming and Using a Vertical List

```
%let dsn1 = ae;
%let dsn2 = cm; ①
%let dsn3 = demog;
%let dsncnt = 3;

%macro printall;
```

```
%do j = 1 %to &dsncnt; ②
  proc print data=&&dsn&j; ③
    run;
  %end;
%mend printall;
```

- ① A vertical list of macro variables is created with the root name of DSN
- ② Lists are usually processed within a %DO loop. Here &J serves as the index value.
- ③ The &J<sup>th</sup> value of the list is accessed via the &&DSN&J macro variable reference.

For &j = 2, &&dsn&j resolves as shown in Table 11.1.1a:

**Table 11.1.1a: Resolution of the &&VAR&I Macro Variable Construct**

Macro Variable Reverence	Macro Variable Resolution Notes
<u>&amp;&amp;dsn&amp;j</u> ↓	Effectively this is the array (vertical list) call and it is resolved left to right
<u>&amp;&amp;</u> <u>dsn</u> <u>&amp;j</u> ↓      ↓      ↓	The macro scanner resolves double ampersands into one ampersand and the DSN is constant text. The index value, &J, is resolved to 2.
<u>&amp;</u> <u>dsn</u> <u>2</u> ↓	This creates a new macro variable reference
<u>&amp;dsn2</u> ↓	This is a macro variable reference that can be further resolved
cm	&DSN2 is resolved to CM

### &&VAR—A Special Case of the Vertical List

Like the &&VAR&I construct, macro variables of this form always name another macro variable. In a vertical list there is an index to identify the element within the list. When there is only one element in the list, the index variable is not needed, and three ampersands appear at the start of the reference. Essentially this is a vertical list with one element. Program 11.1.1b prints the specified data table by passing the table name into the macro.

#### Program 11.1.1b: Passing the Value Stored in a Macro Variable

```
%let dsn=macro3.clinics;

%macro printit(dset);
  title1 "First 10 obs of &dset";
  proc print data=&dset(obs=10);
  run;
%mend printit;
%printit(&dsn)
```

Before the macro is called, &DSN must be resolved and the macro call to %PRINTIT becomes:

```
%printit(macro3.clinics)
```

During the execution of the %PRINTIT macro, the data set name is being stored in two places at the same time &DSN and &DSET. We could store the name in only one place, and pass the name of the macro variable instead of its value. In Program 11.1.1c the name of the macro variable is passed into the macro.

**Program 11.1.1c: Passing Only the Name of the Macro Variable**

```
%let dsn=macro3.clinics;

%macro printit(dset);
  title1 "First 10 obs of &&&dset";
  proc print data=&&&dset(obs=10);
    run;
%mend printit;
%printit(dsn)
```

Inside the macro %PRINTIT, the macro variable &&&DSET resolves as shown in Table 11.1.1b:

**Table 11.1.1b: Resolution of the &&VAR&I Macro Variable Construct**

Macro Variable Reference	Macro Variable Resolution Notes
&&&dset ↓	Two resolution passes are required
&& <u>&amp;dset</u> ↓ ↓	The first two ampersands are resolved to one ampersand
& <u>dsn</u> ↓	&DSET resolves to the constant text DSN
&dsn ↓	The macro variable &DSN can be resolved further
macro3.Clinics	

The macro variable &&&DSET contains the name of a macro variable (&DSN) that contains the value of interest.

**SEE ALSO:** Mason (2016) uses triple ampersands to delay resolution of a macro variable whose name was being built using another macro variable.

**Horizontal Lists**

In a horizontal list one macro variable is used to hold a series of values that are to be processed individually. Usually the list of macro variables is generated using an SQL step and is subsequently parsed using either the %QSCAN or %SCAN function.

The macro variable that holds the list of values is parsed within a %DO loop using the %QSCAN function. In Program 11.1.1d a list of data sets is stored in the macro variable &DSNLIST. If you think of this list as an array, &DSNLIST would be the array name, the %DO loop index (&J) would be the array index (word number), and the array call would be the call to the %QSCAN function, %qscan(&dsnlist,&j,%str( )).

**Program 11.1.1d: Using a Horizontal Macro List**

```
%macro printall;
  %local dsnlist dsncnt j;
  %let dsnlist = ae cm demog; ④
  %do j = 1 %to %sysfunc(countw(&dsnlist));; ⑤
    title1 "First 10 obs of %qscan(&dsnlist,&j,%str( ))⑥";
    proc print data=%qscan(&dsnlist,&j,%str( ))⑥(obs=10);
```

```

run;
%end;
%mend printall;
%printall

```

- ④ The macro variable &DSNLIST contains a number of data set names.
- ⑤ The %DO loop is used to step through the list, with the %DO loop index (&J) serving as the word counter.
- ⑥ %QSCAN is used as to retrieve the &J<sup>th</sup> word (data set name) in the list. Effectively the call to %QSCAN becomes the array reference.

### Horizontal versus Vertical Lists

When using the macro language to process a list of values, you will usually use either a vertical or horizontal list. However, there are advantages and disadvantages to each, and these are highlighted in Table 11.1.1c.

**Table 11.1.1c: Horizontal versus Vertical Lists**

Consideration	Vertical List	Horizontal List
⑦ Number of macro variables	One macro variable for each item in the list	One macro variable contains all items in the list
⑧ Storage	Available memory	Maximum macro variable size is 64K bytes per macro variable
⑨ List reference	&&VAR&I	%QSCAN(&VAR,&I,%STR())
⑩ Items in the list	Each item can contain 64K bytes. No word delimiters are needed.	Each item must be definable with a word delimiter. Maximum of 64K bytes across all items.
⑪ Count of items	Item count generally must be known	Item count does not need to be known
<ul style="list-style-type: none"> <li>⑦ The horizontal list only requires a single macro variable as opposed to a list of macro variables. It is often easier to force this single macro variable into the local table especially when created using PROC SQL.</li> <li>⑧ The number of items that can be stored in a vertical list is virtually unlimited, as it depends only on available memory. The horizontal list is limited to a single macro variable, which can contain only 64K bytes.</li> <li>⑨ The list reference for a vertical list contains three ampersands and some programmers find this syntax disconcerting. The reference for a horizontal list is more complex, a call to the %SCAN or %QSCAN function; however, it does not require double or triple ampersands.</li> <li>⑩ When storing items in a vertical list, you do not need to worry about embedded word delimiters, and you can store up to 64K bytes in each macro variable. Items stored in the horizontal list must have word delimiters and each word must not have an embedded delimiter.</li> <li>⑪ Usually when you make a list, be it horizontal or vertical, you will know the number of items. When the number of items in the list is unknown, it is easier to detect and properly process a horizontal list of items.</li> </ul>		

**SEE ALSO:** The basics of macro list processing are covered in Fehd and carpenter (2007), Carpenter (2009), Roosenbloom and Carpenter (2014).

### 11.1.2 Elements of a Dynamic Program

While not always present in every dynamic macro program, there are certain elements that tend to be in macros that dynamically build code. These elements include the following:

- Macro variable values are based on an information source like a SAS data set.
- The SYMPUTX routine or the SQL INTO clause is used to build the macro variables.
- Macro lists, both vertical and horizontal, are common.
- The number of elements in the list is saved in a macro variable.
- %DO loops are used to step through the macro list.

As you learn to write this type of macro, watch for these elements and notice how they fit together. The pattern formed by these elements is repeated over and over again in the examples in this book. Browse through the examples noted in Table 11.2, notice the pattern, and look for the elements described above.

#### Building Macro Variables Based on an Information Source

Whatever the information source, the information must be transferred to macro variables before it can be used by a dynamic macro. Very often a list of values is needed and you will need to build either a vertical or horizontal list of values. Program 11.1.2a creates a vertical list of clinic numbers using a DATA step.

##### Program 11.1.2a: Building a Vertical List Using the DATA Step

```
proc sort data=macro3.clinics(keep=clinnum)
          out=clincodes
          nodupkey; ①
  by clinnum;
  run;
data _null_; ②
  set clincodes end=eof; ③
  call symputx(catt('clin',_n_),clinnum); ④
  if eof then call symputx('clincnt',_n_); ⑤
  run;
%put _user_;
```

- ① The NODUPKEY option is used to build a unique set of clinic codes.
- ② Usually a new data set is not needed and a DATA \_NULL\_ step is used.
- ③ The END= option is used to detect the last observation read by the SET statement.
- ④ Each unique value of CLINNUM will be assigned to its own unique macro variable. The CATT function converts \_n\_ to character, left-justifies, and trims the converted value. The CAT function could also have been used.
- ⑤ On the last observation the value of \_N\_ will contain the number of unique clinic numbers. This value is assigned to the macro variable &CLINCNT.

The individual elements of the list will be &CLIN1, &CLIN2, and so on. A %PUT can be used to show the generated macro variables. Some of these are shown here:

#### Program 11.1.2a (Partial SAS Log): Showing Some of the Generated Macro Variables

```
34  %put _user_;
GLOBAL CLIN1 011234
GLOBAL CLIN10 043320
GLOBAL CLIN11 046789
GLOBAL CLIN12 049060
GLOBAL CLIN13 051345
. . . portions of the SAS Log not shown . . .
```

A horizontal list of values could have also been created. While vertical lists can easily be created in either the DATA step or in an SQL step, horizontal lists are much easier to create in an SQL step. Program 11.1.2b also creates the list of clinic numbers, however this time in a horizontal list.

#### Program 11.1.2b: Building a Horizontal List Using PROC SQL

```
proc sql noprint;
  select distinct clinnum ⑥
    into :clinlist separated by ' ' ⑦
      from macro3.clinics(keep=clinnum);
  %let clincnt=&sqlobs; ⑧
  quit;
%put &=clincnt &=clinlist;
```

- ⑥ The unique values of CLINNUM are determined.
- ⑦ The individual values of CLINNUM are added to the macro variable &CLINLIST as a space separated list.
- ⑧ SQL stores the number of items in the list in &SQLOBS and that value is transferred in the macro variable &CLINCNT.

A %PUT can be used to write these macro variables to the SAS Log. A partial list of clinic numbers is shown here:

```
42  %put &=clincnt &=clinlist;
CLINCNT=27 CLINLIST=011234 014321 023910 024477 026789 031234 033476 036321
038362 043320
```

#### Using the Macro Variables Dynamically

Whether your list of macro variables was created as a vertical or a horizontal list, you will need to step through this list one element at a time, and a %DO loop of some kind is usually used. When you have a vertical list, the index of the iterative %DO loop becomes the subscript to the list of macro variables. For the horizontal list the %SCAN function is used to parse the list and the loop index becomes the word number. These loops may be used to iteratively execute a block of statements (see Section 6.4.1) or to build a specific statement from the inside (see Sections 5.3.2, and Exercise 4 in Section 6.6).

If a vertical list of macro variables contained a series of data set names, a SET statement could be constructed using a %DO loop, where &ROOT2 contains the second data set name to include.

```
set
%do i = 1 %to &rootcnt; ⑨
  &&root&i ⑩
%end;
; ⑪
```

- ⑨ &ROUTCNT is the total number of data sets to be read.
- ⑩ This indirect macro variable reference is used to point to the individual data set names. When &I is 3, &&ROOT&I becomes &ROOT3, which refers to the third data set.

- ⑪ The SET statement is completed with a semicolon. The semicolons associated with the %DO and %END statements are “used up” by the macro facility, while this one is left behind and becomes the terminator of the SET statement.

The power of this technique is that the programmer who wrote the above SET statement had no idea of what the names of the data sets were to be or even how many data sets would be listed. This SET statement will change dynamically as the list of data sets changes.

### 11.1.3 Creating Data Independence

One of the goals of dynamic programming is to achieve data independence. To the extent possible, we would like items in our programs that depend on the data—items like data set names, variable names, variable values, and perhaps formats—to be replaced with macro variables. This makes our programs independent of the data, and therefore more generalized. The simple PROC SUMMARY step in Program 11.1.3a operates on a specific SAS data set and uses three variables from that data set.

#### Program 11.1.3a: Nongeneralized PROC SUMMARY

```
title1 'Summary of Height and Weight';
proc summary data=macro3.clinics;
  class region;
  var ht wt;
  output out=sumry n= mean= stderr=/autoname;
run;
```

When generalizing we ask ourselves what will change if I use a different data set or a different classification variable? Data-dependent items are bolded in Program 11.1.3a and converted to macro variables in Program 11.1.3b.

#### Program 11.1.3b: Generalized PROC SUMMARY

```
%macro sumry(dsn=,classlst=,varlst=);
title1 "Summary of &varlst";
proc summary data=&dsn;
  class &classlst;
  var &varlst;
  output out=sumry n= mean= stderr=/autoname;
  run;
%mend sumry;
%sumry(dsn=sashelp.shoes, classlst=region subsidiary, varlst=sales)
```

Removing the data dependencies is an important step in generalizing your program. It is a necessary step when converting your programs to work dynamically; however, there are other steps that must also take place (see Section 11.1.4 for other steps toward building dynamic programs).

### 11.1.4 Elements for Making a Program Dynamic

As is mentioned earlier, when you are first starting to create dynamic programs, you need to consciously think of the process and the steps. As you practice, these will start to come naturally. Eventually you will be writing dynamic programs from scratch; however, in the beginning you will probably often be converting an existing program into one that is more dynamic.

Remember that dynamic programs generally involve the use of lists (although sometimes it is a list of one). So be sure you are comfortable with using macro variable lists. Most programs that build and process a list have certain elements. Initially, as you learn to create programs that run themselves automatically, make a conscious effort to make sure that these elements exist in your program. They may not be intuitive as you learn; however, as you become proficient at building this type of program, the elements themselves become automatic.

The order that you think about and apply these elements to your program is less important than making sure that each is considered.

1. Identify a source for the information that will drive the process.
2. Generalize or write your program to accept a macro variable list.
3. Use a %DO loop to process across all the items in the list.
4. Within the loop, access the individual elements by using either a vertical or horizontal list form:

	<b>Vertical Lists</b>	<b>Horizontal Lists</b>
List Reference	&&item&i	%qscan(&itemlist,&i,%str( ))
List characteristics	One macro variable per item with virtually unlimited number of items permitted.	One macro variable stores all items; maximum of 64k of information allowed; requires word separators.

5. Build a list of macro variables (or a macro variable containing the list). Vertical or horizontal list, it is your choice.
6. For vertical lists (a list of macro variables), be sure to save the count of the number of elements in the list.

As you make the conversion, be sure that you think about each of the above elements.

### 11.1.5 Controlling the Program with Data

One of the hallmarks of a dynamic program is that it self-adjusts according to the data that it is using. The data or information source can take on many forms and can be used in a number of different ways (Section 11.2 discusses information sources in more detail).

A simple, but rather common, example of a program that dynamically adjusts according to some aspect of the data would be one that builds logical branching based on the value of a macro. Available only within a macro, the use of the macro %DO block is often combined with the %IF-%THEN/%ELSE statements. Program 5.3.1a demonstrates this type of branch. In Program 11.1.5 the number of observations in a data set has been stored in the macro variable &NOBS. This value is then used to determine whether the data set is to be summarized or printed.

#### Program 11.1.5: Combining a %IF-%THEN/%ELSE with %DO

```
%macro showdat;
...code not shown...
%if &nobs ge 10 %then %do;
  proc means data=statdata mean n stderr;
  var ht wt;
  title "Analysis Data - &nobs Observations";
  run;
%end;
%else %if &nobs gt 0 %then %do;
  proc print data=statdata;
  var subject ht wt;
  title "Data NOT Summarized";
  run;
%end;
%else %put Data Set STATDATA is empty;
...code not shown...
%mend showdat;
```

Although this author does not generally recommend the technique, branching can also include the use of the %GOTO statement, as is shown in Section 8.2.2.

**SEE ALSO:** Yindra (1997) has an example of a macro that also branches based on the number of observations in a data set.

### 11.1.6 List Processing Basics

Common to most dynamic macros is the use of indirect macro variable references, and these tend to be stored as lists of values. There are two basic forms of lists, vertical and horizontal. Horizontal lists are usually parsed with the %SCAN or %QSCAN function.

Vertical lists are discussed in more detail in Section 11.3, and horizontal lists are discussed in Section 11.4.

You can sometimes process a list of values without actually building the list. The CALL EXECUTE routine can be used from within a DATA step to execute macro code (see Section 11.5). The macro language can sometimes be avoided altogether (*emphasis on avoided* as this is an avoidance technique) by using a DATA step to write the code that would otherwise be written by the macro (see Section 11.6).

Regardless of which of these techniques is most appropriate for your situation, you should understand the basics of each of these approaches, not only so that you can pick the one best suitable for your problem, but also so that you can follow someone else's code.

**SEE ALSO:** Fehd and Carpenter (2007) discuss these four approaches to list processing in the macro language.

### 11.1.7 Iterative Step Execution

Once a list has been created, you will need to step through the list. Generally this will be accomplished using some type of %DO loop. When the number of items in the list is known, both vertical and horizontal lists are usually processed using an iterative %DO loop. You can step through a horizontal list with an unknown number of items by using either a %DO %WHILE or a %DO %UNTIL loop. Sections 11.3 and 11.4 both have examples that use these kinds of %DO loops.

### 11.1.8 Building Statements

In dynamic programs it is not at all unusual to need to create or write SAS statements. You can use the macro language in a number of ways to build individual SAS statements. Macro logic based on the %IF statement is often combined with the various forms of the %DO loop to create full statements or just portions of the statements. Examples in Sections 5.2.2, 5.3.1, and 5.3.2 introduce some of these techniques, while most of the examples in the remainder of this chapter directly address issues that are related to building statements.

There is a difference between substituting entire statements and writing the appropriate statement directly. This difference is emphasized in the program fragments that follow in this section. In the following example, the SET statement is defined conditionally by substitution. Either the data set WORK.COND or WORK.GENERAL will be used depending on the value in &COND:

```
%macro tryit;
  data new;
    %if &cond=YES %then %str(set cond);
    %else %str(set general);
  run;
%mend tryit;
```

It is not necessary to create and substitute the entire statement within the macro text. In the previous example, the SET statement could also have been written without using the %STR function:

```
%macro tryit;
data new;
set
  %if &cond=YES %then cond; ①
  %else general;
; ②
run;
%mend tryit;
```

- ① The %IF-%THEN/%ELSE is used inside what will become the SET statement.
- ② The SET statement is terminated with this semicolon.

When there is more than one item (when a list is involved), you can also use the %DO loop very effectively to build statements. The DATA step shown in the following example (which is also a portion of a macro) is a part of the solution to Exercise 6.3 (Section 6.6, Exercise 3). Assume that in a previous step a series of macro variables (&REG1, &REG2, and so on) had been created to hold the unique values of the character variable REGION (the values in this case are '1', '2', '3', and so on). This DATA step will be used to create a separate data set for each unique value of REGION. For example, all observations with REGION='1' will be written to the data set R1:

```
%macro RegionRpt(dsn=macro3.clinics);
. . . code not shown . . .
data %do i = 1 %to &total; r&reg&i %end;; ③
  set clinics;
  /* Build the &total output statements;
  %do i = 1 %to &total; ④
    %if &i>1 %then else; ⑤
    if region="&&reg&i" then output r&reg&i;
  %end;
  run;
. . . code not shown . . .
%mend regionrpt;
```

- ③ The %DO loop builds the DATA statement by naming all of the data sets to be created. The resulting DATA statement will look something like this:

```
data r1 r2 r3;
```

- ④ This loop creates the statements used to OUTPUT each observation to the appropriate data set and will be executed once for each unique value of REGION.
- ⑤ For three regions, the resolved statements might be as follows:

```
if region="1" then output r1;
else
  if region="2" then output r2;
else
  if region="3" then output r3;
```

**MORE INFORMATION:** The DATA statement and a series of IF statements are built in Program 11.2.6. Program 12.1.1 builds a SELECT statement from the “inside out”.

**SEE ALSO:** You can find an additional example of a dynamically built SET statement in Tassoni, Chen, and Chu (1997).

Smith (1997) includes several examples of dynamic code building. These include the use of %DO loops in PROC FORMATS and in AXIS statements.

## 11.2 Information Sources

The information that is used to control or guide the dynamic macro can be found in a very wide variety of forms and places. Remember that this information will be used to build one or more macro variables, so first visualize the information that is needed in the macro variable(s) and then visualize how you can get this information into a SAS data set or a similar source. Once in a SAS data set, it is simply a matter of using the SYMPUTX routine or an SQL INTO clause to build the needed macro variables.

A number of examples throughout this book show various ways of creating or using control information from which you can build the macro variables. The following table suggests some sources for control information as well as locations within this book with examples:

**Table 11.2: More Information on Information Sources**

Type of Information Source	Initial Discussion	Example Sections / Programs
SASHELP and Dictionary views	11.2.1 and 11.2.2	6.2.2, 6.6.1, 6.6.5, 8.1.1a, 8.2.1e, 8.3.2f, 9.2.1b, 12.1.1, 12.1.2a, 12.1.2b, 12.2.1a, 12.2.1b, 12.2.2, 12.3.1, 12.4.2a, 12.4.2c, 14.6.2
Automatic Macro Variables	11.2.3	2.6, 8.3
%SYSFUNC and DATA step functions	11.2.4	12.1.2d, 12.1.2f, 12.3.1, 12.3.2b, 12.3.3, 12.4.1, 12.4.2b, 12.4.2d, 12.4.3b, 12.4.7, 13.2.2, throughout chapters 11, 12, and 13
Results of operating system operations	11.2.5	12.1.2d, 12.1.2e, 13.1.1, 13.1.3
Data set and file metadata OUT= option with PROC CONTENTS	11.2.6	6.2.3, 12.1.2f, 12.4.2d 12.1.2c
The analysis data itself	11.2.7	6.4.1, 6.4.2, Q6.2, Q6.3, 12.3.2b
Control data sets built strictly to control macro operations	11.2.8	12.4.8
SET Statement options	11.2.9	13.3.2, 13.3.3b, 13.3.3c
SAS System options	11.2.4	12.3.1, 12.3.2a,

**SEE ALSO:** Hadden (2015) introduces a number of metadata sources.

### 11.2.1 Using SASHELP Views

A series of SAS views have been defined and stored under the SASHELP *libref*. These views contain valuable information about the current status of your system and operating environment. These views can be especially useful to the macro programmer because they can be used as a source for building macro variables.

Although the SASHELP views discussed below can also be used within an SQL step, a similar, though not as extensive, set of tables can be accessed, while in an SQL step, by using the *libref* DICTIONARY (see

Section 11.2.2). In terms of information content there is some overlap between the SQL DICTIONARY tables and the SASHELP views. When you are using SQL and you have a choice between the two types of tables, the DICTIONARY tables will often be a bit quicker.

Although they seem to be data sets in almost all aspects, views in fact contain no data. Instead, they contain the instructions needed to build the table when the view is requested. The availability of these views is very important to the dynamic macro programmer as they provide a great deal of information about the SAS environment during macro execution. This information can be translated into macro variables, which in turn can be used to drive the dynamic application. Some of the primary views available in SASHELP are shown in Table 11.2.1.

**Table 11.2.1: Selected SASHELP Views**

View Name	View Description
VCATALOG	List of objects and entries within a catalog
VCOLUMN	List of variables within data sets
VEXTFL	<i>Fileref</i> s and their assigned paths
VINDEX	Indexes including types
VMACRO	Each macro variable, its scope, and its value
VMEMBER	Names of all entities controlled by SAS (data sets and catalogs), their <i>librefs</i> , and their physical location
VOPTION	Current setting of each system option
VSCATLG	<i>Librefs</i> and names of all SAS catalogs
VSLIB	List of all <i>librefs</i> and their associated physical location (path)
VSTABLE	List of all data sets and their associated <i>libref</i>
VSTYLE	List of ODS styles and their location
VSVIEW	List of all VIEWS and their associated <i>libref</i> , similar to VVIEW
VTABLE	List of all data sets and views, including information like the number of observations and modification dates
VTITLE	Definition and number of TITLES and FOOTNOTES
VVIEW	List of defined views (includes the views in SASHELP)

The contents and attributes of these views are fairly straightforward. Try opening each one using the VIEWTABLE in the Display Manager. While it should be fairly obvious what each contains, it may not be immediately obvious as to how or when you will use any particular view in one of your macros. The important point is that you will now know what is available when you do need it.

The SASHELP views are used as you would any of the control data sets mentioned elsewhere in this chapter. Simply read the appropriate values into one or more macro variables and then use them in your program. Because they are views and not data sets, they are not physically constructed until requested. This means that they are always current.

Although the macro in Program 11.2.1 just does a PROC PRINT, we want its title to be the next available title. If we assume that we do not know what the last defined title number was, we will need to determine the next available title number dynamically. The view SASHELP.VTITLE contains the list of all currently defined titles and footnotes. It is read using an SQL step and the maximum title number currently in use is saved in a macro variable that is used later.

#### Program 11.2.1: Selecting the Next Available Title

```
%macro look(dsn);
%local maxn;
* Determine the next available title;
proc sql noprint; ①
  select max(number) ②
    into :maxn ③
    from sashelp.vtitle ④
      where type='T'; ⑤
quit;

title%eval(&maxn+1) ⑥ "Listing of &dsn";
proc print data=&dsn;
  run;
title%eval(&maxn+1); ⑦

%mend look;
```

- ① An SQL step is used to build the macro variable containing the largest title number.
- ② The MAX function collects the largest value of the variable NUMBER.
- ③ The maximum value of NUMBER is placed INTO the macro variable &MAXN.
- ④ SASHELP.VTITLE is used as the information source. This view has one observation per title or footnote.
- ⑤ Only lines for titles are included (footnotes have TYPE='F').
- ⑥ The title is assigned the next title number.
- ⑦ Like a good child, it cleans up after itself by resetting the new title after the PROC PRINT is complete.

**CAVEAT:** Although the maximum number of title lines is 10, this macro does not do any checking to make sure that an attempt to create more than 10 titles is not made.

**MORE INFORMATION:** The next available title is also determined in the example in Program 12.2.1. A series of catalog names are recovered using SASHELP.VSCATLG in Program 12.1.1. SASHELP.VSTABLE is used in Program 12.1.2a to retrieve a list of data sets.

**SEE ALSO:** Lafler (2010) introduces SASHELP views and DICTIONARY tables. An introductory discussion of metadata as well as the use of SASHELP.VCOLUMN is presented by Spicer (2003).

Davis (2001) describes the SASHELP views and the corresponding DICTIONARY tables used by SQL. Hamilton (1998) concentrates on the SQL DICTIONARY tables. Casas (2002) uses DICTIONARY.TABLES and DICTIONARY.COLUMNS to build macro variables from within an SQL step.

A list of macro variables is cleared using SASHELP.VMACRO and CALL EXECUTE in an example by Rhoads and Letourneau (2002).

The basics, along with a couple of caveats, for using VCOLUMN, VCATALOG, and VMEMBER are presented by Kelly (2000). VCOLUMN is also used by Goddard (2003) in an example that writes RTF files using ODS.

Troxell and Chen (2003) use SASHELP.VCATALG to determine whether a macro has been compiled. SASHELP.VMACRO is used by Yu and Huang (2003) in a macro that generates return codes.

## 11.2.2 Using SQL DICTIONARY Tables

As was shown in Section 11.2.1 PROC SQL has full access to the views in the SASHELP library. However, in order for SQL to more easily access information about its environment, SAS has also constructed a series of DICTIONARY tables. These tables are similar to the SASHELP views described in Section 11.2.1; however, they can only be accessed from within an SQL step.

The list of these DICTIONARY tables has been expanding with new releases of SAS. A partial list of some of the more commonly used tables is shown in Table 11.2.2.

**Table 11.2.2: Selected DICTIONARY Tables Used with PROC SQL**

Table Name	Table Description
CATALOGS	Lists catalogs, members, entries, and entry types
COLUMNS	Data set attributes
COLUMNS	Data set attributes
EXTFILES	External file information
INDEXES	Lists existing indexes
MACROS	Symbol table attributes; macro variables and their scopes
MEMBERS	SAS controlled data sets, catalogs, views, and so on.
OPTIONS	Current settings of system options
TABLES	Attributes of SAS tables (data sets)
TITLES	Current title settings and values
VIEWS	Attributes of SAS views

To be able to use these tables, you will need to know their column names. You can list the variable attributes (name, type, length, and label) by using the SQL DESCRIBE statement. The SQL step in Program 11.2.2a writes the attributes of the DICTIONARY.COLUMNS table to the SAS Log.

### Program 11.2.2a: Using DESCRIBE to View Table Attributes

```
proc sql ;
  describe table dictionary.columns;
  quit ;
```

Program 6.2.3a uses PROC CONTENTS to build a list of variable names. This step could be eliminated by using the DICTIONARY.COLUMNS table. Program 11.2.2b harvests the variable names from DICTIONARY.COLUMNS, which has one row per variable per data set per library.

### Program 11.2.2b: Using DICTIONARY.COLUMNS to Build a List of Variables

```
%macro varlist2(lib=,dsn=);
%local i;
* Collect the variable names;
proc sql noprint;
select distinct name into :varname1- ①
  from dictionary.columns ②
  where (libname=upcase("&lib") & ③
memname=upcase("&DSN"));
quit;
```

```
%do i = 1 %to &sqlobs; ④  
  %put &i &&varname&i;  
%end;  
%mend varlist2;  
  
%varlist2(lib=macro3,dsn=clinics)
```

- ❶ A vertical list of macro variables is created. Remember that starting in SAS 9.4 the upper bound of the list does not need to be specified as long as the initial element is followed by a dash.
- ❷ DICTIONARY.COLUMNS is used as the source table for the list of variable attributes.
- ❸ The WHERE is used to subset for the library and data set of interest. The UPCASE function is used as the values in DICTIONARY.COLUMNS are stored in uppercase.
- ❹ An iterative %DO is used to step through the list of macro variables. In a practical application the list would be used inside this loop.

**MORE INFORMATION:** The SQL DICTIONARY tables are similar to the SASHELP views discussed in Section 11.2.1. A list of data set names is built using DICTIONARY.TABLES in Program 12.1.2b.

**SEE ALSO:** Whitlock (2000b) provides a nice introduction to the SQL dictionary tables, and a brief overview of the DICTIONARY tables is given by Bruchecker (1998b).

Tomb and Carter (2001) and LeBouton and Rice (2000) both use DICTIONARY.TABLES to build a list of like-named data sets.

Lists of variables that are to be renamed are built by Ravi (2003) with the use of DICTIONARY.TABLES and DICTIONARY.COLUMNS.

Mao (2003) uses DICTIONARY.TABLES to establish a list of data sets from a specified library as well as creating a list of macro variables using an SQL step.

DICTIONARY.CATALOGS is used by Whitlock (2001b) to establish a list of formats.

DICTIONARY.EXTFILES is used by Gunshenan (2003) to build a list of external files.

### 11.2.3 Automatic Macro Variables

Automatic macro variables were first introduced in Section 2.6 and further discussed in Section 8.3. These macro variables are established and maintained by SAS. You can view the current values of the automatic macro variables by using a %PUT statement.

```
%put _automatic_;
```

These macro variables store information about current SAS settings, current date and time values (see Section 2.6.1), server and operating system information (see Section 2.6.5), and error and return codes (see Section 8.3.2).

### 11.2.4 %SYSFUNC and DATA Step Functions

With over 450 DATA step functions, it is very difficult for the DATA step programmer to be familiar with all of them. Quite a few of these functions will only rarely, if ever, be used in the DATA step; however, because of the %SYSCALL statement and the %SYSFUNC macro function, many of those same functions are golden for the macro programmer. As a macro programmer you *must* become familiar with these functions that are otherwise only rarely useful in the DATA step. The use of %SYSCALL and %SYSFUNC are introduced in Section 7.4.

Table 11.2.4 shows three categories of DATA step functions that are not usually used in the DATA step, but are extremely valuable to the macro programmer, especially macro programmers who are writing dynamic macros.

**Table 11.2.4: Selected Categories of Functions Used in Dynamic Macros**

Category	Description	Example Sections
External Files	Work with files other than SAS files on your operating system	7.4.2d, 8.2.4, 13.2.2a, 13.2.2b
SAS File I/O	Access data set information	11.2.6
Special	Eclectic collection of functions	7.4.2b, 7.4.3, 8.3.1b

The GETOPTION function is an example of a function in the special category shown in Table 11.2.4. It can be used to retrieve the current setting of a system option. Very often in dynamic programs, we want to change the value of a system option and at the end of our program to reset it to its original value. In the macro %HOLDOPT shown in Program 11.2.4 the value of a system option is retrieved and saved so that it can be used to reset the option later.

#### Program 11.2.4: Retrieving the Value of a System Option

```
*****
/* Macro: HoldOpt */
/* Programmer: Pete Lund */
/* Date: September 2000 */
/* Purpose: Holds the value of a SAS option in a macro */
/* variable. The value can then be used to reset options */
/* to a current value if changed. */
*/
/* Parameters: */
/* OptName - the name of the option to check */
/* OptValue - the name of the macro variable that will */
/* hold the current value of the option */
/* The default name is made up of the word */
/* "Hold" and the option name. For */
/* example, if OptName=Notes, OptValue */
/* would equal HoldNotes */
/* Display - Display current value to the log (Y/N) */
/* The default is N */
/*
%macro HoldOpt(
    OptName=,      /* Option to hold value */ ①
    OptValue=XX,   /* Macro var name to hold value*/ ②
    Display=N);   /* Display value to the log */

@if %substr(&sysver,1,1) eq 6 ③
    and ((%length(&OptName) gt 4 and &OptValue=XX)
        or %length(&OptValue) gt 4) %then %do;
    %put WARNING: Default variable name of Hold&OptName is too long for
V&sysver..;
    %put WARNING: Please specify a shorter name with the OptValue= macro
parameter.;
    %goto Quit;
%end;
@if &OptValue eq XX %then %let OptValue = Hold&OptName; ④
%global &OptValue; ⑤
%let &OptValue = %sysfunc(getoption(&OptName)); ⑥
@if &Display eq Y %then ⑦
```

```
%put The current value of &OptName is &&&OptValue;
%Quit:
%mend;
```

Source: %HOLDOPT, Pete Lund, Looking Glass Analytics

- ➊ The option name of interest is stored in &OPTNAME.
- ➋ &OPTVALUE will hold the name of the macro variable that will contain the system option value.
- ➌ For Version 6 and before, which does not allow names of lengths greater than 8, the default naming of the macro variable may be limiting. Check the version of SAS using &SYSVER.
- ➍ If a name for the macro variable is not otherwise supplied, the name will be the option name preceded by HOLD.
- ➎ Make the macro variable global.
- ➏ The GETOPTION function is used to retrieve the system option whose name is stored in &OPTNAME.
- ➐ The DISPLAY parameter allows the user to show or display the value of the macro variables generated by the macro %HOLDOPT. This macro assumes that the value of &DISPLAY will be in uppercase. The %UPCASE function could have been used to force this to be true.

```
%if %upcase(&Display) eq Y %then
```

The following three calls to %HOLDOPT will save the values of the DATE, OBS, and LINESIZE options:

```
%HoldOpt(OptName=date)
  %put &=holddate;
%HoldOpt(OptName=obs, display=Y)
%HoldOpt(OptName=ls, OptValue=myls)
  %put &=myls;
```

The SAS Log shows that the macro variables have been created:

```
59  %HoldOpt(OptName=date)
60    %put &=holddate;
HOLDDATE=DATE
61  %HoldOpt(OptName=obs, display=Y)
The current value of obs is 9223372036854775807
62  %HoldOpt(OptName=ls, OptValue=myls)
63    %put &=myls;
MYLS=96
```

If these system options are changed in the user's program, they can be reset to their original values with the following OPTIONS statement:

```
options &holddate obs=&holdobs ls=&myls;
```

This macro requires the user to know whether a given option is a keyword-style option, such as `ls=96`, or a toggle option such as `DATE` or `NODATE`. When the second argument of the GETOPTION function is `KEYWORD`, the function returns the value in keyword format (as needed). In %HOLDOPT the code at ➏ could be rewritten to use the `KEYWORD` option.

```
%let &OptValue = %sysfunc(getoption(&OptName, keyword));
```

If this change is made in %HOLDOPT, one could reset the DATE, OBS, and LS options merely by specifying the macro variable.

```
options &holddate &holdobs &myls;
```

**MORE INFORMATION:** The GETOPTION function is also used in %FMTSRCH in Program 12.3.2a. The physical location of a library is returned using the PATHNAME function in Program 12.1.2d.

**SEE ALSO:** The %HOLDOPT macro is discussed further in Lund (2001a and 2003b). Hughes (2016a) uses the GETOPTION function to examine computer performance and memory utilization. Glass and Haden (2016) obtains information from the SYSIN option with the GETOPTION function.

## 11.2.5 Retrieving Operating System Information

Operating system information is often stored in environmental variables, which you can think of as the operating system's macro variables. There is more than one way for you to access this information. In Section 8.1.1 there are several examples that demonstrate the retrieval of information stored in environmental variables. These include the use of the %SYSGET function and the %SYSEXEC statement, which are discussed further in Section 8.2.4.

## 11.2.6 Using Data Set Metadata

Every SAS data set stores information about the data set as a part of the data set itself. Most, but not all, of this metadata is surfaced when you use PROC CONTENTS. This metadata can be very useful to the macro programmer, and there are a number of ways to retrieve it dynamically.

- PROC CONTENTS with the OUT= option (see Section 6.2.3a)
- SASHELP.VCOLUMNS (see Section 11.2.1)
- DICTIONARY.COLUMNS (see Section 11.2.2)
- DATA step I/O functions (see Section 11.2.6)

### DATA Step I/O Functions

The macro function %OBSCNT in Program 11.2.6a resolves to the number of observations in the data set named in the macro parameter. The number of observations is retrieved from the data set's metadata using DATA step I/O functions.

#### Program 11.2.6a: Retrieving the Number of Observations from the Data Set's Metadata

```
%macro obscnt(dsn); ①
%local dsnid nobs;
%let nobs=.;

/* Open the data set of interest;
%let dsnid = %sysfunc(open(&dsn)); ②
/* If the open was successful get the;
/* number of observations and CLOSE &dsn;
%if &dsnid %then %do; ③
    %let nobs=%sysfunc(attrn(&dsnid,nlobs)); ④
    %let dsnid =%sysfunc(close(&dsnid)); ⑤
%end;
%else %do; ⑥
    %put Unable to open &dsn - %sysfunc(sysmsg());
%end;

/* Return the number of observations; ⑦
&nobs ⑧
%mend obscnt;
```

- ① The user passes the name of the data set into the macro using &DSN.
- ② The selected data set is opened using the %SYSFUNC and OPEN functions. When opened, it is assigned an identification number, which is stored in &DSNID.

- ③ If the data set was found and was opened successfully, the macro variable &DSNID will be greater than 0. The %IF expression will, therefore, be true.
- ④ The ATTRN function is used to determine the number of observations in the data set. In this case, the NLOBS option returns the number of observations, excluding those marked for deletion. The ATTRN function can be used to make a number of queries on the data set once it is opened. These include password and indexing information, as well as the number of variables and the status of active WHERE clauses.
- ⑤ The data set should be closed after retrieving the desired information.
- ⑥ When the OPEN is unsuccessful, you might want to write a message to the LOG. The SYSMSG( ) function returns the reason the OPEN failed.
- ⑦ This must be either a macro comment or a PL1 (block style - /\* . . . \*/) comment. An asterisk-style comment would result in the comment itself being written as text along with the number of observations, and this would result in syntax errors if the macro is used as it is in the following %PUT statement. See Section 5.4.1 for more on macro comments.
- ⑧ Because this is the only text in the macro that is not part of a macro programming statement, the resolved value of &NOBS will, in effect, be returned to the calling program. Its value will be a period (.) if the data set was not opened successfully.

As a macro function, %OBSCNT can be used in a number of ways including in a %PUT. Here the SAS Log shows the outcome of two %PUT statements. The first is successful and the second is for a data set that does not exist.

```
111  %put There are %obscnt(sashelp.shoes) observations in sashelp.shoes;
      There are 395 observations in sashelp.shoes
112  %put There are %obscnt(sashelp.shss) observations in sashelp.shss;
      Unable to open sashelp.shss - ERROR: File SASHELP.SHSS.DATA does not exist.
      There are . observations in sashelp.shss
```

There are a great many data set attributes that can be retrieved using these DATA step functions. Some that are in examples in this book include:

**Table 11.2.6: DATA Step I/O Function Examples**

DATA Step Function	Sections/Programs
OPEN and CLOSE	In each of the following
ATTRN with NLOBS	9.2.2d, 11.2.6a
ATTRN with NVAR	12.4.1, 12.4.2d, 12.4.2e
FETCH	9.2.1b
FETCHOBS	9.2.2d
SET	9.2.2d
VARNAME	12.4.1, 12.4.2d, 12.4.2e
VARNUM	12.4.6

The use of these functions is very fast and will tend to be the fastest method for retrieving this type of information.

### PROC CONTENTS with the OUT= Option

When PROC CONTENTS is used with the OUT= option, a data set is generated that contains much of the metadata stored in the data set. The new data set will have one observation per variable.

**Program 11.2.6b: Using PROC CONTENTS to Access Metadata**

```
%macro dsncnt(lib);
proc contents data=&lib.._all_ ⑨
  out=cont
  noprint;
run;
data _null_;
  set cont;
  by memname;
  if first.memname then call symputx(memname,nobs,'1'); ⑩
run;
%put _local_; ⑪
%mend dsncnt;
%dsncnt(macro3)
```

- ⑨ Use of the \_ALL\_ keyword allows CONTENTS to access the metadata for all data sets in the library. The metadata is returned in the data set named in the OUT= option.
- ⑩ The number of observations (NOBS) is written into a macro variable with the same name as the data set. For the data set MACRO3.CLINICS, which has 80 observations, &CLINICS will contain the value 80.
- ⑪ Show the macro variables and their values. The %PUT generates the following in the SAS Log.

```
DSNCNT BIOMASS 48
DSNCNT CLINICS 80
DSNCNT DBDIR 3
DSNCNT DEMOG 3
DSNCNT FLDDIR 4
DSNCNT LIB macro3
DSNCNT MEDHIS 4
DSNCNT PHYSEXAM 4
DSNCNT VARDIR 13
```

**MORE INFORMATION:** Similar DATA step functions can be used to access directory and file metadata. See Program 12.1.2f, which builds a list of data sets using file metadata functions. The %SYSGET function is used in Section 8.1.1 to access system environmental variables, and %SYSEEXEC statement is used in Section 8.2.4 to retrieve directory information.

**SEE ALSO:** Hamilton (2001) discusses various problems, including errors returned by the ATTRN function, associated with capturing the number of observations in a data set. He also introduces and discusses the macro %MTANYOBS in detail.

The ATTRN function is also used by Zirbel (2002) in a macro that compares two data sets.

The ATTRN function is used to get several data set attributes, including the number of observations, in Allen (2003).

Yindra (1997) has an example of a macro that makes a decision branch based on the number of observations in the data set.

The %OBSCNT macro shown in Program 11.2.6 was originally adapted from a similar macro in *SAS Macro Language: Reference, First Edition* (p. 242), and another example can be found in *SAS 9.4 Macro Language: Reference, Fourth Edition* (Chapter 11, Example 5). Both examples use ATTRN to retrieve the number of variables as well as the number of observations.

Kretzman (1992) discusses a similar macro. The related functions, ATTRC and VARNAME, are used by Bramley (2001). Conley (2000) uses the ATTRN, VARNAME, and VARLABEL functions to retrieve and use variable labels

### 11.2.7 Using Data Tables to Control a Process

Sometimes the analysis data itself can be used to control the process (see Sections 6.4.1 and 6.5.2). Very often using the data this way requires that the data are read more than once. The first pass of the data is used to build a macro list (either vertical or horizontal depending on your needs). And then this list is used to process the analysis data.

In Program 11.2.7a the macro %BREAKUP is used to create a series of data sets, one for each level of a classification variable. A vertical list of macro variables is created, with one macro variable for each unique level of the classification variable.

#### Program 11.2.7a: Using the Analysis Data to Control the Process

```
%macro breakup(dsn=,classvar=);
%local classcnt i;
proc sql noprint;
    select distinct &classvar ①
        into :cval1- ②
        from &dsn;
    %let classcnt=&sqllobs; ③
    quit;

data
    %do i=1 %to &classcnt;
        class_&&cval&i ④
    %end;
    ;
    set &dsn;
    %do i=1 %to &classcnt; ⑤
        %if &i>1 %then else; ⑥
        if &classvar = "&&cval&i" then output class_&&cval&i; ⑦
    %end;
    run;

%mend breakup;
```

- ① The unique levels of the classification variable (&CLASSVAR) are determined.
- ② The values are written into a numbered list of macro variables named &CVAL1, &CVAL2, and so on.
- ③ The total number of classification levels is saved in the macro variable &CLASSCNT.
- ④ The DATA statement is created by listing the names of the new data sets. Each data set will include the level of the classification variable, which is stored in &&CVAL&I. For the variable SEX the levels might be 'F' and 'M', resulting in data sets named CLASS\_F and CLASS\_M.
- ⑤ A %DO loop is used to create a series of conditionally executed OUTPUT statements.
- ⑥ Other than for the first IF statement, we would like each IF statement to be preceded by an ELSE.
- ⑦ Each observation will be written to the appropriate data set.

Because the levels of the classification variable are determined dynamically, the macro will work for any character classification variable for any data set, as long as the classification variable does not include any characters that would be considered to be invalid in a data set name. Invalid characters could be removed using the COMPRESS function such as: compress(&splitvar,, 'kn').

A portion of the SAS Log shows that the variable SEX can be used to create two new data sets: one for each level of the variable.

```
193 %breakup(dsn=macro3.clinics, classvar=sex)

NOTE: There were 80 observations read from the data set MACRO3.CLINICS.
NOTE: The data set WORK.CLASS_F has 32 observations and 20 variables.
NOTE: The data set WORK.CLASS_M has 48 observations and 20 variables.
```

In Program 11.2.7a the assumption is made that the value of the classification variable can be used in the data set name. If the value contains unacceptable characters, the DATA step will fail. In Program 11.2.7b the compress function is added whenever the data set is named (❸ and ❹) in Program 11.2.7a.

#### Program 11.2.7b: Using COMPRESS to Remove Unacceptable Characters from the Data Set Name

```
%macro breakup(dsn=,classvar=);
%local classcnt i;
proc sql noprint;
  select distinct &classvar
    into :cval1-
      from &dsn;
%let classcnt=&sqlobs;
quit;

data
  %do i=1 %to &classcnt;
    class_%sysfunc(compress(&&cval&i,,kn)) ❸
  %end;
  ;
  set &dsn;
  %do i=1 %to &classcnt;
    %if &i>1 %then else;
    if &classvar = "&&cval&i" then
      output class_%sysfunc(compress(&&cval&i,,kn)); ❹
  %end;
  run;

%mend breakup;
%breakup(dsn=sashelp.cars, classvar=make)
```

Source: Solution suggested by Marty Hultgren, SAS Institute

- ❸ The 'kn' modifiers in the third argument of the COMPRESS function causes it to remove all characters other than digits, letters, and underscores.

**MORE INFORMATION:** A data set is broken up vertically in Program 12.4.1.

**SEE ALSO:** Carpenter and Callahan (1988) discuss two similar macros, %BREAKUP and %SPLITUP that you can use to control program flow and output organization. Another example of this programming technique can be found in Wobus and Gober (1997). Ferriola (2016) introduces and discusses the use of control tables.

## 11.2.8 Creating and Using Control Files

A control file is a data set or something that looks like a data set, which can be used to control the execution of some process. The advantage to the programmer is that changes to the process are implemented through changes to the control file not through changes to the program.

The example shown in this section is based on a survival analysis study. The analysis was all macro driven SAS code, and there were nearly 180 different survival analysis models in the study. The analysts did not want to have to change the macro code for each analysis; consequently, the specification for each model was made as a row in a CSV file using Excel. A portion of the control file is shown in Table 11.2.8. In the actual study there were close to 180 rows and 40 columns in the control file.

**Table 11.2.8: Control File for a Survival Analysis Study**

	A	B	C	D	E	F	G
1	number	code	analysistype	refpop	censorrule	eventcodes	arvregimen
2	1.1	ph38_6	prelim	38	6	132	na
3	1.2	ph38_6	prelim	38	6	132	nn
4	1.3	ph38_6	prelim	38	6	132	pi
5	1.4	ph38_6	prelim	38	6	132	nannpi
6	1.5	ph38_6	prelim	38	6	132	xxxxx
7	2.1	ph38_6	prelim	38	6	249	na
8	2.2	ph38_6	prelim	38	6	249	nn
9	2.3	ph38_6	prelim	38	6	249	pi

The analysis is driven by the %SURVIVAL macro and each row in the control file becomes a separate call to this macro. Assuming that the column headers of this file roughly correspond to parameter names, the second row (analysis model number 1.2) would eventually result in the following macro call:

```
%survival(anumb=1.2,acode=ph38_6,atype=prelim,refpop=38,censor=6,
           eventcode=132,arvreg=nn)
```

Of course we do not want just one macro call, we want a series of macro calls, and we do not want to type each one separately.

The first step is to move the information in the control file into macro variables. Although we could use a PROC IMPORT step to first create a SAS data set, in this case we only want to create the macro variables so we can directly import the information and create the macro variables in a single DATA step. Program 11.2.8a contains the DATA step that reads the CSV file and creates the macro variable lists (in this example, a series of vertical lists are created). See Section 11.3 for more on the construction and use of vertical macro variable lists.

#### Program 11.2.8a: Building Macro Variable Lists from a CSV Control File

```
filename xlscntrl "&path\data\control.csv"; ①
data _null_;
  infile xlscntrl dlm=',' truncover firstobs=2; ②
  length number code analysistype refpop
         censorrule eventcode arvregimen $8;
  input number $ code $ analysistype $ refpop $ ③
        censorrule $ eventcode $ arvregimen $;
  i+1; ④
  ii=left(put(i,4.)); ⑤
  call symputx('anumb'||ii,trim(number),'1'); ⑥
  call symputx('acode'||ii,trim(code),'1');
  call symputx('atype'||ii,trim(analysistype),'1');
  call symputx('refpop'||ii,trim(refpop),'1');
  if censorrule ne '-' then ⑦
    call symputx('censor'||ii,trim(censorrule),'1');
  else call symputx('censor'||ii,' ','1');
  call symputx('eventcode'||ii,trim(eventcode),'1');
  call symputx('arvreg'||ii,trim(arvregimen),'1');
  call symputx('count',ii,'1'); ⑧
run;
```

- ① The location of the control file is specified using a FILENAME statement.
- ② The first record of the control file contains column headers, which are skipped using the FIRSTOBS option.
- ③ Each record is read into the PDV. For ease of handling, each variable is read into a character string.

- ④ Each row is counted.
- ⑤ The counter is converted to a left-justified string so that it can be appended onto the macro variable names. This step could have been avoided through the use of the CAT or CATT function (see Program 11.1.2a).
- ⑥ Each element of each macro variable list is created using the SYMPUTX routine. Notice that the third argument is set to 'l', which causes each of these macro variables to be added to the local symbol table.
- ⑦ Because we are using the DATA step, we can also apply logic and conditional assignments.
- ⑧ The total number of elements in the lists is saved in the local macro variable &COUNT.

Once the macro variable lists have been created, they can be used to build the calls to the macro %SURVIVAL. A macro %DO loop is used to step through and used the lists. Program 11.2.8b shows the %DO loop.

#### **Program 11.2.8b: Using the Macro Variable Lists to Build Calls to the %SURVIVAL Macro**

```
%do k = 1 %to &count;
  %survival(anumb=&&anumb&k,
             acode=&&acode&k,
             atype=&&atype&k,
             refpopcode=&&refpop&k,
             activevar=&&censor&k,
             regimen=&&arvreg&k,
             eventcode=&&eventcode&k)
%end;
```

Since a macro %DO loop cannot exist in open code (although this may change in future releases of SAS), the DATA step shown in Program 11.2.8a and the %DO shown in Program 11.2.8b must exist within the context of another macro. Program 11.2.8c shows how these code steps are used together.

#### **Program 11.2.8c: Building and Using Lists within a Macro**

```
%macro BuildSurv;
%local k;

* 11.2.8a;
data _null_;
  . . . . code not shown . . . .

* 11.2.8b;
%do k = 1 %to &count;
  . . . . code not shown . . . .

%mend buildsurv;
%buildsurv
```

**MORE INFORMATION:** Section 11.3 goes into more detail on the creation and use of vertical macro variable lists.

**SEE ALSO:** Ake and Carpenter (2002 and 2003) each go into more detail on the specific survival analysis example used in this section.

---

## **11.2.9 Using SET Statement Options**

When you do not need to create a macro function, and the DATA step can be used, the NOBS= option on the SET statement can be used to determine the number of observations in the SAS data set.

A DATA \_NULL\_ step is used with a SET statement in Program 11.2.9 to place the number of observations into a macro variable through the use of the NOBS= SET statement option.

**Program 11.2.9: Using SET Statement Options**

```
%macro numobs(dsn=);
  data _null_;
    call symputx('numobs', dsnobs, ①'g'); ②
    stop; ③
    set &dsn nobs=dsnobs; ④
    run;
%mend numobs;
```

- ① The macro variable that will hold the number of observations is added to the global symbol table.
- ② CALL SYMPUTX is used to create the macro variable NUMOBS. The number of observations in the data set is assigned to the temporary data set variable DSNOBS during the compilation of the DATA step.
- ③ The STOP statement prevents execution of the SET statement; consequently, no observations are read from &DSN.
- ④ The NOBS= option is used to load the number of observations into the temporary variable DSNOBS.

**CAVEAT:** This technique, which is widely used, generally works correctly. If, however, the data set contains deleted observations, the NOBS= option will return the incorrect number. Since deleted observations are usually the result of deleting observations while in an interactive session (with, for example, PROC FSEDIT), this is generally not a major issue.

**SEE ALSO:** The %SPLIT macro demonstrated by Gerlach and Misra (2002) also uses the NOBS= option on the SET statement.

**11.3 Using &&VAR&I Constructs as Vertical Macro Arrays**

While SAS does not formally define arrays in the macro language, the &&VAR&I macro variable form functions in a way that mimics a macro array, where the &I serves as the array subscript or index. This form is useful when a series of values need to be stored individually. The name of the *array*, “VAR” in this case, can of course be any appropriate name.

When an indirect reference is needed but there is only one element to be stored, an array is not necessary and the macro variable takes the form of &&&VAR. Unlike in the macro array, where the name of the array is known and the indirect reference is established with an index, in this form the name of the macro variable is itself unknown and is used to establish the indirect reference.

**MORE INFORMATION:** The creation of vertical macro variable lists was first introduced in Sections 6.1.4 and 6.2.3. Numerous examples of the use of this type of list occur throughout the book. Programs 6.4.1b, 11.2.2b, and 11.2.7 create and use a vertical list, while the examples in Section 11.2.8 discuss the use of a vertical list to process information in a control file.

**SEE ALSO:** Fehd (1997a, 1997b, and 1997c) and Blood (1992) include macros and discussions on the use of macro arrays in dynamic coding situations.

Indirect array references in the form of &&&VAR&J are used by Kunselman (2001) in a SAS/IntrNet example.

Moors (2016) uses a %DO to step through a list of macro variables.

### 11.3.1 Creating the List of Macro Variables

The list of macro variables can be created using either the SYMPUTX routine in the DATA step (see Section 6.1.4) or through the use of the INTO clause in a PROC SQL step (see Section 6.2.2).

The data set MACRO3.DBDIR (introduced in Section 6.5.1 and shown here in Table 11.3.1) contains a list of the names of data sets in a study. Program 11.3.1 builds a list of macro variables using this data set and PROC SQL.

**Table 11.3.1: View of the Data Set MACRO3.DBDIR**

VIEWTABLE: Macro3.Dbdir		
	dsn	Keyvar
1	DEMOG	SUBJECT
2	MEDHIS	SUBJECT MEDHISNO SEQNO
3	PHYSEXAM	SUBJECT VISIT SEQNO

**Program 11.3.1: Creating a List of Macro Variables Using SQL**

```
proc sql noprint;
  select dsn
    into :dsn1 - ❶
      from macro3.dbdir;
  %let dsncnt = &sqllobs; ❷
  quit;
```

- ❶ A series of macro variables of the form &DSN1, &DSN2, and so on will be created. Starting in SAS 9.4 the upper bound of the list does not need to be specified as long as the initial element is followed by a dash.
- ❷ The number of elements in the list is automatically collected in &SQLOBS and that value is written to the macro variable &DSNCNT.

### 11.3.2 Resolving &&VAR&i

Crucial to the successful use of macro variables of the form &&VAR&I is an understanding of how these macro variables are resolved and used. Because there are adjacent ampersands (&&), the resolution process is necessarily a two-step process. The first resolution pass results in a macro reference that itself must be resolved in a second pass.

In Program 11.3.2 the macro %DO loop in %LISTDSN will execute &DSNCNT times. Inside the %DO the macro variable list &&DSN&I is specified in a %PUT.

**Program 11.3.2: Resolving Members of a Macro Variable List**

```
%macro listdsn;
%do i = 1 %to &dsncnt;
  %put &=i  &&dsn&i;
%end;
%mend listdsn;
%listdsn
```

When & DSNCNT is 3, and the data set MACRO3.DBDIR was used as in Section 11.3.1, the loop creates the macro variable list of three macro variables:

&DSN1 &DSN2 &DSN3
-------------------

On the second pass these in turn further resolve to:

```
DEMOG MEDHIS PHYSEXAM
```

The process of the resolution of macro variables can be viewed in the SAS Log by using the SYMBOLGEN system option. After the first two iterations of the %DO loop, the SAS Log shows how the macro variables in the %PUT are resolved.

#### Program 11.3.2 (SAS Log): Showing the First Two Iterations of the %DO loop

```
81  %listdsn
SYMBOLGEN: Macro variable DSNCNT resolves to 4
SYMBOLGEN: Macro variable I resolves to 1
SYMBOLGEN: && resolves to &.
SYMBOLGEN: Macro variable I resolves to 1
SYMBOLGEN: Macro variable DSN1 resolves to DEMOG
I=1 DEMOG
SYMBOLGEN: Macro variable I resolves to 2
SYMBOLGEN: && resolves to &.
SYMBOLGEN: Macro variable I resolves to 2
SYMBOLGEN: Macro variable DSN2 resolves to MEDHIS
I=2 MEDHIS
```

### 11.3.3 Stepping through a List of Data Sets

We now have the ability to step through a list of data sets. The following %DO loop resides in a macro that processes each data set in the study:

```
%do jj = 1 %to &dsncnt;
  proc fsedit data=livedb.&&dsn&jj ①
    screen=appls.descrn.&&dsn&jj...screen; ②
  run;
%end;
```

- ① The PROC FSEDIT is executed for each data set
- ② The appropriate customized SCREEN (which also named using the associated data set name) is also selected.

Notice the use of the three decimal points ②. More than one is required as the SAS interpreter will see them as part of the macro variable name when they immediately follow a macro variable. The double ampersand requires two scanning passes and each pass absorbs one of the dots.

## 11.4 Horizontal Lists

Unlike the vertical lists discussed in Section 11.3, which use a series of numbered macro variables, horizontal lists use a single macro variable with a number of words. The index to the horizontal list is essentially the word number within the list. %DO loops that step through the list use the word number along with the %SCAN function to parse the list.

**MORE INFORMATION:** Building and using a horizontal list of values is introduced in Section 6.2.2 and further described in Program 11.1.1d.

**SEE ALSO:** Vincent and Ortiz (2016) build and use horizontal lists to work with survey data. Burnette-Isaacs (2016b) creates a horizontal list of data set names using an SQL INTO operator.

### 11.4.1 Creating Horizontal Lists

Horizontal lists are most easily created in a PROC SQL step. They can also be created in a DATA step; however, there are physical limitations that do not apply when using SQL. In Program 11.4.1 distinct values of the variable DSN are read from the data set MACRO3.DBDIR (see Table 11.3.1) and written into the macro variable &DSNLIST.

#### Program 11.4.1: Building a List of Space-Separated Values in a Single Macro Variable

```
proc sql noprint;
  select distinct dsn ①
    into :dsnlist ②
    separated by ' ' ③
      from macro3.dbdir; ④
  %let dsncnt = &sqllobs; ⑤
  quit;

%put &=dsncnt;
%put &=dsnlist;
```

- ① The unique values of DSN are added to the macro variable.
- ② The INTO clause is used to designate the macro variable (&DSNLIST) that is to receive the list of names.
- ③ The SEPARATED BY keyword allows multiple values to be added to &DSNLIST.
- ④ The list of values comes from the data set MACRO3.DBDIR (see Section 11.3.1 for more information on this data table).
- ⑤ The number of items in the list is recovered from the automatic macro variable &SQLOBS and saved in &DSNCNT.

The SAS Log for Program 11.4.1 shows the two macro variables created in this SQL step.

```
111 %put &=dsncnt;
DSNCNT=3
112 %put &=dsnlist;
DSNLIST=DEMOG MEDHIS PHYSEXAM
```

**MORE INFORMATION:** The creation of horizontal macro variable lists is introduced in Sections 6.2.2 and 11.1.1d.

### 11.4.2 Resolving Horizontal Lists

Because a horizontal list contains a series of words, it must be parsed in order to be used. The list is usually parsed using either the %SCAN or %QSCAN function. The macro %LISTDSN in Program 11.4.2 builds the list of data sets and then parses it using the %QSCAN function.

#### Program 11.4.2: Recovering Words from a Macro Variable Containing a List of Values

```
%macro listdsn;
%local dsnlist dsncnt i;
proc sql noprint;
  select distinct dsn
    into :dsnlist separated by ' '
      from macro3.dbdir;
  %let dsncnt = &sqllobs;
  quit;
%do i = 1 %to &dsncnt; ①
  %put &=i %qscan(&dsnlist,&i,%str( )); ②
%end;
%mend listdsn;
%listdsn
```

- ❶ Step through the &DSNCNT elements (words) in the list using a %DO loop.
- ❷ Using &I as the word number, the %QSCAN function is used to retrieve the individual words. It is important to notice here that the call to the %QSCAN function is not a full statement, but rather just a macro expression. While DATA step expressions, like function calls, cannot be used this way in the DATA step, such as in the PUT statement, the macro language is quite happy to process stand-alone expressions such as this one.

The SAS Log shows that for the three words in the list, the call to the %QSCAN function resolves to the &I<sup>th</sup> word in the list of words stored in &DSNLIST:

```
I=1 DEMOG
I=2 MEDHIS
I=3 PHYSEXAM
```

### 11.4.3 Stepping through the Horizontal List

As is demonstrated by Program 11.4.2 the %SCAN or %QSCAN function is used to recover the individual words from the list of words stored in the horizontal list.

The %QSCAN function (see Section 7.2.3) has three arguments: the list, the word number to retrieve, and the word separator. The %QSCAN in Program 11.4.3 demonstrates the use of these arguments:

```
%qscan(&classlist, ❶
      &i, ❷
      |) ❸
```

- ❶ The list of words. This will generally be a macro variable.
- ❷ The word number. This value must resolve to an integer between 1 and the number of words. Numbers outside this range cause the function to return a null value.
- ❸ The word separator. Notice that the value is *not* in quotes. When the list is space separated, a quoting function, such as %STR( ), can be used to preserve a space in the third argument.

In Program 11.4.3 the macro %PRTCLASS is used to print up to 10 observations from each level of a specified classification variable.

#### Program 11.4.3: Parsing and Using a Horizontal List of Values

```
%macro prtclass(dset=sashelp.class,
               classvar=sex,
               prtcnt=10,
               tst=on);
%local classlist classcnt j put;
proc sql noprint;
  select unique &classvar
    into :classlist separated by '|'
    from &dset;
%let classcnt = &sqlobs;
quit;

%if %upcase(&tst)=ON %then %let put=%nrstr(%put );
%else %let put=;

%do j = 1 %to &classcnt; ❶
  %unquote( ❷
    &put title1 "First &prtcnt obs of &classvar =
      %qscan(&classlist,&j,|)";
    &put proc print data=&dset(obs=&prtcnt);
    &put   where &classvar = "%qscan(&classlist,&j,|)"; ❸
    &put run;
  )
%end;
%mend prtclass;
```

- ❸ PROC SQL is used to create the macro variable &CLASSLIST to hold the unique values of the classification variable stored in &CLASSVAR.
- ❹ This macro allows code testing through the use of the &TST macro variable. When turned on (&TST=on), the macro variable &PUT is assigned the quoted keyword %PUT.

```
%prtclass(dset=sashelp.class, classvar=sex, prtcnt=5, tst=on ❹)
```

- ❺ An iterative %DO loop is used to step through the words (values of the classification variable) in the &CLASSLIST macro variable. If the number of levels had not been known, a %DO %WHILE loop could have been used.
- ❻ The %UNQUOTE function (see Section 7.1.6) removes the masking characters from %PUT when testing is turned on. When &TST=ON the SAS Log shows the generated code.

#### Program 11.4.3 (Portion of SAS Log): When Testing Is Turned On

```
title1 "First 5 obs of sex=F"
proc print data=sashelp.class(obs=5)
where sex = "F"
run
title1 "First 5 obs of sex=M"
proc print data=sashelp.class(obs=5)
where sex = "M"
run
```

- ❽ The WHERE clause is built by selecting the appropriate level of the classification variable by using the %QSCAN function to retrieve the &J<sup>th</sup> word.

When testing is not turned on (%UPCASE(&TST) ne ON), a PROC PRINT is written for each level of the classification variable.

**SEE ALSO:** Abbott (2015) makes horizontal lists of macro variables to assess data completeness.

---

## 11.4.4 Counting the Items in a List

In the horizontal list created in Program 11.4.3 we know how many items are in the list because have control of the list's creation. It is not unusual that we need to use a horizontal list for which we do not have the count of the number of items. There are a couple of ways to determine the number of words in a horizontal list.

### Stepping through the List Word-by-Word

The %WORDCOUNT macro shown in Program 11.4.4a steps through the incoming list one word at a time by using a %DO %WHILE loop. This macro function returns the word count without creating any global macro variables.

#### Program 11.4.4a: Counting Words Using %DO %WHILE and %QSCAN

```
%macro wordcount(list);
  /* Count the number of words in &LIST;
  %local count;
  %let count=0; ❶
  %do %while(%qscan(&list,&count+1❷,%str( )) ne %str()❸);
    %let count = %eval(&count+1); ❹
  %end;
  &count ❺
%mend wordcount;
```

- ❶ The count is initialized to 0.
- ❷ By using (&COUNT + 1) the %QSCAN function looks ahead to see if the next word exists.

- ❸ Check to see whether there is a next word. If there is, then enter the loop so that &COUNT can be incremented ❹. The %DO %WHILE will only iterate when the next word exists. When the word does not exist the loop terminates without incrementing the count.
- ❺ Since the &COUNT+1 word was found, increment the value of &COUNT.
- ❻ Return the word count by passing it back.

The use %WORDCOUNT is demonstrated in this section of SAS Log where the macro is called from within a %PUT statement.

```
45  %let vars = name sex age weight height;
46  %put The number of words in |&vars| is %wordcount(&vars);
The number of words in |name sex age weight height| is 5
```

It is very common to parse a horizontal list using a %DO %WHILE loop. The loop shown in Program 11.4.4a is often expanded for a variety or other purposes.

### Using the COUNTW Function

When all you need is the number of words in the list, it is easier and more efficient to gather the information directly. The COUNTW DATA step function is used in the macro function %COUNTW in Program 11.4.4b to count the words directly. This macro has the added advantage of the ability to specify the word delimiter.

#### Program 11.4.4b: Using COUNTW to Count the Words in a List

```
%macro countw(list,dlm);
  %* Count the number of words in &LIST;
  %if %length(&dlm)=0 %then %sysfunc(countw(&list)); ❶
  %else %sysfunc(countw(&list,&dlm)); ❷
%mend countw;
```

A portion of the SAS Log showing the usage of %COUNTW demonstrates the macro results when various delimiters are employed.

```
165  %let vars = name/sex\age weight height;
166  %put The number of words in |&vars| is %countw(&vars,%str( /)); ❸
The number of words in |name/sex\age weight height| is 5
167  %put The number of words in |&vars| is %countw(&vars); ❹
The number of words in |name/sex\age weight height| is 4
```

- ❶ When the &DLM parameter is not specified (❹), the comma should not appear following the first argument. The presence of the comma, with or without a second argument is sufficient to change the behavior of the COUNTW function. When no delimiter is specified, the default word delimiter list is used.
- ❷ When a word delimiter is specified, it is used as the second argument of the COUNTW function.
- ❸ Three word delimiters are specified (here a blank is included as the first character in the list).
- ❹ No delimiters are specified, so the default list, which does not include a backslash (), is used.

## 11.5 Using CALL EXECUTE

Although sometimes not as flexible as either horizontal or vertical lists of macro variables, CALL EXECUTE has the distinct advantage of not requiring the generation of a list of values (in either a horizontal or a vertical list) in the first place.

CALL EXECUTE (introduced in Section 6.5) is a DATA step routine. When the CALL EXECUTE is executed, its argument is immediately passed to the macro facility for evaluation. Arguments that contain macro statements and macro calls are immediately executed. Non-macro text or text generated by a macro is placed in a stack for execution after the completion of the current DATA step.

This routine is typically used when the values contained in a data set are to be applied as parameters to a macro. Because of the timing issues between the DATA step and the macro facility, as well as with the stack and how it is loaded, uses of the CALL EXECUTE can become very complicated very quickly.

In Program 11.2.8a a CSV control file is read into a DATA step, and a series of vertical macro variable lists are created. Later in Program 11.2.8b those lists are converted into a series of calls to the macro %SURVIVAL. In Program 11.5 we avoid the creation of the macro variable lists by using the CALL EXECUTE routine.

#### Program 11.5: Using CALL EXECUTE to Initiate a Series of Calls to the %SURVIVAL Macro

```
%* Test version of the SURVIVAL macro to check parms;
%macro survival(anumb=, acode=, atype=, refpopcode=, ①
                  activevar=, regimen=, eventcode=);
    /* Test parms transfer;
    %put *****&=anumb *****;
    %put _local_;
%mend survival;

filename xlscntrl "&path\data\control.csv"; ②

data _null_;
    infile xlscntrl dlm=',' truncover firstobs=2;
    length number code analysistype refpop ③
          censorrule eventcode arvregimen $8;
    input number $ code $ analysistype $ refpop $
          censorrule $ eventcode $ arvregimen $;

    if censorrule ne '-' then censor=censorrule; ④
    else censor=' ';

    call execute(catt( ⑤
                      '%survival(anumb=',number, ⑥
                      ',acode=',code, ⑦
                      ',atype=',analysistype,
                      ',refpopcode=',refpop,
                      ',activevar=',censor,
                      ',regimen=',arvregimen,
                      ',eventcode=',eventcode,
                      ')') ⑧
                ); ⑨
run;
```

- ① A dummy version of the %SURVIVAL macro is created to test the transfer of the parameters.
- ② The CSV control file is located.
- ③ The individual variables are specified for the PDV and read from the control file.
- ④ Because we are using the DATA step, logic is available for our use.
- ⑤ CALL EXECUTE accepts a single text argument. Here the CATT function is being used to concatenate character strings and the resolved values of the variables.
- ⑥ The macro call is enclosed in single quotes to prevent it from being immediately executed during the compilation phase of the DATA step. We need the concatenation to take place during the execution phase, which places the fabricated macro call in a stack for execution after the DATA step terminates.
- ⑦ The variable name is not quoted, which allows it to be resolved to the value it contains for this observation.
- ⑧ The %SURVIVAL specification is closed.
- ⑨ The CATT and the CALL EXECUTE functions are closed.

A portion of the SAS Log, which is generated by the %PUT \_LOCAL\_, shows that the transfer of information to the %SURVIVAL macro was successful.

```
*****ANUMB=1.1 *****
SURVIVAL ACODE ph38_6
SURVIVAL ACTIVEVAR 6
SURVIVAL ANUMB 1.1
SURVIVAL ATYPE prelim
SURVIVAL EVENTCODE 132
SURVIVAL REFPORPCODE 38
SURVIVAL REGIMENT na
*****ANUMB=1.2 *****
```

In the first two of the eight macro calls generated by the CALL EXECUTE routine, you can see how the character string generated by the CATT function is passed to CALL EXECUTE.

```
%survival(anumb=1.1,acode=ph38_6,atype=prelim,refpopcode=38,
activevar=6,regimen=na,eventcode=132)
%survival(anumb=1.2,acode=ph38_6,atype=prelim,refpopcode=38,
activevar=6,regimen=nn,eventcode=132)
```

**MORE INFORMATION:** CALL EXECUTE is introduced in Section 6.5. Similar to CALL EXECUTE, the DUSUBL function (see Section 8.5.1) can also be used in dynamic programming situations.

**SEE ALSO:** Michel (2005) introduces and covers some of the caveats associated with CALL EXECUTE.

Whitlock (1997) provides a good overview to the CALL EXECUTE routine.

A list of macro variables is cleared using SASHELP.VMACRO and CALL EXECUTE in an example by Rhoads and Letourneau (2002).

Jiang (2003) builds a macro call that is executed through a CALL EXECUTE.

## 11.6 Writing %INCLUDE Programs

Like the CALL EXECUTE (see Section 11.5), which is used in the DATA step partly to avoid the creation of macro variable lists, you can also use the DATA step to write code directly, avoiding not only the macro variable lists, but the CALL EXECUTE as well.

By default the DATA step's PUT statement writes to the SAS Log; however, when a FILE statement is specified the result of the PUT can be routed to a file. This file can contain SAS statements or even macro code. If the file is brought back into SAS using the %INCLUDE statement, the code contained in the file will be executed.

In Program 11.5 the contents of a control file are used to create a series of macro calls through the use of CALL EXECUTE. In Program 11.6 the same control file is used to create the same macro calls; however, the CALL EXECUTE routine is not used. In this program the macro calls are written to a file and then that file is included for execution using the %INCLUDE statement.

**Program 11.6: Writing Code by Using the DATA Step PUT Statement**

```

%* Test version of the SURVIVAL macro to check parms;
%macro survival(anumb=, acode=, atype=, refopcode=,
                 activevar=, regimen=, eventcode=);
  %* Test parms transfer;
  %put *****&anumb ****;
  %put _local_;
%mend survival;

filename xlscntrl "&path\data\control.csv"; ①
filename inc11_6 "&path\chapter 11\sas programs\Survivalcalls11_6.sas";

data _null_;
  infile xlscntrl dlm=',' truncover firstobs=2;
  file inc11_6;
  length number code analysistype refpop
         censorrule eventcode arvregimen $8;
  input number $ code $ analysistype $ refpop $
        censorrule $ eventcode $ arvregimen $;

  if censorrule ne '-' then censor=censorrule;
  else censor=' ';

  textval=catt( ②
    '%survival(anumb=',number,
    ',acode=',code,
    ',atype=',analysistype,
    ',refopcode=',refpop,
    ',activevar=',censor,
    ',regimen=',arvregimen,
    ',eventcode=',eventcode,
    ')'
  );
  put textval; ③
  run;
%include inc11_6; ④

```

- ① A *fileref* is established for both the incoming control file (XLSCTRL) and the outgoing file (INC11\_6), which will later be included for execution ④.
- ② A character variable is created using the CATT function that contains the code that is to be written to the file.
- ③ The PUT statement writes the text stored in the DATA step variable TEXTVAL to the file specified using the *fileref* INC11\_6.
- ④ The %INCLUDE statement is used to point to the file that is to be included back into the program stream. The file that is included can be inspected by using the editor, and the first two macro calls are shown here (the code wraps to fit into the text box for this book).

```

%survival(anumb=1.1,acode=ph38_6,atype=prelim,refopcode=38,activevar=6,reg
imen=na,eventcode=132)
%survival(anumb=1.2,acode=ph38_6,atype=prelim,refopcode=38,activevar=6,reg
imen=nn,eventcode=132)

```

In this particular example the use of the PUT and %INCLUDE is a reasonable solution to this problem. As the code that is to be written becomes more complex, the DATA step itself can become very difficult to write and debug. The advantage is that this solution requires no macro experience, but ultimately is rarely the best solution for dynamic coding situations.

**MORE INFORMATION:** Program 12.4.1a uses this technique to build a %LET statement.

**SEE ALSO:** Graebner (1998) uses the DATA \_NULL\_ step to write a PROC REPORT step. Reading (2000) uses SAS data sets to construct a macro that is then brought in through the use of a %INCLUDE. Chow (1999) discusses both the %INCLUDE and the CALL EXECUTE. Johnson (2001) demonstrates the use of the %INCLUDE as part of a discussion of writing code that writes code. Morrill, Wiser, and Zhao (2002) build an extensive macro and macro call that is included for execution. Miralles (2016) uses this technique to write HTML code for a web report platform. A DATA \_NULL\_ step is used by Krenzke *et al* (2014) to write a macro definition.

## 11.7 Writing Applications without Hardcoded Data Dependencies

An application generally consists of a series of interrelated programs. They may or may not include a user interface, but almost certainly they are going to contain macro language elements. Since a well-written set of programs can have broader application than originally intended, by writing the programs to be as dynamic as possible (that is, by removing as many hardcoded data dependencies as is possible) the programs become more reusable.

A dynamic program should be able to gather as much of the information that it needs to operate on its own—without programmer intervention. Assume that you need to perform a PROC MEANS on all numeric variables that start with the letters WT. (There may be nonnumeric variables in the data set that have names that also start with WT, so we can't simply use VAR WT:;) You are given only the name of the data set and the variable prefix. A dynamic macro will be able to determine the names of the available numeric variables that meet the naming requirements and will then build the appropriate VAR statement. If the list of analysis variables changes in the future, a macro written using dynamic programming techniques will adjust without any recoding.

Let's say that you have written a series of interesting and perhaps even complex SAS programs that perform a variety of data entry operations, data checks, exception reporting, statistical analyses, and summary reporting. Since the next study or project is somewhat similar to the one you just completed, you might be planning to build another set of programs based on (cannibalized from) the ones that you just finished. Wouldn't you rather build the programs once? If the macros are able to somehow gather the information that they need to execute successfully whenever they are run, and not when you initially write the macros, then you will not need to modify the programs for each task or project. The macros are dynamic and can adjust to the task.

There are a number of ways that macros can search out and find the information they need at run time. Section 11.2 discusses several of the primary information sources that can be used by your macro. These data sets and information sources are in turn used to create a series of SAS macro variables that are available to the programs and macros of the application. All project, data set, and variable-specific information is stored in the macro variables and hence never in the programs themselves. Once implemented all the programs in your application become data independent.

Most dynamic programs will use one or more of these information sources. For complex dynamic applications it is often necessary to fully describe all the data tables, their names and attributes, as well as the variables in each of those data tables, including their names and attributes. Usually the only practical way of storing and maintaining this information is through the use of one or more data tables. These tables are usually referred to as control tables, and since their use is a bit involved they are discussed or used in many of the examples in Section 11.7.

The great advantage of writing dynamic programs is that it is possible to write them without hardcoding things like data set and variable names. This means that when data set attributes change, you can make the corresponding changes in your control file, not your program. Let "them" redefine the project. Your code is ready.

**SEE ALSO:** Jim Sattler (2003) discusses the elements of programs that he calls “Data Driven.” Molter, Millard, and Paciocco (2003) use metadata to construct SQL queries, and Luo and Luo (2003) discuss some macros that can be used in metadata construction.

Shen (2003) uses Excel spreadsheets to build the metadata used to drive the dynamic programs.

### 11.7.1 Generalized and Controlled Repeatability

At the completion of a successful project everyone should be happy with what you have done as a SAS programmer. If everything went fairly well, then you are pleased, the boss is pleased, and the client is pleased. Of course your job is not really completed. The problem is that you now need to document what you did, create the data dictionaries (even though ideally the dictionaries and the documentation should come first). Reality is that we generally do not have the luxury of the time needed to wrap up the first project before starting the next. And when we do have the time we need to go back and reconstruct and document what was done.

When you prepare for that next study or project, if it has similarities to the first there is a strong temptation to copy and modify the existing programs. This means that you will need to reconstruct what you have done and then modify the existing programs for the upcoming project by changing data set names, variable names, and variable attributes. Once modified, these programs will of course then have to go through the validation and documentation process as well.

Wouldn't you rather build the programs so that they will be ready to go for the next study regardless of the number of data sets, names of variables, and error-check specifications? Wouldn't you rather just validate your programs once and not over and over again? Imagine the savings in change-control management alone with only one version of each program.

Many studies, including most clinical trials, are very data intensive. Very often there is a large number of distinct SAS data sets, each with a diverse suite of variables, and the data sets themselves might range from a few to many observations. Just keeping track of these data sets becomes a chore for the database manager.

Management issues become even more intense when an application is developed that must operate against these data sets. As part of the data management, analysis, and reporting process, numerous SAS programs are usually written to support the application.

Very often names of data sets, variables, and other data set and project-specific information is embedded within these programs. This makes it difficult, time-consuming, and expensive to modify the application for another similar project. A dynamic and automated application will overcome these limitations by avoiding any project, data set, or variable dependencies.

How then do we create an application that will work for any number of data sets, with any number of observations, and with any combination of variables? How can the application be written so that it requires little or no recoding when being ported from one project to the next?

Fortunately, although there are major differences between studies, many of the tasks are similar. Most studies require data entry and data validation. Many of the exception and adverse-event captures and reports are similar. Since these major events are similar across projects, it ought to be possible to generalize our programs so that they need not be modified for each project.

Indeed, it is possible to build the SAS programs so that they are general enough to work for each of your studies. The answer lies in the creation of data dictionaries that can be used as control files in order to build a series of macro variables, which are in turn used dynamically by the application. The key is to build a structure into your programs that is based on those things that are common to all projects. Some of these commonalities include project identification, library and folder relationships, data sets with specifiable characteristics, variables within data sets with specifiable characteristics, and variable-specific value constraints.

**SEE ALSO:** In two papers presented by Carpenter and Smith (2000 and 2001), a number of considerations regarding the building and maintaining of dynamic programs are discussed.

## 11.7.2 Setting Up Project Control Files

The data dictionaries or control files become both the starting point and heart of the control process for a dynamic application. This means that relative to your programs the addition of a new data set into the study or a change in a data set's variables might be as simple as changing one control file. Done properly, all of the programs that depend on these control files will require **no** modification when the control files are changed. This also implies that implementing a new study is as simple as building a new set of data dictionaries, and, of course, this is something that you should be doing first anyway.

The control files can be SAS data sets, Excel tables, or even flat files. With the use of PROC IMPORT and the myriad of SAS/ACCESS tools a large number of possibilities exist. The way that you build and maintain these files should be consistent with your work flow. How you create these control files is much less important than their accessibility. The kinds and types of information contained within these control files will depend, of course, on how you use them.

The example application described in the examples in the following sections is based on an actual, albeit more complex application. The original application was a double data-entry system that was also used for data validation, data set comparisons, data integrity checks, and data process control. The application programs had to be used for multiple cross platform studies with the ability to implement a new study within a day.

Several separate control files were developed to provide the data-dependent information necessary to enable the macros of the application to be written independent of the data and still operate successfully. The three control data sets described here are some of the primary ones that you might need for this type of dynamic application. For demonstration purposes each has been simplified substantially.

### DBDIR

Data set definitions

### VARDIR

Variables within data sets

### FLDDIR

Data field constraints.

For the examples shown below vertical macro variable lists were used (see Section 11.3). Each column or variable in these data sets will become a vector of macro variables with one macro variable for each data value in the data set.

### DBDIR

This data set (Table 11.7.2a) contains one observation for each data set in the project. This data set was first introduced in Section 6.5.1 and is also used in several examples earlier in this chapter. In addition to the data set name, variables in this type of data set often include data sheet page number, key variables (primary key), data set label, secondary keys, and other data-set-specific information such as might be required by the various analysis programs. The simplified version used in these examples contains only the data set name (DSN) and the variables that form the primary key (KEYVAR).

**Table 11.7.2a: MACRO3.DBDIR**

Obs	dsn	Keyvar
1	DEMOG	SUBJECT
2	MEDHIS	SUBJECT MEDHISNO SEQNO
3	PHYSEXAM	SUBJECT VISIT SEQNO

## VARDIR

This data set contains one observation for each variable in each data set. For each row or variable, it contains variable-specific items such as formats, labels, and variable length. Variables that are in all of the data sets within the project have ALL as the data set name so that they do not need to be constantly repeated.

**Table 11.7.2b: Variable Attribute Control File**

Obs	dsn	Var	VarType	Label
1	ALL	SUBJECT	\$8	Patient number
2	ALL	PTINIT	\$8	Patient initials
3	DEMOG	CENTER	\$3	Clinic number
4	DEMOG	DOB	8	Date of birth
5	DEMOG	SEX	\$1	Sex (M or F)
6	DEMOG	RACE	\$1	Race Code
7	MEDHIS	SEQNO	8	History Sequence Number
8	MEDHIS	MEDHISNO	8	Medical History Number
9	MEDHIS	MHDT	8	Date of medical history
10	PHYSEXAM	VISIT	8	Visit Number
11	PHYSEXAM	SEQNO	8	Exam Sequence Number
12	PHYSEXAM	PHDT	8	Physical exam. date
13	PHYSEXAM	WT	8	Weight

The data sets DBDIR and VARDIR are used to build the data dictionary and to document the data sets and the variables that they contain.

## FLDDIR

The data set MACRO3.FLDDIR identifies data constraints for each data entry field or variable. These constraints can be used to build data exception and error-trapping reports. The advantage of this approach is that constraint changes are implemented in the data set and not in the code itself.

**Table 11.7.2c: Data Validation Control File**

Obs	dsn	Var	ChkType	ChkText
1	DEMOG	CENTER	notmiss	
2	DEMOG	RACE	list	('1','2','3')
3	MEDHIS	MHDT	format	date7.
4	MEDHIS	SUBJECT	equals	medhisno

Any number of different types of checks are possible. Four common types of checks are included in Table 11.7.2c. The variable CHKTYPE specifies the type of check and the variable CHKTEST contains supplemental information when needed by the specific check.

### NOTMISS

The variable may not contain missing values

### LIST

The value must be in the list of values in CHKTEXT

**FORMAT**

The formatted value of the variable (using the format in CHKTEXT) must not be missing. User-defined formats are permitted.

**EQUALS**

The two variables should have the same value

**MORE INFORMATION:** The FLDDIR control file is used to create the code used to make the constraint checks in Program 11.7.5.

### 11.7.3 Using Control Files to Build Macro Variable Lists

Each of the control files is used to create a series of macro variables. The observations are counted and the observation number becomes a part of the macro variable name. This results in names such as &DSN1, &DSN2, &DSN3, and so on. Although the macro language does not have an array statement per se, this series of macro variables can be used as a vector of values. This vector effectively becomes a macro array.

The creation of macro variable lists is discussed earlier in this chapter in Sections 11.1.1, 11.3, and 11.4. Program 11.3.1 writes the contents of MACRO3.DBDIR into a vertical list, while Program 11.4.1 creates a horizontal list. Which type of list you choose to create will depend on your personal preferences and how the list is to be used. Generally I find the vertical list form to be more flexible.

Depending on your application, you might wish to place these lists in the global symbol table, where they will be available throughout the application, or in a local table when they are needed for a specific purpose. This also should depend on the specific application. Probably the most important consideration is that you make the choice deliberately and not leave the table assignment to chance. When using the SYMPUTX routine, the assignment can be made explicitly through the use of the third argument (see the discussion for Program 6.1.2b). Table assignment for macro variables created through the use of the SQL INTO clause is not as straightforward.

In Program 11.7.3 the list of data sets stored in DBDIR is loaded into a macro list; however, the number of data sets is first counted and then %LOCAL statements are used to force all of the macro variables in the list into the local symbol table. This requires two passes of the incoming data, but allows us to control which symbol table is to receive the macro variables (in this case the local table).

#### Program 11.7.3: Forcing a List of Macro Variables into the Local Symbol Table with SQL

```
%macro UseDSN;
%local dsncnt i; ③
proc sql noprint;
    select count(dsn) ④
        into: dsncnt
        from macro3.dbdir;
%do i = 1 %to &dsncnt; ⑤
    %local dsn&i; ⑥
%end;
select dsn
    into :dsn1 - ⑦
    from macro3.dbdir;
quit;

/* other code to use the &DSNxx list goes here; ⑧
%put _local_; ⑨
%mend usedsn;
%let dsn2 = in the global table; ⑩
%usedsn ②
%put &=dsn2; ⑩
```

- ❶ Just to show that the %LOCAL works, the macro variable &DSN2 is placed in the global symbol table.
- ❷ The macro %USEDSEN is called.
- ❸ The macro variables &DSNCNT and &I are placed in the local symbol table. We cannot explicitly place the remaining macro variables (&DSN1, &DSN2, and so on) into the local table because we do not yet know how many there are in the list. That number has to be determined dynamically.
- ❹ We read the incoming data set and count the number of data sets. This count is saved in &DSNCNT, which is already in the local table. If the %OBSCNT macro (see Program 11.2.6a) had been used, the SQL step and a pass of the data could have been avoided.
- ❺ A %DO loop is used to establish a succession of %LOCAL statements. Each %LOCAL statement must be created individually as we cannot place the %DO inside the %LOCAL statement.
- ❻ The &I will be resolved before the %LOCAL statement is executed. This will establish each of the macro variables in the list on the local table.
- ❼ The individual macro variables in the list are created. Since they are already established on the local symbol table (by the %LOCAL statement), these macro variables will also be local (even &DSN2 which also has a global version). Without the %LOCAL statement &DSN2 would have been written to the global symbol table.
- ❽ The code that uses this local list would typically occur here, perhaps as a series of macro calls. Remember that any macro variables in this local table will be available to any nested macros.
- ❾ To help understand this example, this %PUT statement was added to write the local macro variables and their values to the SAS Log.
- ❿ To show that the global version of &DSN2 was not modified, the %PUT statement writes the global version of &DSN2 to the SAS Log.

The SAS Log shows that the macro variables were successfully written to the local symbol table.

**Program 11.7.3: (Portion of SAS Log): Showing That Macro Variables Were Successfully Written to the Local Symbol Table**

```

USEDSEN DSN1 DEMOG
USEDSEN DSN2 MEDHIS
USEDSEN DSN3 PHYSEXAM
USEDSEN DSN4 VITALS
USEDSEN DSNCNT      4
USEDSEN I 5
USEDSEN SQLEXITCODE 0
USEDSEN SQLOBS 4
USEDSEN SQLOOPS 24
USEDSEN SQLRC 0
USEDSEN SQLXOBS 0
80   %put &=dsn2;
DSN2=in the global table

```

**SEE ALSO:** Yao (1997) discusses a macro that derives control values from a flat file.

---

#### 11.7.4 Using Control Files to Create Empty Data Sets

In a clinical study it is important that the attributes (variable names, length, type, labels, and so on) be consistent with the specifications of the data dictionary. One way to ensure that is to build the data sets themselves based on the information contained in the data dictionary, which in our examples are the control files described in Section 11.7.2.

If we are to build a series of data sets, we need to know the names of the data sets (DBDIR) and the list of variables and their attributes within each of those data sets (VARDIR). Since this information is stored in control data sets, we can read the information into macro variable lists and then loop through the lists. In this case two nested loops will be needed, with the outer loop cycling through the data sets, and the inner loop cycling through the variables for each of the data sets in the outer loop.

Program 11.7.4 builds a series of zero observation data sets that will be used as prototypes for the analysis data sets. For each data set the list of variables in the KEEP= data set option, the LENGTH statement, and the LABEL statement is built dynamically. Notice that in the building of each of these statements, the outer loop (&JJ) increments once for each data set, while the inner loop (&KK) cycles through all possible combinations of data sets and variables. A macro %IF statement selects the appropriate variables for a given data set.

#### Program 11.7.4: Building a Series of Zero Observation Data Sets

```
%macro bldempty;
/* Create lists;
proc sql noprint; ①
  select dsn
    into :dsn1-
      from macro3.dbdir;
%let dsncnt=&sqlobs;
select dsn, var, vartype, label
  into :vdsn1|,
      :vvar1|,
      :vtyp1|,
      :vlbl1|
    from macro3.vardir;
%let varcnt = &sqlobs;
quit;

%do jj = 1 %to &dsncnt; ②
  /* One data step for each data set;
  /* The JJ loop steps through the data sets;
  data &&dsn&jj(keep= ③
    /* Build the var list to keep for this DB;
    /* The KK loop steps through the variables list;
    %do kk = 1 %to &varcnt; ④
      /* include the ALL variables in every data set;
      %if &&dsn&jj=&&vdsn&kk or &&vdsn&kk=ALL ⑤
        %then &&vvar&kk; ⑥
    %end;
    );
    * Use length to define variable attributes;
    length
      %do kk = 1 %to &varcnt;
        %if &&dsn&jj=&&vdsn&kk or &&vdsn&kk=ALL
          %then &&vvar&kk &&vtyp&kk; ⑦
      %end;
      ;
    * Define the variable labels;
    label
      %do kk = 1 %to &varcnt;
        %if &&dsn&jj=&&vdsn&kk or &&vdsn&kk=ALL
          %then &&vvar&kk = "&&vlbl&kk"; ⑧
      %end;
      ;
    stop; ⑨
  run;
%end;
%mend bldempty;
%bldempty
```

Within the DATA step, using a variable list appropriate to that data set, %DO loops are used to build the ③ KEEP= data set option, ⑦ LENGTH statement, and the ③ LABEL statement.

- ① An SQL step is used to build the appropriate macro variable lists from the control files.
- ② The outer (&JJ) %DO loop steps through the list of data set names from DBDIR.

- ③ A KEEP= option is created for the data set &&DSN&JJ.
- ④ The inner (&KK) loop steps through all the variables for all the data sets.
- ⑤ The %IF statement is used to determine whether the given &&VVAR&KK variable is in the &&DSN&JJ data set.
- ⑥ Variable names that are in the &&DSN&JJ data set are added to the option. For the DEMOG data set the DATA statement becomes:

```
data DEMOG(keep= SUBJECT PTINIT CENTER DOB SEX RACE );
```

- ⑦ The LENGTH statement is generated using similar code as was used for the KEEP= option. For the DEMOG statement the LENGTH statement becomes:

```
length SUBJECT $8 PTINIT $8 CENTER $3 DOB 8 SEX $1 RACE $1 ;
```

- ⑧ For the DEMOG data set the LABEL statement becomes:

```
label SUBJECT = "Patient number" PTINIT = "Patient initials" CENTER =
"Clinic number" DOB = "Date of birth" SEX = "Sex (M or F)" RACE = "Sex
Code" ;
```

- ⑨ The STOP statement prevents the output of any observations. A portion of the SAS Log shows that the data set DEMOG was created with zero observations. It is anticipated that each variable will be uninitialized. This note could be eliminated if the RETAIN statement with initial values had also been coded.

```
NOTE: Variable SUBJECT is uninitialized.
NOTE: Variable PTINIT is uninitialized.
NOTE: Variable CENTER is uninitialized.
NOTE: Variable DOB is uninitialized.
NOTE: Variable SEX is uninitialized.
NOTE: Variable RACE is uninitialized.
NOTE: The data set WORK.DEMOG has 0 observations and 6 variables.
```

Since the variable loop encompasses all variables in all data sets, the %IF ⑤ is used to select the appropriate variables from the JJ<sup>th</sup> data set. Two %DO loops are used to coordinate the two lists of macro variables. The outer loop (with index of &JJ) steps through the list of data sets. The inner loop (with index &KK) steps through all the variables for all the data sets. Since we are interested only in the variables for the data set identified by &&DSN&JJ, that value is compared to the value of the data set in the inner loop (&&VDSN&KK).

### 11.7.5 Using Control Files to Create Data Validation Checks Dynamically

In the previous example the list of data set variables encompasses all data sets; consequently, when we want to build a list of variables for a given data set we need to coordinate the lists by matching the data set name. An alternative is to build the second (and potentially longer) list only when it is needed. We can then use IF-THEN logic or WHERE processing to build the secondary list that contains only the elements that are needed at that time. For example, the list could contain the variables within a given data set.

Taking this approach the values in the FLDDIR data set are not loaded into macro variables until they are needed. This means that if we are within a macro loop that spans data sets (&&DSN&JJ), we can create the field checklist appropriate only for that particular data set. This avoids the %IF-%THEN checking of data set names that was required in Program 11.7.4, and consequently, the symbol table for the field checks will only include those macro variables that are needed at that time for the &&DSN&JJ data set.

In Program 11.7.5 we want to perform field checks for the data sets in the study. The checks are listed in FLDDIR, which contains one observation per check, and there can be any number of checks per variable in any given data set. If a new check is to be added, the code will not change; all we have to do is add another line to the FLDDIR data set.

**Program 11.7.5: Implementing Field Checks Based on a Control File**

```
%macro FldChk;
%local i jj dsncnt fldcnt;
/* Create list of data set names;
proc sql noprint; ①
  select dsn,keyvar
    into :dsn1-,
      :keys1-
      from macro3.dbmdir;
%let dsncnt=&sqlobs;
quit;

/* Step through the list of data sets;
%do jj = 1 %to &dsncnt; ②

  /* Determine field checks for this data set;
  proc sql noprint;
    select dsn,var,chktype,chktext
      into :fdsn1-,
        :fvar1-,
        :ftyp1-,
        :ftxt1-
        from macro3.flmdir
        where (upcase(dsn)=upcase("&&dsn&jj")); ③
%let fldcnt=&sqlobs;
quit;

%if &fldcnt > 0 %then %do; ④
  * Perform field checks;
  data CheckError_&&dsn&jj(keep= &&keys&jj _obs_ var ⑤
                                msg text value chkdate);
    set macro3.&&dsn&jj;

    * Date these field checks;
    retain chkdate %sysfunc(today());
    format chkdate date9.;

    * Specify lengths for the descriptor variables;
    length var $15 value $25 text msg $100;

    * Note the observation number;
    _obs_ = _n_;

    /* Build the Field error checks;
    %do i = 1 %to &fldcnt; ⑥
      %if %upcase(&&ftyp&i) = LIST %then %do; ⑦
        if &&fvar&i not in&&ftxt&i then do; ⑧
          var = "&&fvar&i"; ⑨
          msg = 'Value is not on list';
          text = "&&ftxt&i";
          value = &&fvar&i;
          output CheckError_&&dsn&jj; ⑩
        end;
      %end;
      %else %if %upcase(&&ftyp&i) = NOTMISS %then %do;
        if missing(&&fvar&i) then do;
          var = "&&fvar&i";
          msg = 'Value is missing';
          text = "&&ftxt&i";
          value = &&fvar&i;
          output CheckError_&&dsn&jj;
        end;
      %end;
    %end;
  %end;
```

```

%else %if %upcase(&&ftyp&i) = FORMAT %then %do;
  if left(put(&&fvar&i,&&ftxt&i))='.' then do;
    var = "&&fvar&i";
    msg = 'Formatted Value is missing';
    text = "&&ftxt&i";
    value = &&fvar&i;
    output CheckError_&&dsn&jj;
  end;
%end;
%else %if %upcase(&&ftyp&i) = EQUALS %then %do;
  if &&fvar&i ne &&ftxt&i then do;
    var = "&&fvar&i";
    msg = "&&fvar&i NE &&ftxt&i";
    text = catt("&&ftxt&i is ",&&ftxt&i);
    value = &&fvar&i;
    output CheckError_&&dsn&jj;
  end;
%end;
%end; /* Complete loop to build field checks;
run;
%end; /* Complete %DO block if fieldcnt>0 for this JJ;
%end; /* complete the JJ across data set loop;
%mend fldchk;

```

- ➊ Build the list of data sets that are to be checked. For observation identification, include the list of their key variables.
- ➋ Use a %DO loop to pass through the &DSNCNT data sets to be checked.
- ➌ Build a list of field checks that are to be performed for the &&DSN&JJ data set. Notice the use of the WHERE to eliminate checks for other data sets.
- ➍ Only perform checks for data sets that have one or more defined checks.
- ➎ Create an error report data set. When checking the DEMOG data set it will be named CHECKERROR\_DEMOG. Its variables need to be sufficient so that the data manager can find the specific observation with the error.
- ➏ Use the %DO loop to cycle through all the field checks for this data set.
- ➐ Each check has a check type (&&FTYP&I), which can be matched to the specific code associated with this type of check.
- ➑ The DATA step IF statement is constructed to perform the actual field check.
- ➒ The variables needed to identify the problem are constructed. These include the name and value of the variable with the problem.
- ➓ Each data observation with a problem is written to the data set collecting the error messages. There will be one observation for each data error and any given data observation can have multiple errors.

There are several types of field checks, including LIST, which specifies a list of acceptable values. When &&FTYP&I = LIST ␗, an IF-THEN/DO block is defined that checks to see whether the value stored in the variable named in &&FVAR&I is in the list stored in &&FTXT&I ␒. When the value is not in the list, the field is in error, and a series of assignment statements are executed ␓. For the second observation in FLDDIR,

OBS	DSN	VAR	CHKTYPE	CHKTEXT
2	DEMOG	RACE	list	('1','2','3')

the DO block becomes the following:

```

if RACE not in('1','2','3') then do; ⑧
  var = "RACE"; ⑨
  msg = 'Value is not on list';
  text = "('1','2','3')";
  value = RACE;

```

```
output CheckError_DEMOG;
end;
```

This DATA step dynamically expands to accommodate new field checks. When the macro was written, the programmer had no specific information about the name of the data set, the name of the variable, the type of field check, or the acceptable values for the field check.

**MORE INFORMATION:** The macro %FIXRAW in Section 12.4.8 uses a different approach to build code to alter data values.

## 11.8 Building SAS Statements Dynamically

When building SAS statements dynamically, it is very typical to build them “from the inside out”. That is the macro code appears inside what will become the SAS statement. The first example of this was seen in Program 5.3.2b, which is repeated here. This is a very important coding technique and is well worth your time investment in understanding it well.

### Program 5.3.2b: Building Statements from the Inside by Using Macro Code

```
%macro allyr2(start=10,stop=14);
  data allyear;
    set ❶
    %do year = &start %to &stop;
      yr&year(in=in&year) ❷
    %end;; ❸
    year = 2000 ❹
    %do year = &start %to &stop;
      + (in&year*&year) ❺
    %end;; ❻
    run;
%mend allyr2;
%allyr2(start=11, stop=13)
```

- ❶ The SET keyword is coded, which initiates the SET statement, but the names of the incoming data sets have yet to be written.
- ❷ The macro %DO loop will write these data set names after &YEAR has been resolved.
- ❸ The first semicolon closes the %END statement and the second semicolon will close the SET statement, which was begun at ❶.
- ❹ The assignment statement is started, but not completed.
- ❺ The expression used in the assignment statement is written by the macro %DO loop.
- ❻ The first semicolon terminates the %END statement and the second terminates the assignment statement begun at ❹.

This call to the macro %ALLYR2 [%allyr2(start=11, stop=13)] generates the single DATA step shown here:

```
data allyear;
set yr11(in=in11)
     yr12(in=in12)
     yr13(in=in13);
year = 2000 +
      (in11*11) +
      (in12*12) +
      (in13*13);
run;
```

Both the SET statement and the assignment statement have been generated dynamically through the use of a %DO loop to insert a series of items within the statement.

In Program 11.7.5 a series of statements are also dynamically completed. A snippet from that program shows that while a list element of the form &&VAR&I is used within the various statements being constructed, the %DO loop is not used within the statement. In this case there is only one element to insert into the statement; however, since the block of statements is created multiple times, the %DO surrounds the entire block.

#### **Program 11.7.5 (Partial): Showing Completion of Statements without a %DO loop**

```
%do i = 1 %to &fldcnt;
    . . . . code not included . . .
    %else %if %upcase(&&ftyp&i) = NOTMISS %then %do;
        if missing(&&fvar&i) then do;
            var = "&&fvar&i";
            msg = 'Value is missing';
            text = "&&ftxt&i";
            value = &&fvar&i;
            output CheckError_&&dsn&jj;
        end;
    %end;
    . . . . code not included . . .
```

**MORE INFORMATION:** A number of other examples in this book use this type of coding techniques. Program 12.1.1 builds a SELECT statement using a %DO on the inside of the statement.

## **11.9 More Than Just the Macro Coding**

Truly dynamic applications tend to have multiple macros that effectively coordinate with each other. This coordination, especially when being applied across projects or studies, relies on more than just the macros themselves to be successful. While the topics discussed in this section are not necessarily “macro topics”, macro programmers who write successful dynamic macros will take these concepts into consideration when developing their macros.

For a dynamic application to work consistently for multiple projects, consistency in a number of areas must be built into the system. This implies that there are established conventions that the programmer can rely on, which might include the following:

- Naming conventions
- Directory structure
- The way that the SAS environment is configured and initiated
- Macro libraries
- Control of *libref* and *fileref* definitions

**SEE ALSO:** Thompson (2015) discusses work flow and strategies for creating robust and reusable code.

### **11.9.1 Naming Conventions**

You will not always have control over the naming of macro variables, data set variables, and data sets within your dynamic program. The names that you cannot control will hopefully be specified in control files such as those described in Section 11.7.

When you do have the ability to pick names, be as consistent as you can. In several earlier sections in this chapter (see Programs 11.3.1, 11.3.2, and 11.7.3) the names of the data sets stored in MACRO3.DBDIR are

written to a list of macro variables. In each case the macro variables are named &DSN1, &DSN2, and so on. By always using the same naming convention for this macro variable list, programs that use this list can be more general. In Program 11.9.1a we want to determine the mean and count of height and weight of males and females.

#### Program 11.9.1a: Inconsistent Naming of Variables

```
proc summary data=macro3.clinics;
  class sex;
  var ht wt;
  output out=summry
    n= n_ht weight_n
    mean= mHT Mean_wt;
run;
```

The PROC SUMMARY is successful; however, the names of the variables are inconsistent and unpredictable.

**Table 11.9.1a: Showing Inconsistent Naming Conventions**

Obs	sex	_TYPE_	_FREQ_	n_ht	weight_n	mHT	Mean_wt
1		0	80	80	80	67.4500	161.775
2	F	1	32	32	32	65.0000	145.875
3	M	1	48	48	48	69.0833	172.375

There are a number of options that you can use to help you with consistency. In Program 11.9.1b the /AUTONAME option is used when generating the desired statistics.

#### Program 11.9.1b: Using /AUTONAME to Achieve Consistent Variable Names

```
proc summary data=macro3.clinics;
  class sex;
  var ht wt;
  output out=summary
    n=
    mean= /autoname;
run;
```

Notice that the name of the OUT= data set has been changed. In Program 11.9.1a it is named SUMMRY and in 11.9.1b it is SUMMARY. In automated systems it is easy to overlook small differences like this – especially subtle misspellings.

A printout of WORK.SUMMARY shows that the /AUTONAME option creates consistent and predictable variable names. The new variables are always named as a combination of the analysis variable and the statistic (*variable\_statistic*). Not only are the names predictable, but the order of the variables is also predictable (and controllable by order of the specification of the analysis variables and the statistics).

**Table 11.9.1b: Consistent Variable Names Using /AUTONAME**

Obs	sex	_TYPE_	_FREQ_	ht_N	wt_N	ht_Mean	wt_Mean
1		0	80	80	80	67.4500	161.775
2	F	1	32	32	32	65.0000	145.875
3	M	1	48	48	48	69.0833	172.375

There are a couple of nice benefits gained by the /AUTONAME option and these serve to further demonstrate naming principles. First, related variables are named with a consistent prefix. In this case all the statistics based on the variable HT start with HT\_. This allows us to gather all the statistics on this variable by using the HT\_ list abbreviation. Second, we can depend on all the variables containing a mean to have a name that finishes with \_MEAN. From the macro programmer's perspective we can now harvest all statistics for a given variable without knowing what statistics have been requested, or we can harvest all the variables containing a given statistic without knowing the analysis variable(s).

In Program 11.9.1c, the variables in the SUMMARY data set, created in Program 11.9.1b, are listed using a PROC CONTENTS step and this list is used in a DATA step that collects the names of variables with selected attributes.

#### **Program 11.9.1c: Collect Variable Names with Selected Attributes**

```
proc contents data=summary ❶
  out=cont
  noplay;
run;

data _null_;
  set cont end=eof;
  if index(name, '_Mean') then do; ❷
    * Gather the names of variables with the MEAN;
    i+1; ❸
    call symputx(catt('Mean',i),name); ❹
  end;
  if index(name, 'ht') then do; ❺
    * Gather the names of variables with a height statistic;
    j+1;
    call symputx(catt('HT',j),name);
  end;
  if eof then do;
    call symputx('meancnt',i); ❻
    call symputx('HTcnt',j);
  end;
run;
```

- ❶ PROC CONTENTS is used to create a list of variable names.
- ❷ Note the variables whose name contains '\_Mean'.
- ❸ Increment the variable counter.
- ❹ Save the variable name containing the MEAN in a macro variable list (&MEAN1, &MEAN2, and so on).
- ❺ Repeat for variables with HT in the name.
- ❻ When all the variable names have been read, save the counter for each of the two lists.

---

## **11.9.2 Directory Structure**

Applications that will be used across projects, systems, and even platforms must be written to accommodate the variety of situations that will be encountered. As a developer of the application, you will need to take into consideration a number of aspects that will need to be coordinated within the framework of the application. One of your critical issues will be to build into the application the ability to locate and use, not only the programs that make up the application, but also the data that is specific to a particular project or task within the application.

The overall structure of the directories, naming conventions used within the application, and the location of libraries and files all play pivotal roles in determining the success and maintainability of your application. You need to be able to create a logical directory structure that will be reusable for each project or task. Since many of the programs and macros used within the application will require dynamic project-specific information, naming conventions must be established and strictly adhered to. The location of data, both

project-specific and data that is general to the overall application, must be specified and consistent. These locations usually depend heavily on the directory structure.

Although the examples shown in this section are all based on applications written for directory-based operating systems (like Windows), the concepts apply equally well to file-based systems. Also, the choices of path and directory structures presented here are specific to the presented application and are not intended to be dictates for your applications.

There is more than one philosophy regarding the appropriate use of folder and directory structure when setting up a project or study. Aside from the all too often encountered, 'put it anywhere'-location does not really matter philosophy known as anarchy, there are four structures that are commonly attempted. These structure types can include the following:

- flat or random (no discernable subfolder structure)
- data
- task
- project

Regardless of which variation of structure you find to be best for you, prior planning is essential. Before implementation, planning is important. Fortunately, if you plan carefully, the addition of new subfolders for specific tasks is generally easily accommodated.

Which to choose? The determining factor as to which type of structure will be chosen will be based on where in the structure hierarchy the primary or common element resides. Because I use it the most often, only the project structure is discussed in detail below. More information on the other structures can be found in Carpenter and Smith (2001).

## **Flat Structures**

Flat structures minimize subdirectories by placing most data and program folders on the same level. While this structure is the least complex (nothing is hidden in subfolders), it is not really suitable for anything other than simple projects and tasks. Because all folders for all projects are on the same level, individual projects might be harder to organize. Simple projects that make use of this structure tend to not be dynamic and probably do not need to take advantage of the concepts discussed in this chapter.

## **Data Structure**

When a large common database is established to be used by a number of projects or tasks, the data sets themselves might drive the structure making it higher in the file hierarchy. In this scheme the programs for all projects and tasks reside in directories under the type of data with which they are associated.

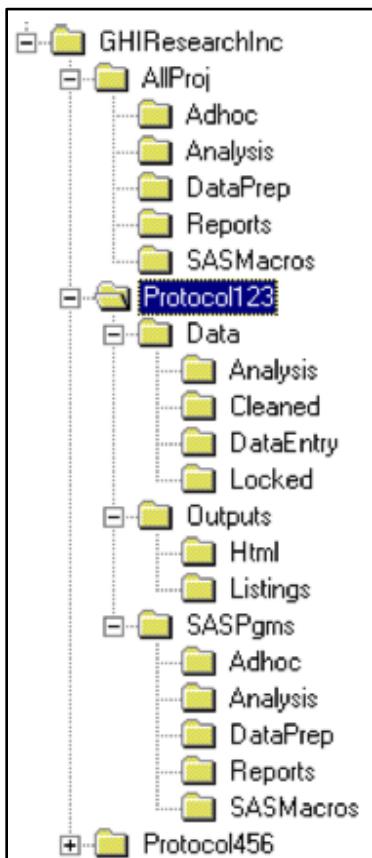
## **Task Structures**

Task-oriented structures are used when the task or report is foremost. This structure might be employed when a common task is applied across projects and for a variety of data sets. The task itself is the constant.

In this type of structure the data from different studies or projects may or may not share a common subdirectory. Usually this structure is used when the data sets are very similar and the analysis and data prep programs will work for any study under consideration.

## **Project Structure**

Since many of the applications that I have installed are project-based, each with their own independent data sets and analysis programs, I most commonly employ a project-oriented structure. This gives me the most flexibility when establishing applications that need to be portable and dynamic. The structure follows a hierarchy with the project highest in the directory tree. Typically everything needed for a given project is located in a subfolder to that project. This includes data and output, as well as the programs used in that task or project. Figure 11.9.2 shows a typical directory structure for a Windows OS project.

**Figure 11.9.2: Project oriented directory structure**

It is not essential that you use either this particular structure or these directory names. What is very important is for you to think about what names and structure will work best for you. For instance my \SASPgms folder often also contains a \Control folder to hold setup programs that create formats, user-defined functions, and the autoexec program, and under the \Data folder I will often have a \Control folder to hold the project's control files (see Section 11.7.2).

As you develop your application, you will probably find that some programs, especially macro tools, are used by more than one project. Rather than copy these tools for each project, you can locate them in the \AllProj folder, which is made available to all the projects by including it in the autocall macro libraries. The autocall library will include the project-specific \SASMacros subfolder as well as the \AllProj\SASMacros directory.

Because all information specific to a project is stored under one folder, it is very easy to move, archive, and locate information on that project. When the same structure is used consistently for each project, it becomes fairly easy to locate specific information across all projects, such as the AE data in the \Data\Analysis subdirectory for all projects.

When using a structure such as the one shown in Table 11.9.2 all the directories and subdirectories for a project have a common root portion of the path. You can take advantage of this by specifying a global macro variable to hold the path. Specification and use of this macro variable is discussed in Section 11.9.3.

**SEE ALSO:** Carpenter and Smith (2001) discusses alternate directory structures.

### 11.9.3 Using the AUTOEXEC File

The autoexec program is an ordinary SAS program; it can contain DATA steps, macro calls, and macro variable definitions just as can any other program. Typically this program is named AUTOEXEC.SAS and is used to define the environment of the current SAS session. By default whenever the SAS System is started, it searches for a program called AUTOEXEC.SAS in the !SASROOT directory, and when present, this program is automatically executed.

Actually you can customize this process by renaming the program and/or by placing it in some other directory. Rather than change the name, it is more straightforward to change its location to a project or task-specific location. SAS must be pointed to the new location, and under Windows operating systems, this is accomplished through the use of the -AUTOEXEC option that is invoked at system initialization. This option is demonstrated in the example dealing with creating shortcuts below. For other operating systems, you should consult your SAS Companion for specific details.

A portion of a sample AUTOEXEC.SAS program is shown in Program 11.9.3. This AUTOEXEC is used to set up the environment for an application that uses the directory structure described in Table 11.9.2.

#### Program 11.9.3: Portion of an AUTOEXEC.SAS

```
%* Establish GLOBAL Macro variables for this application;
%* Set up the general path for this application;
  %global /readonly path=e:\clinical\GHIREsearchINC; ①
%* Project name;
  %global /readonly project = Protocol123; ②
%* Environment - test (tst) or production ( );
  %global /readonly tst =;      /* Production; ③
  /*global /readonly tst = tst;  /* Test environment;

* Define the location for the autocall macro libraries;
options mautosource ④
  sasautos=("&path\&project&tst\SASpgms\sasmacros"
            "&path\allproj\SASpgms\sasmacros");

* Define Librefs and Filerefs for this project;
%libnames ⑤
```

- ① The upper portion of the path is specified in the global read-only macro variable &PATH. When transporting to another system or server, this is the only part of the path that will change.
- ② Since we are using the project directory structure, the project name is included in path information as a separate variable.
- ③ If we want to create a separate environment for testing it is possible to do so by adding a testing ‘switch’.
- ④ The autocall library locations include the use of the three global macro variables.
- ⑤ All LIBNAME and FILENAME statements are located in the %LIBNAMES autocall macro, and each LIBNAME and FILENAME specification uses the three global macro variables used for determining location (see Section 11.9.4).

In this AUTOEXEC three macro variables that will be used throughout the application to determine position are globalized. These are &PATH, &TST, and &PROJECT. The macro variable &PATH is used to declare the top of the directory structure. Since all aspects of the application are beneath this portion of the path, this makes the application portable from machine to machine or from one place on the network to another. The macro variable &TST is used to set up a parallel test/production environment. During production, the value of &TST is set to null. A project-specific name or code is stored in &PROJECT. Together these three macro variables are used to define the full directory structure down to the project level (see Section 11.9.4 for more on their use in LIBNAME and FILENAME statements).

Usually you will want to use a different AUTOEXEC.SAS program for each project or task that you are working on. Under Windows this is easily accomplished by creating a shortcut for each task and by making sure that each shortcut points to its own specific AUTOEXEC program. Creating and using a shortcut under windows is discussed in Section 14.6.1.

**MORE INFORMATION:** Although not available as an option to all macro programmers, some programming environments allow the modification of the SAS configuration file. Section 10.5.4 discusses the modification of the SASAUTOS environmental variable in the configuration file.

Section 14.6.1 looks at the modification of Windows shortcuts.

**SEE ALSO:** Carpenter and Smith (2000 and 2001) discusses the use of the AUTOEXEC and the specification of a flexible path structure.

Carpenter (2012, *Carpenter's Guide to Innovative SAS® Techniques* Sections 14.2 and 14.3) discusses the autoexec and configuration files in more detail.

#### 11.9.4 Unifying *fileref* and *libref* Definitions

In a dynamic application the assignment of *librefs* (and *filerefs*) must be controlled in a unified way. The developer must ensure that *librefs* and *filerefs* are correctly standardized and named so that the application will find the correct data sets and macro libraries. To make the programs as transportable as possible, try very hard to never bury path information instead of *librefs* and *filerefs* within the programs.

Rather than defining a *libref* or *fileref* within a program ‘when you need it’ consider moving the definitions up higher in your program flow. Whenever possible, define the *librefs* and *filerefs* in a single location within the application. A possible location might be a macro such as %LIBNAMES. A portion of this macro is shown in Program 11.9.4.

##### Program 11.9.4: Using %LIBNAMES to Control *librefs* and *filerefs*

```
%macro libnames;
  * Libnames used in applications;
  libname appls ("&path\allproj\SASpgms\appls"
                 "&path\&project&tst\SASpgms\appls")
                 access=readonly;

  * Filenames;
  filename dbdirsa " &path\allproj\SASpgms\primary\dedsn.sas";
  filename vrdirsas " &path\allproj\SASpgms\primary\devar.sas";
  filename cmprlog  " &path\&project&tst\list\compare\compare.log";
  filename cmprlst   " &path\&project&tst\list\compare\compare.lst";

  * Primary Project libnames;
  libname coded "&path\&project&tst\dictionary\live\coded";
  libname notcoded "&path\&project&tst\dictionary\live\notcoded";
  libname editlog "&path\&project&tst\data\live\editlog";
%mend libnames;
```

Notice that each of the paths uses the global &PATH, &PROJECT, and &TST macro variables, which were introduced and discussed in Section 11.9.3. The directory structure shown in this macro is loosely based on the same project directory structure shown in Table 11.9.2; however, it also uses additional directories and subdirectories.

When this macro is called from within the AUTOEXEC program (❸ in Program 11.9.3), all your *librefs* and *filerefs* will automatically be available to all the macros and programs within your dynamic application.

# **Chapter 12: Examples of Dynamic Programs**

<b>12.1 File Management.....</b>	<b>335</b>
12.1.1 Copy an Unknown Number of Catalogs .....	336
12.1.2 Appending Unknown Data Sets.....	336
<b>12.2 Controlling Output.....</b>	<b>342</b>
12.2.1 Coordinating Titles (or Footnotes) .....	342
12.2.2 Auto Display of ODS Styles.....	344
12.2.3 Consolidating ODS OUTPUT Destination Data Sets.....	345
<b>12.3 Adapting Your SAS Environment .....</b>	<b>346</b>
12.3.1 Maintaining System Options .....	346
12.3.2 Building and Maintaining Formats.....	347
12.3.3 Working with Libraries and Directories .....	350
<b>12.4 Working with Data Sets and Variables.....</b>	<b>351</b>
12.4.1 Splitting a Data Set Vertically .....	352
12.4.2 Creating a List of Variable Names from Procedure Output .....	353
12.4.3 Parsing Individual Values from an Existing Horizontal List.....	360
12.4.4 Placing Commas between Words .....	364
12.4.5 Quoting Words in a List.....	365
12.4.6 Checking for Existence of Variables .....	366
12.4.7 Removing Repeated Words from a List.....	367
12.4.8 Controlled Data Corrections and Manipulations .....	369

In Chapter 11 a number of techniques used to write dynamic macros and applications were discussed. The examples used to illustrate those techniques were designed to explain and were not always of practical value. Chapter 12 presents additional examples of these same techniques, from the perspective of functional macros and macro tools.

For the examples in this chapter and indeed for all of the code examples throughout the book, if you want to execute these sample programs, then be sure to follow the setup instructions. Remember that all of the data sets and programs are available for download, so you do not need to retype either the code or the data. For instructions on accessing and setting up the programs and data, see the “Example Code and Data” section within this edition’s “About This Book” front matter.

---

## **12.1 File Management**

The macros in this section perform operations on files (catalogs and data sets) rather than on lines of code. This means that in each case, you need to be able to gather the appropriate information on the operating system (for example, filenames) and create macro and data set variables. SAS gathers a great deal of the system information for us and we can access it in a number of ways (see Section 11.2).

**SEE ALSO:** Geary (1997) discusses a macro that produces summary information on a series of SAS data sets.

A disk space utilization macro is presented in Mast (1997).

A series of directories are built using a macro presented by Dynder, Cohen, and Cunningham (2000).

Under Windows operating systems, Dynamic Link Libraries (DLLs) are often used for operating system operations. Roper (2001) discusses the use of Win32APIs to access DLLs.

### 12.1.1 Copy an Unknown Number of Catalogs

The macro %CATCOPY in Program 12.1.1 copies catalogs found in the TEST environment into a production area. You do not need to know the names of the catalogs prior to execution. However, a filter is available to select catalogs that begin with certain types of names.

#### Program 12.1.1: Copy an Unknown Number of Catalogs

```
%macro catcopy(test=,prod=);
/* test - libref for the test area
 * prod - libref for the production area
 */;

* Determine catalogs in TEST area;
proc sql noprint;
  select memname
    into :cname1-
      from sashelp.vscatlg ①
        where libname="%upcase(&test)"
          and substr(memname,1,2) in ('WE', 'DE', 'PH'); ②
  %let catcnt = &sqllobs; ③
  quit;

proc datasets nolist;
  copy in=&test out=&prod memtype=catalog;
  select
    %do i = 1 %to &catcnt;
      &&cname&i ④
    %end;
  ;
  quit;
%mend catcopy;

/*%catcopy(test=tstappls,prod=appls) */
%catcopy(test=sashelp,prod=work) ⑤
```

- ① The SASHELP.VSCATLG view has one row per catalog.
- ② Subset the list of catalogs by *libref*(&TEST) and the first two letters of the catalog name.
- ③ Save the number of catalogs that are to be copied.
- ④ Build the SELECT statement from the inside out, where the catalog names are stored in the macro list &&CNAME&I.
- ⑤ Call the macro using dummy libraries for testing purposes.

**SEE ALSO:** The Technical Support section of *SAS Communications*, Volume xxii, 4th Qtr. 96 (p. 43) has a similar example that uses the BUILD procedure to build the new catalogs with modifications.

Rook and Yeh (2001) discuss several alternative methods for concatenating and copying members of catalogs.

### 12.1.2 Appending Unknown Data Sets

This macro is taken from an application where a separate data set was created for each subject in the study. The data sets each contained subject-specific information including various data entry and edit status

indicators. In order to create a unified subject status report, it was necessary to combine these individual data sets by concatenating them into one. The problem, of course, was that the number of subjects and the associated code (used to name the data set) was constantly changing, and therefore a dynamic solution was required.

There are several ways that we can use to determine the names of the data sets that meet the selection criteria. These include the use of the following:

- the view SASHELP.VTABLE
- the SQL DICTIONARY table TABLES
- PROC CONTENTS to create a data set
- operation system utilities through the X statement
- the PIPE device type on the FILENAME statement.

The techniques to use each of these information sources are discussed in the examples below. These techniques do not all work equally well (the difference is primarily in performance, but includes ease of coding as well), so the discussion includes the advantages and disadvantages of each.

You can create sample data sets for use with the examples in this section by executing Program 12.1.2BuildData.

## Using SASHELP.VTABLE

Program 12.1.2a determines the list of data sets using the SASHELP.VTABLE view, which has one row per data set per library currently available to SAS. The macro then places the data set names into a series of macro variables.

### Program 12.1.2a: Using SASHELP.VTABLE to Generate a List of Data Set Names

```
%macro appndsn(lib=DECNTRL);
%local i;
* Determine the data sets, make a macro var for each;
* use data set of the form &lib..INxxxxxx ;
data _null_;
  set sashelp.vtable(keep=libname memname) end=eof; ①
  where libname="%upcase(&lib)" & memname='IN'; ②
  call symputx(catt('dsn',_n_),memname,'1'); ③
  if eof then call symputx('dsncnt',_n_,'1'); ④
run;

proc datasets library=work nolist;
  delete alldsn;

  * Append the data sets;
  %do i = 1 %to &dsncnt; ⑤
    append base=alldsn data=&lib..&&dsn&i;
  %end;
  quit;
%mend appndsn;
%appndsn(lib=WORK)
```

- ① The SASHELP.VTABLE view is used to create a list of all of the data sets in the application.
- ② The WHERE statement subsets the entries to those data sets of interest, such as data sets that start with the letters IN in the &LIB library.
- ③ A macro variable of the form &DSN1, &DSN2, and so on, is created for each data set.
- ④ The number of data sets is counted.
- ⑤ The selected data sets are combined using the APPEND statement in PROC DATASETS. This is more efficient than dynamically building a SET statement, as was done in Program 5.3.2b, and requires fewer data sets to be open at any given time.

There is a difference between SASHELP.VTABLE and SASHELP.VSTABLE. The latter contains only the member and library information, and either could be used in this macro. When either the number of data sets or the number of libraries available to SAS becomes large, the construction of this view in the DATA step can become time-consuming. This will tend to be the slowest of the alternative approaches shown in this section.

## Using DICTIONARY.TABLES

The SQL table DICTIONARY.TABLES contains similar information to SASHELP.VTABLE (one row per data set); however, the table itself is generally constructed faster than the SASHELP view. In Program 12.1.2b the DATA step used in Program 12.1.2a is replaced by an SQL step, which draws its information from DICTIONARY.TABLES.

### Program 12.1.2b: Using DICTIONARY.TABLES to Build a List of Data Sets

```
%macro appndsn(lib=DECNTRL);
%local i;
* Determine the data sets, make a macro var for each;
* use data set of the form &lib..INxxxxxx ;
proc sql noprint;
  select memname
    into :dsn1-
      from dictionary.tables 6
        where libname="%upcase(&lib)" &
              substr(memname,1,2)='IN';
  %let dsnct=&sqlbols;
  quit;

. . . . remainder of the macro is not shown . . .

%appndsn(lib=WORK)
```

- 6** DICTIONARY.TABLES is used to generate the list of data sets that meet the inclusion criteria.

## Using PROC CONTENTS

Because the SASHELP.VTABLE is a view, it must be created each time it is called. Because it always builds a list of all members of all established *librefs* (the full list is created before the WHERE clause is applied), building the list can be a time-consuming process. A PROC CONTENTS step with the OUT= and NOPRINT options will be a faster alternative to build this list.

In Program 12.1.2c the OUT= option on the PROC CONTENTS statement will create a data set with one observation per variable per data set. The macro variable list is then built using an SQL step.

### Program 12.1.2c: Using CONTENTS to Build the List of Data Sets

```
%macro appndsn(lib=DECNTRL);
%local i;
* ALLCONT will have one observation for each variable in
* each data set in the &LIB library;
proc contents data=&lib.._all_ 7
  out=allcont(keep=memname
               where=(memname=: 'IN'))
  noprint;
run;

* Build the list of unique data sets;
proc sql noprint;
  select distinct memname 8
    into :dsn1-
      from allcont;
```

```
%let dsncnt=&sqlobs;
quit;

. . . . remainder of the macro is not shown . . . .
```

- ⑦ PROC CONTENTS will be applied to all the data sets in the &LIB library. The outgoing data set, ALLCONT, will have multiple observations per data set.
- ⑧ The DISTINCT option causes SQL to place only unique values of MEMNAME into the macro variables.

Using PROC CONTENTS becomes increasingly faster than SASHELP.VTABLE as the number of data sets increases. However, even PROC CONTENTS might become too slow as the number of data sets increases even more. If the PROC CONTENTS approach becomes too slow, then you might consider using operating-system-level commands, which can be specified by using the X statement.

## Using the X Statement

Many directory-based operating systems have operating-system-level commands that can be used to construct a list of files in a directory. In Program 12.1.2d the PROC CONTENTS step is replaced with an X statement that contains the operating system DIR command (for Windows based systems).

### Program 12.1.2d: Using the X Statement to Create a List of Files

```
%macro appn.dds (lib=DECNTRL);
%local i depath;

* Create a list of all data sets in the &LIB library;
%let depath = %sysfunc(pathname(&lib)); ①
x dir "&depath\in*.sas7bdat" /o:n /b > "&depath\dirhold.txt"; ②

data _null_;
  infile "&depath\dirhold.txt" truncover end=eof; ③
  input memname $20.; ④
  name = scan(memname,1,'.');// ⑤
  call symputx(catt('dsn',_n_),name,'l');// ⑥
  if eof then call symputx('dsncnt',_n_,'l');// ⑦
run;

. . . . remainder of the macro is not shown . . . .
```

- ① Determine the path of the *libref* of interest and store it in &DEPATH.
- ② The X statement is used with the system's DIR command (Windows O/S) to build a list of files in the &DEPATH directory. This list is routed to a text file (&DEPATH\DIRHOLD.TXT), where it is stored for later use.

DIR is a DOS command that is recognized in the Windows sub-session initiated by the X statement. This command is controlled by switches (/o:n /b >)

/o:n	list files in alphabetic order
/b	bare format eliminates heading and summary information from the list
>	route the output to the file that follows

Although not shown in the code above, the NOXWAIT system option is usually used to automatically close the command window opened by the X statement.

```
options noxwait;
%appn.dds (lib=WORK)
```

- ❸ The incoming text file is identified using the INFILE statement. Because the lengths of the data set names might not be constant, the TRUNCOVER option is used.
- ❹ The name of the data set file is read from the text file named on the INFILE statement. The data set filename will be of the form IN1234.SAS7BDAT.
- ❺ The filename portion is retrieved using the SCAN function.
- ❻ The macro variable list is constructed one data set name at a time.
- ❼ The total data set count is saved after reading the last data set name.

The use of the operating system to build the list of data sets is substantially faster than the PROC CONTENTS. This makes sense as the management of files is what the O/S is designed to do.

## Using the PIPE Device Type

The FILENAME statement can be used to point to more than *just* a file. It also supports a *device type* option that enables the user to specify the access method to be used when reading from the specified file(s). One of these device types is PIPE, which enables the user to pass operating system commands directly through the FILENAME statement to the operating system. SAS can then, in effect, use the *fileref* to directly point to the result of the command. Essentially a virtual file is created rather than an actual file as was done with the X statement in Program 12.1.2d.

Program 12.1.2e uses an X statement to build a file containing the list of data sets of interest. This file is then read as data using an INFILE statement. The PIPE device type on the FILENAME statement can be used to combine these two operations. In Program 12.1.2e the list of the data sets is gathered and held in memory.

### Program 12.1.2e: Using the PIPE Device Type on a FILENAME Statement

```
%macro appndsn(lib=DECNTRL);
%local i depath;

* Create a list of all data sets in the &LIB library;
%let depath = %sysfunc(pathname(&lib));
filename list pipe ❸
    %unquote(%bquote(')dir "&depath\in*.sas7bdat" /o:n /b %bquote('));

data _null_;
  infile list truncover ❹
      end=eof;
  input memname $20.;

. . . . remainder of the macro is not shown . . . .

```

- ❸ The PIPE device type routes the results of the DIR command to a virtual file that can be accessed through the use of the LIST *fileref*. The %BQUOTE function is needed to mask the single quotes so that the macro variable &DEPATH can be resolved. Unlike in the X statement, shown in Program 12.1.2d, the macro quoting must be removed prior to the execution of the FILENAME statement. This is accomplished with the %UNQUOTE function.

When using SAS 9.4 or later, you can simplify the FILENAME statement by using the autocall macro %TSLIT.

```
filename list pipe %tslit(dir "&depath\in*.sas7bdat" /o:n /b );
```

- ❹ The LIST *fileref* is used on the INFILE statement to access the virtual list of data set names.

**SEE ALSO:** Sisson (2016) uses the PIPE engine and the X statement to make directories and to move files.

## Using DATA Step Functions

You can also build a list of files in a directory directly by using DATA step functions. In Program 12.1.2f directory information functions are used to read the names of data set names contained in a specified library.

### Program 12.1.2f: Using DATA Step Functions to Read Names from a Directory

```
%macro appnndsn(lib=DECNTRL);
%local filrf rc did memcount i dsn;
* Determine the data set names of the form &lib..INxxxxxx ;

%let filrf=mydata; ①
%let rc=%sysfunc(filename(filrf, %sysfunc(pathname(&lib)))); ②
%let did=%sysfunc(dopen(&filrf)); ③

%let memcount=%sysfunc(dnum(&did)); ④
%if &memcount > 0 %then %do;

  proc datasets library=work nolist;
    delete alldsn;

    /* Append the data sets;
    %do i = 1 %to &memcount; ⑤
      %let dsn=%sysfunc(dread(&did, &i)); ⑥
      %if %upcase(%substr(&dsn,1,2))=IN
          and %scan(&dsn,2,.)=sas7bdat %then %do; ⑦
          append base=alldsn data=&lib..%scan(&dsn,1,.);
      %end;
    %end;
    quit;

  %end;
%let rc=%sysfunc(aclose(&did)); ⑧
%let rc=%sysfunc(filename(filrf)); ⑨

%mend appnndsn;
%appnndsn(lib=WORK)
```

- ① Create a macro variable to hold the name of a temporary *fileref*.
- ② The FILENAME function's first argument is expected to be a macro variable (named without an ampersand). The temporary name is used to establish the *fileref* to the same location as the *libref* containing the data set names. A *fileref* is needed here, so we can access the directory contents as files instead of data sets.
- ③ The directory is opened and a directory ID number is saved in &DID.
- ④ The DNUM function returns the number of members in a directory.
- ⑤ A %DO loop is used to step through each of the &MEMCOUNT files.
- ⑥ The DREAD function is used to read the name of the &T<sup>h</sup> file.
- ⑦ Filenames that meet the criteria will be used to build an APPEND statement.
- ⑧ Close the directory when we are finished.
- ⑨ Clear and eliminate the temporary *fileref*.

**SEE ALSO:** Widawski (1997a) also uses the X statement to build a list of files. Chen and Gilbert (2002) build a list of SAS programs and place them in a batch file for later execution. Long (2003) builds and concatenates a list of Zip files.

The PIPE device type is used by Mao (2001), Rook and Yeh (2001), and LeBouton and Rice (2000) to build a list of files under UNIX, and both Kelley (2003) and Wang (2003) use it in Windows examples. Johnson (2001) uses it to execute various operating system commands, and Chakravarthy (2003) uses it in a non-macro example.

Heaton-Wright (2003) uses SASHELP.VTABLE to build a macro variable. Troxell and Chen (2003) show examples of the use of both SASHELP.VSTABLE as well as DICTIONARY.TABLES.

Under Windows based operating systems, Dynamic Link Libraries (DLLs) can also be used for operating system operations. Roper (2001) discusses the use of Win32APIs to access DLLs.

Murphy (2003a) builds the list of data sets using the Output Delivery System (ODS).

## 12.2 Controlling Output

Many of your programs are going to generate reports and figures, and there are a number of aspects of these reports that you might need to control dynamically. Remember to start with your information source and then build and use the macro lists. Very often the biggest issue is finding the information source that you can use to control the process.

### 12.2.1 Coordinating Titles (or Footnotes)

At times, you might need to write a macro that will use titles, but you might not know what the next available title number is. Of course, using an arbitrary number could wipe out important titles, so we need to be able to determine which titles have already been defined. There are a couple of ways to do this. The SQL table DICTIONARY.TITLES and the SASHELP.VTITLE view both have one row for each title and footnote that are currently defined. Each has the column TYPE that takes on the values of 'T' and 'F' for titles and footnotes, respectively. Each also has a column for the number of the title or footnote (NUMBER).

The macro %PRINTIT in Program 12.2.1a uses the DICTIONARY.TITLES table to determine the largest title number that is currently in use. This number is stored in the macro variable &T. This value in turn allows us to specify the next available title number.

#### Program 12.2.1a: Title Control Using DICTIONARY.TITLES

```
%macro printit;
%local t u v;
proc sql;
    reset noprint; ①
    select max(number) into :t ②
        from dictionary.titles ③
        where type='T'; ④
    quit;

%if &t > 8 %then %let u=9; /* title loss */ ⑤
%else %let u = %eval(&t + 1);
%let v=%eval(&u+1); ⑥

title&u ⑦ 'First custom title';
title&v 'Second custom title';
proc print data=sashelp.class(obs=5);
run;
title&u; ⑧
%mend printit;
title1 'test of custom titles'; ⑨
%printit
```

- ❶ Rather than use the NOPRINT option on the PROC SQL statement, it is specified on the RESET statement. Either approach is equally effective; however, using NOPRINT on the PROC statement requires less typing.
- ❷ The largest title number currently in use is written to the macro variable &T.
- ❸ DICTIONARY.TITLES is used as the information source. This table has one row per title and footnote.
- ❹ We are only interested in the titles.
- ❺ A decision has to be made when more than 8, of the possible 10 titles, are in use. If they are in use, this program sacrifices titles 9 and 10 for the custom titles. The sacrificed titles could be saved and restored later, but this macro does not do so (see Program 12.2.1b for an alternate approach that does restore these titles).
- ❻ &U and &V are the title numbers for the two custom titles.
- ❼ The custom title numbers are used in TITLE statements.
- ❽ The custom titles are cleared.
- ❾ For this example TITLE1 is in use. Therefore, &U will equal 2.

One of the shortcomings of Program 12.2.1a is that an existing TITLE9 or TITLE10 will be lost. This is overcome in Program 12.2.1b, which uses a DATA \_NULL\_ to obtain the same title information. In this program the information source is the view SASHELP.VTITLE.

#### Program 12.2.1b: Using SASHELP.VTITLE to Retrieve Information on the Titles

```
%macro printit;
%local i t u v ttl9 ttl10;
data _null_;
  set sashelp.vtitle(keep=type number text ❶
                     where=(type='T')) end=eof;
  if number gt 8 then call symputx(catt('ttl',number),text,'l'); ❷
  if eof then do;
    call symputx('t',left(put(number,2.)), 'l'); ❸
  end;
  run;

%if &t > 8 %then %let u=9;
%else %let u = %eval(&t + 1);
%let v=%eval(&u+1);

title&u 'First custom title';
title&v 'Second custom title';
proc print data=sashelp.class(obs=3);
run;
title&u; ❹
%if &t>8 %then %do i=9 %to &t; ❺
  title&i &&ttl&i; ❻
%end;
%mend printit;
title1 'test of custom titles';
/*title9 'my title9';*/ ❻
/*title10 'my title10';*/
%printit
```

- ❶ The view SASHELP.VTITLE contains one line per title and footnote:

#### TYPE

T=title, F=footnote

#### NUMBER

number of the title or footnote

**TEXT**

text of the title or footnote

- ② When the title number is 9 or 10, save the title text (&TTL9 and &TTL10) so that it can be restored after the custom titles are no longer needed.
- ③ Save the largest title number in &T.
- ④ After printing, clear the custom titles.
- ⑤ When TITLE9 or TITLE10 have been overwritten, restore their original values
- ⑥ The original values have been stored in &TTL9 and &TTL10.
- ⑦ Dummy title statements used for testing.

**MORE INFORMATION:** A next available title is also determined in the example in Section 11.2.1 through the use of an SQL step and the SASHELP.VTITLE view.

## 12.2.2 Auto Display of ODS Styles

When you want to sample available ODS styles, including customized styles, it can be quite bothersome to try them out one at a time. Program 12.2.2 looks up all available styles and generates a standard report using the EXCELXP tagset and each of the styles.

### Program 12.2.2: Automatically Examining Each of the Available ODS Styles

```

title1 '12.2.2 Showing Styles in Excel';
%macro showstyles;
%local i stylecnt;
proc sql noprint;
  select scan(style,2,'.')
    into: style1-
      from sashelp.vstyle ①
        where scan(style,1,'.')='Styles';
  %let stylecnt = &sqllobs; ②
  quit;

%do i = 1 %to &stylecnt; ③
ods tagsets.excelxp ④
  path="&path\chapter 12\Results"
  file="&&style&i...xls" ⑤
  style=&&style&i ⑥
  options(sheet_name="&&style&i" ⑦
    embedded_titles='yes'
    );
  title2 "Using the &&style&i Style";
  proc report data=sashelp.class;
    column sex name age height weight;
    define sex      / order;
    define name     / display;
    define age      / analysis mean f=4.1;
    define height   / analysis mean f=4.1;
    define weight   / analysis mean f=5.1;
    break after sex / summarize suppress;
    rbreak after /summarize;
    run;
  %end;
  ods tagsets.excelxp close;
%mend showstyles;

%showstyles

```

- ❶ The names of the available styles are stored in the SASHELP.VSTYLE view. The macro variable list &&STYLE&I contains these names.
- ❷ &STYLECNT holds the number of available styles.
- ❸ A %DO loop is used to step through the members of the list.
- ❹ The EXCELXP tagset is used to generate a series of spreadsheets, one for each style. The location of the spreadsheet will depend on the file designation used with the FILE= option. This example is coded to work with the example Windows directory structure that accompanies this book (see “Example Code and Data” in the “About This Book” section at beginning of this book).
- ❺ Each spreadsheet is named using the style name.
- ❻ The style itself is specified using the STYLE= option.
- ❼ Although not really needed here, since each workbook contains a single sheet, the sheet is also named using the name of the style.

It would be nice to be able to write the table for each style to its own sheet rather than to separate files; unfortunately, because of the way that the style information is written to the spreadsheet, this is not possible through ODS.

**SEE ALSO:** A similar example is discussed in Carpenter, 2012, *Carpenter’s Innovative SAS Tips and Techniques*, Section 11.2.3.

### 12.2.3 Consolidating ODS OUTPUT Destination Data Sets

When multiple data sets are created using the ODS OUTPUT destination, they can be dynamically consolidated by collecting the names of the individual data sets in a macro variable. Typically the ODS OUTPUT statement applies to the procedure immediately following the statement; however, it can apply across multiple procedure steps by using the PERSIST option. Usually when we use the OUTPUT destination with a procedure, we create a single data set; however, when a BY statement is used, a separate data set is created for each iteration of the BY statement. In both of these cases we might prefer to have a single data set rather than a series of data sets.

In Program 12.2.3 we would like to consolidate the data sets generated by two separate PROC UNIVARIATE steps, each with a different BY variable. Since a data set will be generated for each BY variable level, for each of the PROC steps, we do not know how many data sets will be created. The MATCH\_ALL= option can be used to create a macro variable to hold the names of all of the data sets generated by the ODS OUTPUT statement.

#### Program 12.2.3: Creating a List of Data Set Names with MATCH\_ALL

```
ods output Moments( ❶
                     match_all=namelist ❷
                     persist=proc❸)=work.Moments;❹

proc sort data=sashelp.class out=class;
  by age;
  run;
proc univariate data=class;
  by age; ❺
  var weight;
  run;

proc sort data=sashelp.class out=class;
  by sex;
  run;
proc univariate data=class;
  by sex; ❻
  var weight;
  run;
ods output close;
```

```
%put &=namelist;
data allmoments;
  set &namelist;❷
run;
```

- ❶ The MOMENTS generated by the UNIVARIATE procedure are requested to be written to a SAS data set named WORK.MOMENTS ❹.
- ❷ The MATCH\_ALL= option names a macro variable (&NAMELIST) that will contain the names of the data sets generated by this ODS OUTPUT statement.
- ❸ The ODS OUTPUT statement is to persist across PROC step boundaries, until it is closed.
- ❹ The data set to be created by the ODS OUTPUT statement is to be written to WORK.MOMENTS. If more than one data set is generated, successive data sets will be named WORK.MOMENTS1, WORK.MOMENTS2, and so on. Because of the persistence level and the use of the BY statements, it is anticipated that multiple data sets will be generated.
- ❺ A separate data set will be generated for each level of AGE.
- ❻ A separate data set will be generated for each level of SEX.
- ❼ The macro variable &NAMELIST will contain the list of all data sets generated while the ODS OUTPUT statement is active, and this list is then used in a SET statement to append the data sets.

```
601 %put &=namelist;
NAMELIST=WORK.MOMENTS WORK.MOMENTS1 WORK.MOMENTS2 WORK.MOMENTS3
WORK.MOMENTS4 WORK.MOMENTS5 WORK.MOMENTS6 WORK.MOMENTS7 ❽
```

## 12.3 Adapting Your SAS Environment

As we write macros that are more sophisticated, we often need for them to be able to detect and work with the environment in which they run. They might need to change options and then later restore the original values. SAS accesses locations on the operating system through *librefs* and *filerefs* and these points of reference are often maintained from within the macro language. When formats are stored permanently, the catalog that contains them must be made available. The process of creating formats, saving the formats, and making them available can also be controlled through the macro language.

### 12.3.1 Maintaining System Options

It is not unusual for you to need to write a macro that either needs to determine the current value of a system option or needs to reset an option to a previous value. In either case, your macro will need to be able to access the list of current OPTION settings. There are a couple of ways to do this: the GETOPTION function (Program 11.2.4) and SASHELP.VOPTION.

The data view SASHELP.VOPTION is used to determine the current option settings. This view takes the form of one observation per option with the columns of OPTNAME and SETTING holding the option name and value, respectively. In Program 12.3.1 the macro %STOREOPT creates a list of global macro variables that will contain the current option settings of interest.

#### Program 12.3.1: Store Selected Option Values Retrieved from SASHELP.VOPTION

```
%macro storeopt(oplist);
%local tem;
%if &oplist ne %then %do;
  %let oplist = %cmpres(&oplist); ❶

  /* Quote each option name individually;
  %let tem = %sysfunc(quote(&oplist)); ❷
  %let tem = %qsysfunc(tranwrd(&tem,%str( ),%bquote(,"))); ❸

  * Retrieve the current option settings;
  data _null_;
```

```

set sashelp.voption;
if optname in: (%upcase(&tem)) then do; ❸
    call symputx(optname,setting,'g'); ❹
end;
run;
%end;
%mend storeopt;
%storeopt(linesize pagesize obs    mlogic date) ❺

```

- ❶ Extra spaces are removed from the list of options.
- ❷ The option names are going to be used with an IN operator, so they must be quoted. Double quotes are added before and after the list of options.
- ❸ The TRANWRD function is used to convert each space between words to a " , " . For the macro call shown with this example (see ❻) the value of &TEM is now "linesize", "pagesize", "obs", "mlogic", "date"
- ❹ The IN: comparison operator is used to determine whether the current option that has been read in from SASHELP.VOPTION is in the list of options in &OPLIST. This comparison requires the words to be quoted. The %UPCASE function not only converts to uppercase, but it also removes the macro quoting which in turn forces &TEM to be resolved before the comparison is compiled.
- ❺ A global macro variable is created with the name of the option and the option's value.
- ❻ The macro %STOREOPT is called with a list of option names.

Once the option values have been collected, they can be restored after they are subsequently modified. To restore these option settings, one would use the following OPTION statement:

```
options ls=&linesize ps=&pagesize obs=&obs &mlogic &date;
```

In %STOREOPT the SASHELP.VOPTION view must be created and read each time the macro is called. Also, it does not contain any information about the form of the option (keyword or on/off), nor does it contain information about the settings of the graphics options (controlled through the GOPTIONS statement). These shortcomings can be rectified through the use of the GETOPTION DATA step function, which is discussed in the macro %HOLDOPT in Program 11.2.4.

**SEE ALSO:** You can also save and restore system options through the use of procedures (OPTSAVE and OPTLOAD), and in the windowing environment through the use of Display Manager commands (DMOPTSAVE and DMOPTLOAD).

### 12.3.2 Building and Maintaining Formats

Customized formats, which can take on a variety of forms, can be built with PROC FORMAT using several different approaches. They can be built by hand using procedure statements, such as the VALUE statement, or they can be based on relationships in a SAS data set. Once created, these formats can be stored temporarily or permanently, and when stored permanently, the FMTSEARCH option can be used to designate the various format catalogs that can be searched for the customized formats. The macros in this section work with this process.

#### Checking the Format Search Path

The list of libraries for SAS to search, when looking for catalogs that might hold user-defined format definitions, is set using the FMTSEARCH system option. The macro %FMTSRCH in Program 12.3.2a checks whether a specified library is on the search path. If it is not, then the library can optionally be added.

**Program 12.3.2a: Checking and Adding Libraries to the Format Search Path**

```
%macro fmtsrch(lib=macro3,add=);
  %local optval insrch;
  /* Check to see if a libref is in the format search path
  /* lib   libref to check
  /* add   If &add is not null, then add the libref, if
  /*       it is not already in the fmtsearch value
  /*
  /* Macro actions:
  /* writes &INSRCH to the LOG
  /*   0 if &LIB is not on path
  /*   >0 if &LIB is on path
  /* &ADD is not null & &LIB is not in search path:
  /*     OPTION statement is written to add &LIB to FMTSEARCH;

  /* Determine if the library is on the fmt search path;
  %let optval = %sysfunc(getoption(fmtsearch)); ①
  %*put &optval;
  %let insrch = %index(&optval,%upcase(&lib)); ②
  %put &=insrch;

  %if &insrch = 0 & &add ne %then %do; ③
    /* Add &lib to format search,
    /* Remove the trailing close parenthesis;
    %let optval =
      %substr(&optval,1,%eval(%length(&optval)-1)); ④
    /* Add the library;
    options fmtsearch=&optval &lib); ⑤
  %end;
%mend fmtsrch;
```

- ① Retrieve the current value of the FMTSEARCH option. The returned value is enclosed in parentheses. Depending on how the FMTSEARCH option has been specified, the value retrieved by GETOPTION might not include the default WORK and LIBRARY *librefs*.
- ② Check if &LIB is currently on the list of libraries to be searched (&INSRCH is > 0 if &LIB is found and 0 if it is not). This approach might not work if a library on the FMTSEARCH path is a concatenated library.
- ③ When &ADD is not null and &LIB is not already on the path, we want to add &LIB to the search path.
- ④ The value returned by the GETOPTION function is surrounded by parentheses. The closing parenthesis is removed from &OPTVAL in preparation for appending the new library in &LIB.
- ⑤ The OPTIONS statement that will rebuild the format search path is specified. &OPTVAL already has the open parenthesis. The closing parenthesis is added here.

When &ADD is null, the macro returns the result of its check by writing to the SAS Log (0 if &LIB is not in the path and a value > 0 if &LIB is in the format search path). The following SAS Log shows three different calls to the %FMTSRCH macro:

```
73  %fmtsrch(lib=work,add=)
INSRCH=2 ⑥
74  %put %sysfunc(getoption(fmtsearch));
(WORK LIBRARY)
75  %fmtsrch(lib=macro3,add=)
INSRCH=0 ⑦
76  %put %sysfunc(getoption(fmtsearch));
(WORK LIBRARY)
77  %fmtsrch(lib=macro3,add=x)
INSRCH=0 ⑧
78  %put %sysfunc(getoption(fmtsearch));
(WORK LIBRARY MACRO3)
```

- ❶ &INSRCH indicates that the WORK library is on the format search path. This is verified with the subsequent display of the current value of the FMTSEARCH option.
- ❷ The MACRO3 library is not on the format search path.
- ❸ Because the &ADD parameter is not blank, MACRO3 is to be added to the format search path.

## Building a Format from a Data Set

When a data set contains two or more paired columns, it can often be advantageous to use them to build a format. Preparation of the data set for use by PROC FORMAT is not difficult, but it requires the user to know the various options and variable names used by the procedure. The macro %MKFMT in Program 12.3.2b performs this conversion and builds the format. It also uses the %FMTSRCH macro (Program 12.3.2a):

### Program 12.3.2b: Creating a Format in a Specified Library

```
%macro mkfmt(lib=, dsn=, fmtname=, from=, too=);

  /* When the fmt name is unspecified,
   * use the incoming var name (FROM) as the format name;
  %if &fmtname= %then %let fmtname=&from; ❶

  * Eliminate duplicate values of &from;
  proc sort data=&dsn(keep=&from &too)
    out=temp
    nodupkey;
  by &from;
  run;

  data control(keep=fmtname start label type); ❷
    set temp(rename=(&from=start &too=label));
    length type $1 fmtname $18;
    retain fmtname "&fmtname" type ' ';
    * Determine the format type;
    If _n_=1 then type = vtype(start); ❸
    run;

  proc format
    %if &lib ne %then library=&lib; ❹
    cntlin=control;
  run;

  %if &lib ne %then %fmtsrxh(lib=&lib,add=x); ❺
%mend mkfmt;
```

- ❶ When a format name is not provided by the user, the name of the START (&FROM) variable will be used.
- ❷ These variables (at a minimum) are needed by PROC FORMAT:

#### FMTNAME

specifies the name of the format to be created.

#### START

specifies the value that is to be converted.

#### LABEL

specifies the value that START will be converted to.

#### TYPE

specifies a numeric (N) or character (C) format.

While TYPE is not strictly required by PROC FORMAT, it is needed in this macro because character format names will not be written to start with \$.

- ③ The type of format (numeric or character) is determined by the type of the START variable using the VTYPE function.
- ④ When &LIB is specified, the catalog &LIB..FORMATS is used to store the format.
- ⑤ If this library is not already on the format search path it is added using the %FMTSRCH macro (see Program 12.3.2a).

The following call to %MKFMT creates a permanent format in MACRO3.FORMATS using the variables CLINNUM and CLINNAME from the data set MACRO3.CLINICS. The format name will be CLNAME and it will map clinic numbers to clinic names.

```
%mkfmt(lib=macro3,dsn=macro3.clinics,
      from=clinnum,to=clinname,
      fmtname=clname)
```

**SEE ALSO:** Lopez (1998) discusses some macro tools that can be used to dynamically create SAS formats. Pete Lund (2003c) discusses a more sophisticated macro for the management of the FMTSEARCH= option.

### 12.3.3 Working with Libraries and Directories

There are a number of DATA step functions that can be used within the macro language to establish, check, and clear libraries. These functions can be very helpful in dynamic programs that need to explore locations (directories and libraries). These functions include the following:

#### PATHNAME

determines the path or location pointed to by a given *libref* or *fileref*.

#### LIBNAME

establishes or clears a *libref*.

#### LIBREF

determines whether a given *libref* has been established.

#### FILENAME

establishes or clears a *fileref*.

#### FILEREF

determines whether a *fileref* has been established.

The macro %MKLIB shown in Program 12.3.3 can be used to create a library with a supplied engine. It can be passed either the name of an existing library or a physical path. Sometimes you know an existing *libref* name, but not its corresponding location, and you need to establish a second *libref* with a different engine pointing to the same location, or you might know a physical location and need to establish a new *libref*.

#### Program 12.3.3: Establishing a *libref* Based on the Location of a Second *libref*

```
%macro mklib(lname=, engine=V9, lloc=);
%* Establish a libref
%*   lname    new libref name
%*   engine   engine type (optional)
%*   lloc     path or location
%*           may be an existing libref
%*;

%local rc;
%* Clear the libref if it exists;
%if %sysfunc(libref(&lname))=0  %then ①
```

```
%let rc = %sysfunc(libname(&lname)); ②
/* Determine if &lloc is already a libref;
%if %length(&lloc) le 8 and %sysfunc(libref(&lloc))=0 %then ③
    %let lloc = %sysfunc(pathname(&lloc)); ④
%let rc = %sysfunc(libname(&lname,&lloc,&engine)); ⑤
%if &rc ne 0 %then %do; ⑥
    %put &rc %sysfunc(sysmsg());
    %let rc = %sysfunc(libname(&lname));
%end;
%mend mklib;
```

- ➊ &LNAME contains the *libref* that is to be established. The LIBREF function returns a 0 when its argument is an *established libref*. If the *libref* held in &LNAME already exists, this macro will clear it ②.
- ➋ The LIBNAME function clears a *libref* when no other arguments are included.
- ➌ Determine whether &LLOC contains an existing *libref*, which is restricted to 8 or fewer characters. If these conditions are not met, *assume* that &LLOC contains a physical path.
- ➍ For an existing *libref*, the PATHNAME function is used to retrieve its physical location (path). The path replaces the value *in* &LLOC, which will be used to establish the new library ⑤.
- ➎ &LLOC now *contains* a physical path, and the new library (&LNAME) is established.
- ➏ If the library assignment is unsuccessful (&RC ne 0), then we want to write a diagnostic to the SAS Log and clear the *libref*, which would otherwise be established but pointing to a bad location.

The %MKLIB macro allows us to establish a new *libref* based on the location of an existing library or using a physical path.

```
%mklib(lname=mymac6, engine=v6, lloc=macro3) ⑦
%mklib(lname=mymac9, engine=v9, lloc=c:\temp) ⑧
%mklib(lname=myoops, engine=v9, lloc=c:\temp\doesnotexist) ⑨
```

- ➊ The *libref* MYMAC6 will point to the same location as the MACRO3 *libref*, but will be using the V6 engine.
- ➋ The MYMAC9 *libref* will be established for the directory named in &LLOC.
- ➌ This location does not exist and the *libref* MYOOPS will not be established and diagnostics ⑥ will be written to the SAS Log.

**MORE INFORMATION:** The FILEEXIST function is used in the %MAKEDIR macro in Program 8.2.4. The PATHNAME and FILEEXIST functions are also used in the examples of %SYSFUNC in Program 7.4.2d. Program 8.3.2d discusses the use of the SYSMSG function.

Program 12.1.2f uses DATA step functions to read the names of files (data sets) in a directory.

**SEE ALSO:** Michelsen (2014a) and Thorton (2014) both use the LIBREF function to determine the success of a LIBNAME assignment.

## 12.4 Working with Data Sets and Variables

When working with data, the dynamic application will often need to be flexible enough to be applied to any number of different data sets. Since each data set is different, the macro will need to be able to adjust to different variable lists and different types of variables.

When the data sets and variable lists are generated dynamically in such a way that the developer does not necessarily know what the program will be operating on at the time of execution, the use of macros and macro variables is often especially important.

Fortunately there are a variety of techniques that can be applied to data sets and data set variables, and this section highlights some of those techniques.

**SEE ALSO:** Hahl (1996) and SAS Sample 32470 both describe macros that can be used to DROP any variable that only takes on MISSING values.

Whitaker (1989) provides several utilities to work with lists, including ones that; count and extract words, eliminate duplicate words, build lists with SYMPUT, and build lists with %INCLUDE.

## 12.4.1 Splitting a Data Set Vertically

When working with a very wide data set (lots of variables), sometimes it might be helpful to split it into a number of data sets each with some fraction of the original variables. This process is quite straightforward if you know the names of the variables and into which new data set each variable will go. But what if there are too many variables to easily use to build the needed KEEP options? When splitting the data set dynamically, you will still need to know or be able to know the names of the variables, and how many are to be assigned to each of the new data sets that are to be generated. This information must be gathered automatically. In macro %SPLIT (Program 12.4.1) the user determines how many data sets are to be created and the program determines which variables (and how many variables) to allocate to each new data set.

### Program 12.4.1: Splitting a Data Set Vertically

```
%macro split(dsn=, dsnroot=fred, splitcnt=3);
%local dsid vcnt i cn teach j;

%let dsid = %sysfunc(open(&dsn)); ①
%if &dsid ne 0 %then %do;
  %let vcnt = %sysfunc(atrrn(&dsid,nvar)); ②
  %do i = 1 %to &vcnt;
    %local vars&i;
    %let vars&i = %sysfunc(varname(&dsid,&i)); ③
    %*put &&vars&i; /* Debugging;
  %end;
  %let dsid = %sysfunc(close(&dsid)); ④

  /* Nominal number of variables in each new dataset;
  %let cn teach = %sysevalf(&vcnt/&splitcnt,ceil); ⑤

  data
    %do i = 1 %to &splitcnt; ⑥
      &dsnroot&i(keep= ⑦
        %do j= %eval((&i-1)*&cn teach+1) ⑧
          %to %sysfunc(min(&vcnt,%eval(&i*&cn teach))); ⑨
          &&vars&j ⑩
          %*put &=i &=j &&vars&j; /* Debugging;
      %end;
    )
  %end;
  ;
  set &dsn;
  run;
%end;
%mend split;
```

- ❶ Open the data set of interest so that the number and names of the variables can be harvested from the data set's metadata.
- ❷ The number of variables is returned from the ATTRN function, using the NVAR option.
- ❸ In a %DO loop that cycles through the &VCNT variables retrieve the name of the &I<sup>th</sup> variable, which is stored in the macro variable &&VARS&I.
- ❹ The data set is closed so that it can be used in the following DATA step.
- ❺ Calculate how many variables (&CNTEACH) will be in each new data set if the &VCNT variables are divided (approximately equally) into &SPLITCNT data sets.
- ❻ Loop through the new data sets and build the data set name for each. The data set names will include the KEEP= option appropriate for that data set. Each will be of the form:

Fred1 (keep= a b c)

- ❷ &DSNROOT contains the root portion of the name of the new data set. Unless the root name of the data set is otherwise specified, when &I is 3, &DSNROOT&I will resolve to FRED3.
- ❸ The inner loop steps through the variables needed for this data set. The lower value resolves to the first variable number for this data set.
- ❹ Unless the total number of variables is reached sooner (&VCNT), the upper range of the number of variables kept in this data set will be the number held in &CNTEACH more than the lower bound ❸.
- ❽ The individual variable names are written to the KEEP= option.

Turning on MPRINT reveals the DATA step constructed by %SPLIT.

#### **Program 12.4.1 (Portion of the SAS Log): Showing the DATA Step Written by the Macro %SPLIT**

```
1131 options mprint;
1132 %split(dsn=sashelp.shoes, dsnroot=wanted, splitcnt=3)
MPRINT(SPLIT):   data wanted1(keep= Region Product Subsidiary )
wanted2(keep= Stores Sales
Inventory ) wanted3(keep= Returns ) ;
MPRINT(SPLIT):   set sashelp.shoes;
MPRINT(SPLIT):   run;
```

**MORE INFORMATION:** A data set is broken up horizontally by macro %BREAKUP in Section 11.2.7.

**SEE ALSO:** Henderson (2014) uses OPEN and VARNAME to read SAS data set metadata.

---

#### **12.4.2 Creating a List of Variable Names from Procedure Output**

Sometimes it becomes necessary to create a macro variable that contains a list of variable names. When you deal with programs that were written dynamically, you might not know what those names will be. This section shows several different ways to build this list of variable names.

One procedure that commonly creates new variables is PROC TRANSPOSE. When you use the ID statement, the names of the new variables will be the values taken on by the ID variable (or more importantly, some variation of those values). The following examples retrieve a list of variable names and places them in a macro variable.

#### **Building the List in a DATA \_NULL\_ Step**

A DATA step can be used to retrieve the list of variables, and then write the list to a temporary file, which can subsequently be included, using %INCLUDE, and executed. Program 12.4.2a creates a data set with unknown variable names using PROC TRANSPOSE and then uses a DATA \_NULL\_ step to build a %LET statement to create the macro variable.

**Program 12.4.2a: Building the List in the DATA Step**

```
%macro VList(dsn=, bylist=, id=, var=);
%global varlist;
*** create the transposed data set;
proc sort data=&dsn(keep=&bylist &id &var)
    out=srtdata
    nodupkey;
by &bylist &id;
run;
proc transpose data=srtdata
    out=trnsdata; ①
by &bylist;
id &id;
var &var;
run;

* create a space-delimited variable list using a DATA step ;
data _null_;
set sashelp.vcolumn(where=(libname='WORK' &
                           memname='TRNSDATA' & ②
                           name not in:('_NAME_ ' '_LABEL_')));
end=endeof ; ③
file 'c:\temp\temp.sas' lrecl=70 ; ④
if _n_=1 then put '%let varlist= ' @ ; ⑤
put name @ ; ⑥
if eof then put ';' ; ⑦
run ;

%include 'c:\temp\temp.sas' / source2 ; ⑧
%put &varlist;
%mend VList;
%VList(dsn=sashelp.shoes, bylist=region subsidiary, id=product, var=sales)
```

Source: In the second edition of this book, a similar version of this solution was proposed by Kim Kubasek, SAS Consultant.

- ① The data set WORK.TRNSDATA contains a list of unknown variables.
- ② SASHELP.VCOLUMN is used to recover the list of variables for this data set. There will be one row per variable. For the purposes of this macro, we are not interested in the variables \_NAME\_ and \_LABEL\_.
- ③ The END= option is used to detect the last observation that will be read from SASHELP.VCOLUMN.
- ④ The %LET statement that we are building will be written to the file "c:\temp\temp.sas".
- ⑤ Before the first variable name is written to the file, the %LET VARLIST= is written to start the statement.
- ⑥ Each variable name is written using a PUT statement.
- ⑦ When all variable names have been written, the semicolon closing the %LET statement is added.
- ⑧ The %INCLUDE brings the %LET statement back into the SAS program where it is executed. The /SOURCE2 option causes the included content (the %LET statement) to be written to the SAS Log.

The SAS Log shows that the %LET statement is correctly constructed and executed:

```
NOTE: %INCLUDE (level 1) file c:\temp\temp.sas is file c:\temp\temp.sas.
391 +%let varlist= Region Subsidiary Boot Men_s_Casual Men_s_Dress Sandal
392 +Slipper Sport_Shoe Women_s_Casual Women_s_Dress ;
NOTE: %INCLUDE (level 1) ending.
VARLIST=Region Subsidiary Boot Men_s_Casual Men_s_Dress Sandal Slipper
Sport_Shoe Women_s_Casual Women_s_Dress
```

When only a subset of variables is desired, the selection of variables could be simplified by using the PREFIX= option on the PROC TRANSPOSE statement:

```
proc transpose data=srtdata
    out=trnsdata
    prefix=type_
;
```

Each of the variables generated by the TRANSPOSE step will now begin with TYPE\_, and it would be possible to subset for them by using the =: comparison operator.

```
set sashelp.vcolumn(where=(name=:`type_`)) end=eof;
```

**MORE INFORMATION:** Section 11.6 describes in more detail the use of the DATA step to build code that is to be executed using %INCLUDE.

## Using DATA Step Functions to Directly Retrieve the Variable Names

In the DATA step there are a series of functions that enable you to examine various metadata properties, including variable attributes. The following DATA step is designed to fit into the previous example right after the PROC TRANSPOSE. This solution uses the variable STR to accumulate the list of variable names; however, since its length is fixed, there is an implied upper limit as to the number of names that can be processed. This would not be a limitation of the previous method.

### Program 12.4.2b: Retrieving Variable Names Using the VNAME Routine

```
..... portions of %VLIST not shown .. . .

data _null_;
    set trnsdata;
    array allc {*} _character_; ①
    array alln {*} _numeric_; ①
    length __name $32 __str $500;
    if dim(allc) then do __i=1 to dim(allc); ②
        call vname(allc{__i}, __name); ③
        * Exclude vars we know we do not want;
        if __name not in('_NAME_ ' '_LABEL_') then
            __str = catx(' ', __str, __name); ④
    end;
    if dim(alln) then do __i=1 to dim(alln); ②
        call vname(alln{__i}, __name); ③
        __str = catx(' ', __str, __name); ④
    end;
    call symputx('varlist', __str, 'g'); ⑤
    stop; ⑥
    run;

%put &=varlist;
%mend VList;
%VList(dsn=sashelp.shoes, bylist=region subsidiary, id=product, var=sales)
```

- ① Separate arrays are set up for the numeric and character variables. These arrays will not include the variables \_\_STR, \_\_NAME, or \_\_I as the array is populated with the variables on the PDV at that point in the compilation process of the DATA step. These three variables are added to the PDV after the arrays are established. We are assuming that these three variables do not already exist in the incoming data set TRNSDATA.
- ② When the array has one or more elements, a DO loop is used to walk through each of the variables in the respective array.
- ③ The CALL VNAME routine returns the name of the variable on the PDV (identified by the array ALLC{\_\_i}) and places it in the variable \_\_NAME.

- ④ Because the value of the variable `_NAME` is the name of the unknown variable in the array, it is then appended to the variable `_STR`, which accumulates all the variable names.
- ⑤ Once all variable names have been collected, the variable `_STR` is written to the macro variable `&VARLIST` using the `SYMPUTX` routine. Notice that in this example the variable `_STR` is assigned an arbitrary length of \$500, and this might be limiting if the list becomes long. The macro in Program 12.4.2a does not have this limitation because the list of names is not stored in a DATA step variable.
- ⑥ Because the programmer is interested in only the names of the variables and not in the data itself, the `STOP` statement is used to prevent further passes of the data.

### Creating the List of Variables Using an SQL Step

One of the easiest ways to build a macro variable containing a list of variable names is with an SQL step. The SQL step has access to both `SASHHELP.VCOLUMN`s and `DICTIONARY.COLUMN`NS (see Section 11.2.2). Both these tables are built internally when they are requested; however `DICTIONARY.COLUMN`NS will generally load faster and is usually preferred.

#### Program 12.4.2c: Retrieving Variable Names Using a PROC SQL Step

```
      . . . . portions of %VLIST not shown . . . .

proc sql noprint;
  select distinct name ①
    into :varlist separated by ' ' ②
    from dictionary.columns ③
      where (libname='WORK' & memname='TRNSDATA' & ④
              name not in ('_NAME_ ' '_LABEL_ ')); ⑤
  quit;

%put &=varlist;
%mend VList;
```

- ① The variable `NAME` contains the names of the variables. The `DISTINCT` keyword should not be needed, but is included here “just to be sure.” The names might not be unique if multiple data sets were being used.
- ② The macro variable `&VARLIST` will contain an alphabetic space separated list of the variable names.
- ③ `DICTIONARY.COLUMN`NS has one observation per variable per data set per library.
- ④ Select the specific data set of interest.
- ⑤ Exclude variables that are not needed.

By using SQL to directly load the macro variable, the length limitation (of the variable `_STR`) encountered in Program 12.4.2b is avoided.

**MORE INFORMATION:** In Program 11.2.2b an SQL step uses `DICTIONARY.COLUMN`NS to create a macro variable list containing the variables in a SAS data set.

### Pulling Variable Names Using a Macro Function

Because we are not reading any data and because we only need to create a macro variable, we should be able to avoid the use of the `DATA` and `SQL` steps altogether. The macro `%GETVARS` in Program 12.4.2d acts like a macro function and returns the list of variables in a data set (`&DSET`). This macro makes use of the `VARNAME` function to return the variable name.

#### Program 12.4.2d: Using a Macro Function to Retrieve the List of Macro Variables

```
%Macro GetVars(Dset) ;
  %Local VarList ; ①
  /* open dataset */
  %Let FID = %SysFunc(Open(&Dset)) ; ②
```

```

/* If accessable, process contents of dataset */
%If &FID %Then %Do ;
  %Do I=1 %To %SysFunc(ATTRN(&FID,NVARS)) ; ③
    %Let VarList=&VarList %SysFunc(VarName(&FID,&I)); ④
  %End ;
  /* Close dataset when complete */
  %Let FID = %SysFunc(Close(&FID)) ; ⑤
%End ;
  &VarList ⑥
%Mend ;

%macro VList(dsn=, bylist=, id=, var=);
%global varlist;
*** create the transposed data set;
proc sort data=&dsn(keep=&byst &id &var)
  out=srtdat
  nodupkey;
  by &byst &id;
  run;
proc transpose data=srtdat
  out=trnsdata(drop=_name_ _label_); ⑦
  by &byst;
  id &id;
  var &var;
  run;

%let varlist=%getvars(trnsdata); ⑧
%put &varlist;
%mend VList;

```

Source: Macro %GETVARS was written by Michael Bramley, Kendle International, Inc.

The macro %GETVARS will return a list of all the variables in the data set named in &DSET. This list is stored in the local macro variable &VARLIST.

- ① The macro variable &VARLIST is established in the local symbol table.
- ② The data set of interest is opened so that we can access its metadata. &FID holds the identification number for this file. This macro variable could also have been established on the %LOCAL statement.
- ③ The ATTRN function with the NVARS argument returns the number of variables in the data set.
- ④ The VARNAME function returns the name of the &Ith variable. This variable is appended to the growing list of variables stored in &VARLIST.
- ⑤ Once the data set is no longer needed, it is closed.
- ⑥ The list of variables is “left behind” and effectively returned by the macro. Note that &VARLIST is local to the macro %GETVARS.
- ⑦ Rather than exclude the variables that we don’t want later, they can be dropped in the TRANSPOSE step.
- ⑧ The call to %GETVARS resolves to the list of DATA step variables in the data set TRNSDATA. This %LET statement writes this list of variables to the global version of &VARLIST. The following call to %VLIST:

```
%VList(dsn=sashelp.shoes, bylist=region subsidiary, id=product, var=sales)
```

causes the following %LET to be written:

```
%let varlist = Region Subsidiary Boot Men_s_Casual Men_s_Dress Sandal
Slipper Sport_Shoe Women_s_Casual Women_s_Dress;
```

**SEE ALSO:** SAS Sample 25083 shows a macro that builds this list using very similar techniques.

## Using Perl Regular Expressions to Match Variable Names to a Pattern

The %GETVARS macro in Program 12.4.2d returns the list of all of the variables in a data set. A more sophisticated version of the macro, %BUILDVARLIST shown in Program 12.4.2e, enables the user to select only those variable names that match one or more patterns. Not only does his macro use DATA step functions to access the data set's metadata, it also takes advantage of some Perl Regular Expression pattern matching functions and routines.

### Program 12.4.2e: Using Perl Regular Expression Pattern Matching Functions

```
%Macro BuildVarList( Dset, Type, Patterns ) ;
/*
Define local macro variables:
  VarList = list of variables matching pattern(s)
  VarName = name of current variable
  CurPat  = current pattern being used
  FID     = File IDentifier (for FILE I/O functions)
  RX      = Regular expression ID - SAS says "do not PUT...""
*/
%Local VarList VarName CurPat FID RX I J ;

/*
  Check Access to SAS data set.
*/
%Let FID = %SysFunc( Open( &Dset ) ) ;
%If Not &FID %Then %Do ;
  %Put ERROR: BuildVarList could not access &Dset.. ;
  %Go To Exit ;
%End ;

/*
  Check Type and Patterns. Default Type=NC to allow
  all variable types if none
  specified, otherwise validate Type.
*/
%Let Type = %UpCase( &Type ) ; ①

%If "&Type" EQ "" %Then
  %Let Type = NC ;
%Else
  %If Not %Index( NC, &Type ) %Then %Do ;
    %Put ERROR: BuildVarList found invalid variable type of &Type..
Must be blank, N, or C. ;
    %Go To Exit ;
  %End ;

/*
  If Version 6.xx engine, ensure that the Patterns parameter
  is uppercase.
*/
%If %Index( %SysFunc( AttrC( &FID, ENGINE) ), V6 ) %Then
  %Let Patterns = %UpCase( &Patterns ) ; ②

/*
  If no Patterns specified, default to : to match all
  variables in data set.
*/
%If "&Patterns" EQ "" %Then
  %Let Patterns = : ;
```

```

/*
BuildVarList Macro Main Processing Block: process
patterns (separated by | <an or bar>) to generate
variable list by calling SAS regular expression routines.

NOTE: As some regular expressions may begin with SAS
      arithmetic operators, for example, + or *,
      it is imperative to quote them so that the SAS %Eval
      function is not invoked in the %Do %While Loop.

*/
%Let I = 1 ;
%Let CurPat = %Scan( &Patterns, &I, | ) ;

%Do %While( "&CurPat" NE "" ) ; ③
  %Let RX = %SysFunc( RXParse( &CurPat ) ) ; ④
  /*
  If pattern accepted by parser, then process variables
  in SAS data set.
  */
  %If &RX %Then %Do ;
    %Do J = 1 %To %SysFunc( AttrN( &FID, NVARS ) ) ; ⑤
      /*
      If Type matches, check variable name against
      pattern and add the variable name to the list
      (if it is not already included).
      */
      %If %Index( &Type, %SysFunc( VarType( &FID, &J ) ) ) %Then
        %Do ;
          %Let VarName = %SysFunc( VarName( &FID, &J ) ) ;
          %If %SysFunc( RXMatch( &RX, &VarName ) ) And
              Not %Index( &VarList, &VarName ) %Then ⑥
                %Let VarList = &VarList &VarName ;
            %End ;
        %End ;
      %End ;
    %End ;
  %End ;
  %SysCall RXFree( RX ) ; ⑦
%End ;
%Else
  %Put WARNING: BuildVarList detected an invalid pattern
("&CurPat")...ignoring. ;

  %Let I = %Eval( &I + 1 ) ;
  %Let CurPat = %Scan( &Patterns, &I, | ) ; ⑧
%End ;

/*
  Did any variable names match any of the pattern(s) ?
*/
%If Not %Length( &VarList ) %Then
  %Put WARNING: BuildVarList found no variable names in &Dset matching
specified pattern(s) . ;

%Exit:
%If &FID %Then
  %Let RX = %SysFunc( Close( &FID ) ) ;

  &VarList ⑨
%Mend BuildVarList ;

```

Source: Michael Bramley, Kindle International, Inc.

- ❶ The user can specify the variable type (character or numeric or both).
- ❷ The user specifies one or more patterns that the variable names must match. In Version 6 and before, the names are always uppercase.
- ❸ There might be more than one pattern separated by a bar ( | ). Loop through each one.
- ❹ The RXPARSE function sets up a memory location for the pattern or patterns for which the comparisons are to be made.
- ❺ For this pattern, loop through all the variables in the data set.
- ❻ The comparison is made by the RXMATCH function using the patterns stored by RXPARSE.
- ❼ After the comparisons have been made, the RXFREE routine is used to clear the memory. Notice that the macro variable &RX is coded without the ampersand when RXFREE is used with %SYSCALL.
- ❽ Select the next pattern to match.
- ❾ Return the list of variables.

The series of Perl Regular Expression pattern matching functions starting with RX that are used in this macro have been updated and are no longer documented. New programs should use the PRX series of Perl Regular Functions, which have expanded capabilities.

The use of Perl Regular Expression functions can be quite sophisticated, and if you do not understand these functions well, they might return results that you do not anticipate. Be especially careful when using the RXMATCH and PRXMATCH functions with wildcards like the \* and pattern separators – both of which are used in this macro.

**MORE INFORMATION:** Usage notes, sample macro calls, and additional documentation for the %BUILDVARLIST macro can be found as a part of the online version of Program 12.4.2e.

**SEE ALSO:** Burroughs (2001) also uses the VNAME routine to capture the variable name from an array. The macro %GETVARS is discussed in detail in Bramley (2001) and similar code can be found in the SAS documentation under the VARNAME function. Lund (2003b) discusses a macro function that returns the list of variables in a data set. Wu and Guan (2016) use PRX functions to read and parse a PDF file as data.

### 12.4.3 Parsing Individual Values from an Existing Horizontal List

In dynamic programs it is not at all unusual to build and parse lists of values. In each of the programs in Section 12.4.2 the macro variable &VARLIST is created to hold a list of variables in a data set. This example assumes that you have already created a macro variable that contains a list of data set variables that are to be used as BY variables. In Program 11.7.5 a series of variables are created in the form of &&KEYS&I where each macro variable contains a list of BY variables. For the sake of discussion the examples in this section simplify this to a single macro variable (&KEYFLD), which contains a single list of BY variables.

A sample definition of the macro variable &KEYFLD might be as follows:

```
%let keyfld = investid subject treatid;
```

In the program that uses this list, we might expect to see a BY statement such as the following:

```
by &keyfld;
```

Very often we need to be able to break down the list of values contained in &KEYFLD into individual values. There are a number of reasons why we might want to do this. Consider the use of FIRST. and LAST. processing in the DATA step. In order to use FIRST. and LAST. processing, you need to know the names of the individual variables in the BY list. This enables you to write statements such as this:

```
by investid subject treatid;
if first.treatid then do;
```

To do this dynamically using &KEYFLD, you need to know such things as its component parts (variable names in the list) and potentially the number of items in the list. In Section 11.4.4 two macro functions (%WORDCOUNT and %COUNTW) are shown that can be used to count the number of words in a list. In the examples that follow in this section, the macro %COUNTCLASS is created to count the number of items within a combination of classification variables passed into the macro as a horizontal list. Programs 12.4.3a through 12.4.3d show various solutions to the problem of using individual words in a horizontal list of words.

### Finding the Last Word in a List of Words

The %SCAN and %QSCAN functions can be used to easily retrieve the last word in the list. You may either use a negative word number or the ‘B’ modifier, both of which count the words from the right instead of the left end of the list. The macro %COUNTCLASS in Program 12.4.3a counts the number of observations within a combination of classification variables. The counting is done in a DATA step and FIRST. and LAST. processing is used on the right-most variable in the list of variables.

#### Program 12.4.3a: Using %SCAN to Retrieve the Last Word in the List

```
%macro countclass(dsn=, keyfld=);

proc sort data=&dsn
           out=bytemp;
   by &keyfld; ①
   run;

data counts(keep=&keyfld classcnt);
   set bytemp;
   by &keyfld; ①
   if first.%scan(&keyfld,-1,%str( )) then classcnt=0; ②
   classcnt+1; ③
   if last.%scan(&keyfld,1,%str( ),b) then output counts; ④
   run;

title1 "Counts within &dsn";
title2 "BYLIST: &keyfld";
proc print data=counts;
   id &keyfld;
   var classcnt;
   run;
%mend countclass;

%countclass(dsn=macro3.clinics, keyfld=region)
%countclass(dsn=macro3.clinics, keyfld=region clinnum)
```

- ① The list of variables is used directly.
- ② FIRST. processing is used to detect the start of each group. The %SCAN function returns the last word in the list &KEYFLD by using the negative word number in the second argument.
- ③ The count is incremented for each observation.
- ④ The last word in the list can also be determined by using the ‘b’ modifier in the fourth argument of the %SCAN function.

**MORE INFORMATION:** The %SCAN function is used to find the last word in a list in Program 7.6.1L.

### Using the Word Count to Retrieve the Last Word

You can also retrieve the last word in the list when you know the number of words that the list contains. In Program 11.4.4b the %COUNTW macro function is used to return the number of words in a list, and in

Program 12.4.3b this macro is in turn used in the %SCAN function to find the last word in the list of BY variables.

#### Program 12.4.3b: Using the Word Count to Retrieve the Last Word in a List

```
%macro countw(list,dlm); ⑤
  /* Count the number of words in &LIST;
  %if %length(&dlm)=0 %then %sysfunc(countw(&list));
  %else %sysfunc(countw(&list,&dlm));
%mend countw;

%macro countclass(dsn=, keyfld=);
proc sort data=&dsn
           out=bytemp;
by &keyfld;
run;

data counts(keep=&keyfld classcnt);
  set bytemp;
  by &keyfld;
  if first.%scan(&keyfld,%countw(&keyfld)⑥,%str( )) then classcnt=0;
  . . . . portions of the macro are not shown . . . .
```

- ⑤ The %COUNTW macro function returns the number of words in a list of words and is described in Program 11.4.4b.
- ⑥ By using the number of words in the &KEYFLD list, as returned by %COUNTW, as the second argument in the %SCAN function, the last word in &KEYFLD is returned.

#### Creating and Using a List of Individual Words

Rather than use the horizontal list directly you might need to break the list up into individual words so that each of the words can be used individually (as in a vertical list). When doing this from within a utility macro, there are a couple approaches. In Program 12.4.3c the horizontal list is parsed by the macro routine %MAKEVARS, which passes the new vertical variables out to the calling macro where they can be used.

#### Program 12.4.3c: Parsing and Using Individual Words from the Horizontal List

```
%* COUNTW is Program 11.4.4b.sas;
%macro countw(list,dlm);
  /* Count the number of words in &LIST;
  %if %length(&dlm)=0 %then %sysfunc(countw(&list));
  %else %sysfunc(countw(&list,&dlm));
%mend countw;

%macro makevars(list=);
%local wcnt;
%do wcnt = 1 %to %countw(&list); ⑦
  %let keyvar&wcnt = %qscan(&list,&wcnt,%str( ));
%end;
%mend makevars;

%macro countclass(dsn=, keyfld=);
%local i keycnt;
/* Build a local list of individual key variables;
%do i = 1 %to %countw(&keyfld); ⑧
  %local keyvar&i;
%end;
%let keycnt=%eval(&i-1); ⑨

/* The macro variables created by %makevars will be written
/* to the local table for %COUNTCLASS;
%makevars(list=&keyfld) ⑩
```

```

proc sort data=&dsn
    out=bytemp;
by &keyfld;
run;

data counts (keep=&keyfld classcnt);
    set bytemp;
    by &keyfld;
    if first.&&keyvar&keycnt ⑪ then classcnt=0;
    classcnt+1;
    if last.&&keyvar&keycnt then output counts;
run;
. . . . portions of the macro are not shown . . .

```

- ⑦ The %MAKEVARS macro creates a macro variable (&KEYVAR1, &KEYVAR2, and so on) for each word in the incoming list of words. Only the word count (&WCNT) is forced into the local symbol table.
- ⑧ A list of macro variables of the form &KEYVAR1, &KEYVAR2, and so on, are forced into the local symbol table for %COUNTCLASS through the use of a list of generated %LOCAL statements. The values for these macro variables will be supplied by %MAKEVARS. Because these macro variables will already exist in the local table for %COUNTCLASS, when they are created in %MAKEVARS, they will be written here instead of to the local table for %MAKEVARS.
- ⑨ The number of variables in the list is stored in &KEYCNT. This is the same value as is returned by %COUNTW.
- ⑩ Because the macro variables created by %MAKEVARS ⑦ are already on the local table for %COUNTCLASS, they will not be written to the local table for %MAKEVARS. This effectively passes the values out of %MAKEVARS without writing them to the global symbol table.
- ⑪ The number of the last word in the list of words will be &KEYCNT, and &&KEYVAR&KEYCNT will resolve to the last word. In this case we want to use the last word for our FIRST. and LAST. processing.

In Program 12.4.3c the macro variable &KEYCNT is established to hold the number of words in the list. When the last item in the list of words is desired, the macro variable reference is

&&KEYVAR&KEYCNT ⑪. Since this value is derived using the %COUNTW macro, we could avoid saving it and use the %COUNTW function directly. In Program 12.4.3d &KEYCNT is not created. Instead it is replaced with a call to %COUNTW. This requires us to use a macro call as part of the macro variable name. Intuitively we might try specifying the macro variable as &&KEYVAR%COUNTW(&KEYFLD); however, this specification will not resolve correctly. Because the macro %COUNTW will not be called until after the &&KEYVAR has been resolved, the timing is wrong to use the result of the %COUNTW macro as a part of the macro variable name. Program 12.4.3d delays the resolution of &KEYVAR until after the value generated by %COUNTW is returned.

#### Program 12.4.3d: Using a Macro Call as Part of a Macro Variable Name

```

. . . . portions of this program are not shown . . .

if first.%unquote(%nrstr(&keyvar)%countw(&keyfld))⑫ then classcnt=0;
classcnt+1;
if last.%unquote(%nrstr(&keyvar)%countw(&keyfld)) then output counts;
. . . . portions of the macro are not shown . . .

```

- ⑫ &KEYVAR&KEYCNT is replaced with %unquote(%nrstr(&keyvar)%countw(&keyfld)). The %COUNTW macro will return the same value as &KEYCNT did in Program 12.4.3c. The %NRSTR quoting function is used to delay the resolution of &KEYVAR. Notice that there is now only one ampersand. After the %COUNTW macro has executed and returned a number, the %UNQUOTE is used to allow the resolution of &KEYVAR, which will now have a number appended

to it. If there are two variables in the list (%COUNTW returns a 2), the unresolved macro variable will be &KEYVAR2.

#### 12.4.4 Placing Commas between Words

Sometimes when you have a horizontal list of space-separated words, you need to convert it to be a comma (or other separator) separated list. For example, you might need to convert a list for use with an IN operator. There are a couple of basic approaches that you can take to make the insertions.

##### Building the List a Word at a Time

You can place a comma between words, one word at a time using a %DO %WHILE loop. We step through the list of words one word at a time while inserting a comma between words. Because the %QSCAN function is used to separate the words, a variety of possible initial word separators can be replaced. In Program 12.4.4a successive words are separated and appended back onto a growing comma-separated list.

##### Program 12.4.4a: Using %DO %WHILE to Separate and Rebuild a Word List

```
%macro cstr(list);
  /* Return a list of words as a comma-separated list;
   *local clist count; ①
   %let count=0; ②
   %do %while(%qscan(&list,&count+1) ③ ne %str() ) ④;
     %let count = %eval(&count+1); ⑤
     %if &count=1 %then %let clist = %qscan(&list,&count); ⑥
     %else %let clist = &clist,%qscan(&list,&count); ⑦
   %end;
   &clist
%mend cstr;
```

A portion of the SAS Log shows how the list has been broken apart and reconstructed as a comma-separated list.

##### Program 12.4.4a (Portion of SAS Log): Shows the List Reconstructed with Commas

```
972 %put |%cstr(region)|;
|region|
973 %put |%cstr(region clinnum)|;
|region,clinnum|
974 %put |%cstr(region.clinnum/sex    edu)|;
|region,clinnum,sex,edu| ⑧
```

- ① %CSTR is a macro function and as such, all the macro variables that it generates must be local.
- ② Initialize the word count macro variable (&COUNT) to 0.
- ③ The %QSCAN function will check to see if the &COUNT+1 word exists. Notice that the third argument, which specifies word delimiters, is not used; consequently all the default word separators will be recognized.
- ④ If the requested word does not exist, %QSCAN returns a null value, which is tested for by specifying a %STR function with no space between the parentheses.
- ⑤ The &COUNT+1 word was found, so increment &COUNT.
- ⑥ If this is the first word, retrieve it and place it in &CLIST.
- ⑦ If it is not the first word, append it to &CLIST with a comma between the current value of &CLIST and the new word.
- ⑧ The SAS Log shows that word delimiters other than spaces have also been replaced with commas. We could have selected a specific word delimiter by adding a third argument to the %QSCAN function.

## Using the TRANWRD Function

If you can assume that the word separators are always a single known character (a blank space in this example), the known character could be replaced with a comma using the TRANWRD function. In Program 12.4.4b the TRANWRD function is called using %SYSFUNC.

### Program 12.4.4b: Changing Spaces to Commas with the TRANWRD Function

```
%macro cstr(list);
  /* Return a list of words as a comma-separated list;
   %sysfunc(tranwrd(%cmpres(&list),%str( ),%str(,)))
  %mend cstr;
```

In order to prevent multiple blanks from becoming multiple commas, the %CMPRES autocall macro function is first applied to remove multiple blanks.

## 12.4.5 Quoting Words in a List

In my opinion it is generally not optimal to store special characters such as quotes in a macro variable. I would much rather add them at the time the macro variable is used. That said, there are times when you do not have an option, and you might need to have a macro variable containing a list of quoted words. Fortunately, like in the examples adding commas between words, there is more than one way to quote the words as well.

### Building the List a Word at a Time

The %QSTR macro function in Program 12.4.5a is very similar to the %CSTR macro shown in Program 12.4.4a. Both macros process a list of words one word at a time. %QSTR is used to separate words in an incoming string with commas, as well as to quote the individual words.

### Program 12.4.5a: Using %DO%WHILE to Separate and Quote Words

```
%macro qstr(list);
  /* Return a list of quoted and comma separated words;
  %local qlist count;
  %let count=0;
  %do %while(%qscan(&list,&count+1) ne %str() );
    %let count = %eval(&count+1);
    %if &count=1 %then %let qlist = %str(%')①%qscan(&list,&count);
    %else %let qlist = &qlist%str(%',%)②%qscan(&list,&count);
  %end;
  &qlist%str(%') ③
  %mend qstr;
```

- ① A single quote is pre-appended to the first word using the %STR quoting function. Table 7.1.9 discusses the use of the % sign when masking single quotes with the %STR function.
- ② A ', ' is placed between the existing list and the latest word.
- ③ The final trailing quote is added when the list is passed out of the %QSTR macro.

## Using the TRANWRD Function

Like in Program 12.4.4b, when the word delimiters are known (spaces in this example), the TRANWRD function can be used to simplify the process of adding the quotes.

### Program 12.4.5b: Using TRANWRD to Add Quotes

```
%macro qstr(list);
  /* Return a list of words as a quoted and comma-separated list;
  ④ %str(%')%qsysfunc(tranwrd(%cmpres(&list),%str( ),%str(%' ,%'⑤
  )))%str(%')⑥
  %mend qstr;
```

- ❸ The leading quote is added using the %STR function.
- ❹ The ' , ' is added between words. A space has been added between the first quote and the comma to avoid a NOTE about characters following quotes in the SAS Log.
- ❺ The trailing quote is added using %STR.

The SAS Log shows the quoting for some sample macro calls:

```
207 %put |%qstr(region)|;
|'region'|_
208 %put |%qstr(region clinnum)|;
|'region','clinnum'|_
209 %put |%qstr(region.clinnum/sex    edu)|;
|'region.clinnum/sex','edu'|
```

Notice that the non-blank-separated words remain as a single quoted string and are not individually quoted and comma separated.

### Quoting Words in a SQL Step

When the list of words is to be created in a PROC SQL step, the quoting can be done as the list is generated. In macro %QCHARVAR shown in Program 12.4.5c the list of character variable names in a data set is returned quoted. The quoting is accomplished using the QUOTE function, which returns double quotes.

#### Program 12.4.5c: Returning a Quoted List Using SQL

```
%macro qCharVar(lib=,dsn=);
  %local qlist;
  proc sql noprint;
    select quote(trim(name)) ❷
      into :qlist separated by ","
      from dictionary.columns
      where libname="%upcase(&lib)"
        & memname="%upcase(&dsn)"
        & type='char'; ❹
    quit;
    /* Usage of the quoted list would go here. ;
    %put |&qlist|; ❽
  %mend qcharvar;
```

- ❷ The QUOTE function is used to place double quotes around the trimmed variable name.
- ❸ A comma is used to separate the list of quoted words.
- ❹ In this macro only the names of character variables are desired.
- ❽ The code that would actually take advantage of this quoted list would go here, perhaps as a macro call instead of a %PUT. Like the other macros in this section, this macro could easily be converted to a macro function by replacing the %PUT with just &QLIST. This would increase the usability and flexibility of this macro.

---

### 12.4.6 Checking for Existence of Variables

A process might generate a list of variable names that are to be used in a VAR statement or in a KEEP= option. If the list of variables is supplied by the user and one or more of the variables are not on the data set of interest, errors will be generated. The %VAREXIST macro in Program 12.4.6 can be used to verify that each of the variables in the list is on the data set of interest.

In this macro the VARNUM DATA step function is used with %SYSFUNC to verify each variable's existence, and the result is passed back to the user as a true/false (1/0) value.

**Program 12.4.6: Checking the Validity of a List of Variable Names**

```
%macro varexist(dsn=,varlist=);
  %local dsid i ok;
  %let dsid=%sysfunc(open(&dsn)); ①
  %let i=0;
  %let ok=1; ②
  %do %while(%scan(&varlist,&i+1) ③ ne %str());
    %let i = %eval(&i+1);
    %if %sysfunc(④varnum(&dsid,%scan(&varlist,&i)))=0
      %then ⑤%let ok = 0;
  %end;
  %let dsid = %sysfunc(close(&dsid)); ⑥
  &ok ⑦
%mend varexist;
```

- ① The data set of interest is opened for access.
- ② &OK is initialized to 1 (all variables exist on the data set).
- ③ %SCAN is used to check to see whether there is another variable name in the list (&VARLIST).
- ④ The VARNUM function is used to retrieve the number of the variable, given its name which is retrieved from the variable list. If the variable name is not found, a number 0 is returned.
- ⑤ If the variable is not found, then change &OK to 0. Once any variable on the variable list is not found, the value of &OK can only be 0.
- ⑥ Close the data set.
- ⑦ The value of &OK is returned. 0 indicates that one or more of the variable names on the variable list are not present.

This macro works as a macro function, so it can be used in a %PUT or %IF statement. The call %VAREXIST is used in a macro %IF below.

```
%macro doit;
  %let list= name age sex gg;
  %if %varexist(dsn=sashelp.class,varlist=&list) %then
    %put all are on data set;
  %else %put One or more variables are not on the data set;
%mend doit;
%doit
```

The variable GG does not exist on the data set SASHELP.CLASS, so %DOIT returns:

One or more variables are not on the data set
---

**12.4.7 Removing Repeated Words from a List**

When a list of variable names has been created from a source such as PROC CONTENTS or SASHELP.VCOLUMNS, you can be confident that each variable will occur in the list exactly once. Sometimes, however, the list will be constructed in such a way that a given word or variable name might occur more than once, and these repeated names might cause errors.

The following DATA step shows a simple example where this type of problem can show up. It is likely that the lists of incoming variables from each of the three incoming data sets are similar, and it is possible that they are not exactly the same. The KEEP= option will have some (if not most) variables listed more than once. The %DISTINCTLIST macro in Program 12.4.7 removes duplicate values from a list of words, which in this case would be duplicate variable names.

```
data allyears(keep=&list00 &list01 &list02);
  set year2000(keep=&list00)
      year2001(keep=&list01)
      year2002(keep=&list02);
  ...code not shown...
run;
```

While the overlapping variables will not cause an ERROR in this usage, it would be more convenient to have a single unified list of variables. A consolidated list can be built using the %DISTINCTLIST macro.

```
%let alllist = %distinctlist(&list00 &list01 &list02);
data allyears(keep=&alllist);
  set year2000(keep=&list00)
      year2001(keep=&list01)
      year2002(keep=&list02);
  ...code not shown...
run;
```

The macro %DISTINCTLIST will remove any repeated word within a list. Each word is selected from the incoming list and added to a new list only if it has not already been found. The INDEXW function is used to see whether any given word has already been added to the new list.

#### Program 12.4.7: Removing Duplicated Words with the INDEXW Function

```
%macro DistinctList(list);
  %local dlist i;
  /* Select the first word;
  %let dlist = %scan(&list,1,%str( )); ①
  %let i=2; ②
  %do %while(%scan(&list,&i,%str( )) ne %str()); ③
    /* There is another word.
    /* Has it already been selected?;
    %if %sysfunc(indexw(&dlist,%scan(&list,&i,%str( ))))=0 %then %do; ④
      /* First occurrence of this word add it to the list;
      %let dlist = &dlist %scan(&list,&i,%str( )); ⑤
    %end;
    /* Increment counter to get the next word;
    %let i = %eval(&i+1); ⑥
  %end;
  &dlist ⑦
%mend distinctlist;
```

- ① The first variable name is automatically added to the new list.
- ② Set the word counter for the next possible word.
- ③ Determine whether there is another word to check, and if so, enter the %DO %WHILE loop.
- ④ The INDEXW function is used to see whether the current word has already been added to the new list (&DLIST).
- ⑤ The word was not found; therefore, this is the first occurrence of this word, so add it to the new list.
- ⑥ Increment the word counter.
- ⑦ Return the new list.

**MORE INFORMATION:** %INDEXW is presented as a macro function in Program 7.6.1m.

### 12.4.8 Controlled Data Corrections and Manipulations

When processing large volumes of data, the correction of individual data values can become time-consuming, and if done manually, error-prone. We sometimes need a way to automatically correct as many data errors as possible and to do it in such a way that it is repeatable and consistent. In Program 11.7.5 control files are used to dynamically construct field checks that will alert the data manager to data inconsistencies. In that program, DATA step IF-THEN/ELSE statements were dynamically written to perform the checks. The programs in this section highlight a different approach to solve a similar problem.

These programs were applied to a series of data sets, each of which had a large number of variables. Data inconsistencies could not be corrected at the source (data updates would not consistently include data error corrections). We needed a dynamic solution that would make data corrections as the data was imported for our use.

A control file listing the data set, field of interest, and the type of correction was created. The control file was then used to execute a series of field-correction macros. Because the correction macro was named in the control file as new corrections were uncovered, a new macro could be written without changing any of the underlying programs.

The real beauty of this approach is that new checks and fixes can be implemented by merely adding a row in the control file that points to a new macro. None of the existing programs or macros need to be modified. The program will automatically execute the macros specified in the control file. Let's take a look at how this is accomplished:

1. The control file contains information on what will be changed and the name of the macro that will perform the change.
2. The information in the control file is read into macro variable lists. These lists contain the names of the macros that will be needed to implement the change.
3. The lists are processed within a %DO loop and the macros that implement the changes are called.

#### CSV Control File

Because of its convenience to the data manager, the control information was established as a CSV file. This file contains the name of the data table (DATATABLE), the name of the variable requiring the fix (VARNAME), and the type of action that is to be applied to that field (ACTION). Two supporting columns (FORMAT and CHK\_TYPE) have values when appropriate.

**Figure 12.4.8: CSV Control File Used by the %FIXRAW Macro**

A	B	C	D	E
1 datatable	varname	action	format	chk_type
2 exp_billing	bill_chg_fac	make_n		
3 exp_billing	payor01_as_text	applyfmt	\$payor_fmt.	
4 exp_billing	payor02_as_text	applyfmt	\$payor_fmt.	
5 exp_billing	payor03_as_text	applyfmt	\$payor_fmt.	
6 exp_billing	payor04_as_text	applyfmt	\$payor_fmt.	
7 exp_demopat	age_value	make_n		
8 exp_demopat	dob_text	make_n	mmddyy10.	
9 exp_demopat	pat_adr_fco_as_text	applyfmt	\$AKFips.	
10 exp_demopat	pat_adr_reg_as_text	applyfmt	\$pat_adr_reg_fmt.	
11 exp_demopat	pat_adr_zip	make_n		zip
12 exp_demopat	pat_ethnic_as_text	applyfmt	\$pat_ethnic_FMT.	

The types of potential error corrections such as those shown in Figure 12.4.8 include the following:

#### MAKE\_N

Convert the variable from character to numeric

When CHK\_TYPE=ZIP treat as a ZIP code value

**APPLYFMT**

Update the variable's value with its formatted value

**MAKE\_C**

Convert from numeric to character

**MAKE\_CASE**

Convert to upper, lower, or mixed case

These values are stored in the ACTION column. The value of ACTION not only determines what will be done to the data, but it also names the macro that will perform the operation. The MAKE\_N action will be accomplished using the %MAKE\_N macro.

### Reading and Using the Control Information

The information contained in the control file is read and transferred to a series of macro variable lists. Because each row in the control file contains all the information needed to make any given change, we will be able to step through the lists one change at a time.

#### Program 12.4.8a: Reading and Using the Control Information

```
%macro fixraw;
%local fixcnt i ;

/* Convert the data fixraw file from csv to a SAS data set;
PROC IMPORT OUT= FixRaw ❶
    DATAFILE= "&path\data\FixRaw.csv"
    DBMS=csv REPLACE;
    guessingrows=80;
    GETNAMES=YES;
RUN;

/* Build macro variable lists from the controldata;
proc sql noprint;
select datatable, varname, action, format, chk_type
    into :dsn1 - :dsn999, ❷
        :varname1 - :varname999,
        :action1 - :action999,
        :fmt1 - :fmt999,
        :chktyp1 - :chktyp999
    from fixraw;
%let fixcnt=&sqllobs;
quit;

/* Call the macro specific to this action;
%do i = 1 %to &fixcnt; ❸
    %&&action&i(dsn=&&dsn&i, var=&&varname&i,
                fmt=&&fmt&i, chktype=&&chktyp&i) ❹
%end;
%mend fixraw;
options nomprint;
%fixraw
```

- ❶ The control file is read into a SAS data set (WORK.FIXRAW) using PROC IMPORT.
- ❷ A SQL step is used to read the control information into a series of macro variable lists. The number of items in each list, which is the same as the number of rows in the control file, is saved in the macro variable &FIXCNT. Because the older style of naming the upper bound of the list is used (this client does not yet have SAS 9.4 available), the control file is effectively limited to 999 observations.
- ❸ A %DO loop is used to cycle through each row in the control file.
- ❹ &&ACTION&I contains the name of the macro that is to be executed. When executing macro calls of the form %&VAR, the macro variable reference (&VAR, which holds the name of the macro) will

resolve before the macro can actually be called. For the first row in the control file, the macro call becomes:

```
%make_n(dsn=exp_billing,var=bill_chg_fac, fmt=, chktype=)
```

**SEE ALSO:** Hughes (2016a) calls a macro where the name of the macro is stored in a macro variable.

### Examining the Macro That Performs the Change

The macro specified by the ACTION variable performs a very specific operation. The %APPLYFMT macro shown in Program 12.4.8 applies a format to a variable's value and writes the formatted value back into the original variable. Typically this is used when a variable is supposed to have coded values or values with specific spellings. The format contains a mapping of all known incorrect values into correct values.

#### Program 12.4.8b: Converting a Variable Based on a Format

```
%macro applyfmt(dsn=,var=,fmt=,chktype=);
  /* Make sure the format is present;
  %if &fmt = %then %do; ⑤
    %put ERROR: Format is missing for a FIXRAW request;
    %put ERROR- &=DSN &=var;
    %return; ⑥
  %end;

  /* Make sure that the format has a period;
  %if %index(&fmt,.)=0 %then %let fmt=&fmt..; ⑦

  data &dsn; ⑧
    set &dsn;
    &var = put(&var,&fmt); ⑨
  run;
%mend applyfmt;
```

- ⑤ Verify that a format name has been specified. Write customized ERRORS to the SAS Log if a format has not been supplied.
- ⑥ Terminate the macro if a format is not present.
- ⑦ If the format's period was not supplied by the user, add it.
- ⑧ A DATA step is created that will read and rewrite the named data set.
- ⑨ The variable of interest (&VAR) is rewritten using the PUT function and the specified format.

**MORE INFORMATION:** Macro calls of the form %&VAR were first introduced in Section 9.1.1.

## **Part 4: Miscellaneous Topics and Examples**

<b>Chapter 13 Examples and Utilities to Perform Various Tasks.....</b>	<b>375</b>
<b>Chapter 14 Miscellaneous Topics .....</b>	<b>397</b>

A number of the macros throughout this book were written by SAS programmers other than the author of this book. These macros are noted with the name and occasionally (with their approval) additional contact information for the macro's author. Sometimes macros such as these are passed from programmer to programmer and it is hard to identify the original author. In these cases, I have included the name of the most recent contributor. In order to control content and to stress certain points, some of the macros have been slightly altered from their original form.

As you look over these macros please remember that programming, as in many creative endeavors, is very individualistic. Many of the programs included here might not reflect your style, and some are more efficient than others. They all have been included to demonstrate both techniques and style. In each case, the authors should be complimented on their contributions to the SAS community.

Examples of other macros and utilities are very common in the SAS literature. Proceedings from SAS Global Forum and other SAS user conferences are especially rich in these kinds of programs. Most of these papers are available in electronic form, and they provide an excellent source of examples and explanations.

One of the very best sources of examples of macros can be found on sasCommunity.org in the Macro Language category: [http://www.sascommunity.org/wiki/Category:Macro\\_Language](http://www.sascommunity.org/wiki/Category:Macro_Language). Here you will find examples of macros that perform a very wide variety of tasks. As you write your own macro utilities, hopefully you will take the time to add them to sasCommunity. A new chapter in the macro language documentation is being written that highlights macro examples that focus on commonly encountered problems.

The book *SAS System for Statistical Graphics, First Edition* by Michael Friendly (1991) has a very nice collection of macros that are useful in both statistical and graphical applications.

*Multiple-Plot Displays: Simplified with Macros* by Perry Watts (2002) is a collection of macros that are designed to work in the SAS/GPGRAPH environment. She presents additional SAS/GPGRAPH macros in Watts (2003a and 2003b).

A Macro Language Focus Area has been developed where users can search for macro language related topics. This includes a section entitled "Samples and Tips" that includes information on FAQ:  
<http://support.sas.com/rnd/base/macro/index.html#s1=1>.

# **Chapter 13: Examples and Utilities to Perform Various Tasks**

<b>13.1 Working with Operating System Operations .....</b>	<b>375</b>
13.1.1 Write the First N Lines of a Series of Flat Files .....	375
13.1.2 Storing System Clock Values in Macro Variables .....	378
13.1.3 Executing a Series of SAS Programs .....	379
<b>13.2 Working with the Output Delivery System .....</b>	<b>381</b>
13.2.1 Why You Might Need to Automate with Macros .....	382
13.2.2 Controlling Directories .....	382
13.2.3 Controlling Hyperlinks .....	384
<b>13.3 Working with Data .....</b>	<b>389</b>
13.3.1 Selection of a Top Percentage of Observations .....	389
13.3.2 Selection of Top Percentage Using the POINT Option .....	390
13.3.3 Random Selection of Observations .....	391
13.3.4 Building a WHERE Clause Dynamically .....	394

Through the use of macro examples, this chapter highlights a number of concepts and methodologies that you can use to control how SAS interfaces with the operating environment as well as the data it works with.

For the examples in this chapter and indeed for all of the code examples throughout the book, if you want to execute these sample programs, then be sure to follow the setup instructions. Remember that all of the data sets and programs are available for download, so you do not need to retype either the code or the data. For instructions on accessing and setting up the programs and data, see the "Example Code and Data" section within this edition's "About This Book" front matter. You will find that not every macro language element is explained in every example. You can, however, find an explanation for macro language elements elsewhere in the book (see Appendix 4 to locate examples and explanations of various macro language elements).

---

## **13.1 Working with Operating System Operations**

As you develop SAS programs you will often need to interface with the operating system. This includes accessing OS files, submitting programs, and examining file and directory structures. There have been a number of examples throughout this book that touch on this interface. A few more examples are included here.

**MORE INFORMATION:** The %SYSGET function (see Section 8.1.1) is used to retrieve OS environmental variables. In Section 8.1.4 the %SYSPROD function is used to return SAS product availability information. The %SYSEXEC statement (see Section 8.2.4) is used to execute OS scripts and commands. In Section 9.7 the use of remote servers is discussed.

---

### **13.1.1 Write the First N Lines of a Series of Flat Files**

When you need to view the first few lines of a series of flat files, it is inconvenient to use an interactive tool such as Notepad.

The %DUMPIT macro is a handy utility to perform this operation. It is used to list the first few lines of each of a series of flat files. The example code in Program 13.1.1a dumps the first few lines of some QSAM files, PDSs, and so on, using a list. Notice that this was used on the mainframe, so the FILENAME statement ❸ has a DISP=SHR.

### Program 13.1.1a: Write the First N Lines of a Flat File

```
%macro dumpit (cntout);
  /* create a local counter;
  %local cwj;

  %do cwj=1 %to &numobs;

    /* fileref to identify the file to list;
    filename dump&cjw "&&&invar&cjw" ❸ disp=shr; ❹

    * read and write the first &cntout records;
    data _null_; ❺
      infile dump&cjw end=done; ❻
      * read the next record;
      input;
      incnt+1;
      if incnt le &cntout then list; ❻
      if done then do;
        file print; ❾
        put //@10 "total records for &&invar&cjw is "
          +2 incnt comma9. ;
      end;
    run;

    filename dump&cjw clear;

  %end;
%mend dumpit; * the macro definition ends;

* read the control file and establish macro variables;
data dumpit; ❶
  infile cards;
  input filenam $25.;
  cnt+1;
  newname=trim(filenam);
  * the macro variable INVARi contains the ith file name;
  call symput ('invar'!!trim(left(put(cnt,3.))),newname); ❷
  * store the number of files to read;
cards; ❸
PNB7.QSAM.BANK.RECON
PNB7.QSAM.CHECKS
PNB7.QSAM.CHKNMBR
PNB7.QSAM.CKTOHIST
PNB7.QSAM.DRAIN
PNB7.QSAM.RECON
PNB7.BDAM.BDAMCKNO
PNB7.BDAM.VCHRCKNO
PNB7.QSAM.CS2V3120.CARDIN
PNB7.QSAM.CASVCHCK
PNB7.QSAM.CASVOUCH
PNB7.QSAM.VCHR3120.CARDIN
PNB7.QSAM.VOUCHERS
TAX7.JACKSON
;;
title "City of Dallas - ECI (FINSYS), jobname is &sysjobid";
title2 "List of files to dump";
```

```

proc print data=dumpit;
run;

* Pass the number of records to dump from each file;
%dumpit (25);

```

Source: Clarence William Jackson, CJAC

- ➊ The DATA step is used to read the names of the files ➋ that are to be dumped. These names are stored in the variable NEWNAME in the data set DUMPIT.
- ➋ The list of files in this example is read into the DATA step using the CARDS statement. The list could just as easily have been built using CLIST or using other methods.
- ➌ The SYMPUT routine is used to load the name of the i<sup>th</sup> dump file into the macro variable &INVARi.
- ➍ The *fileref* created in this FILENAME statement is used to identify the name of the file that is to be read and listed.
- ➎ The macro variable &&INVAR&CWJ in the FILENAME statement ➏ was created in the DATA step ➊ using the CALL SYMPUT routine ➌ before the macro %DUMPIT is called.
- ➏ A DATA \_NULL\_ step is used to read and list the lines in the flat file.
- ➐ The INFILE points to the *fileref* that was established in the preceding FILENAME statement ➏.
- ➑ The LIST statement writes to the SAS Log.
- ➒ Note that although this macro lists only the first &CNTOUT lines of each flat file, the entire file is read in order to establish a line count.

Unlike the %DUMPIT macro in Program 13.1.1a, the %LISTLINES macro in Program 13.1.1b dynamically gathers the names of selected files (SAS programs) within a stated directory and then writes the first N lines to a SAS data set.

#### Program 13.1.1b: Listing the First N Lines of a Program

```

%macro ListLines(loc=, linecnt=10);

  * Build a list of files at this location;
  filename flist pipe "dir ""&loc\*.*" /o:n /b "; ➊

  data listlines(keep=filename sasline);
    length filename $85 rawloc $585 sasline $100;
    infile flist truncover;
    input filename $85.; ➋
    rawloc = catt("&loc",'\',filename); ➌

    done=0;
    cnt=0;
    infile dummy filevar=rawloc end=done truncover; ➍
    do until(done or cnt ge &linecnt);
      input sasline $char100.; ➎
      cnt+1;
      output listlines;
    end;
    run;
    filename flist clear;
    title"First &linecnt lines of each program";
    proc print data=listlines; ➏
      run;
  %mend listlines; * the macro definition ends;

  * Pass the number of records to list from each file;
  %listlines (loc=&path\Chapter 12\SAS Programs, linecnt=1) ➋

```

- ❶ A *fileref* is established to point to the SAS programs in the specified location. The DIR command builds a virtual list of filenames matching the criteria. The DIR command switches (/o:n /b) are also used in Program 12.1.2d and are discussed there.
- ❷ Read the name of the SAS file into the variable FILENAME. Each observation read from FLIST will have a distinct filename.
- ❸ The variable RAWLOC will contain the full path information for the file of interest.
- ❹ The RAWLOC variable is used with the FILEVAR= option on the INFILE statement to name the next file to read from.
- ❺ The INPUT statement inside the DO loop will read one or more lines from the file specified in RAWLOC. The values will be stored in the data set LISTLINES.
- ❻ The PROC PRINT step will list the name of the file and the first few lines of each file.
- ❼ The call to %LISTLINES shows the first line for each Chapter 12 program in this book. The first line is supposed to have the name of the program.

**MORE INFORMATION:** Program 10.6.9 uses a related technique to search for macros in the SASAUTOS concatenated *fileref*.

**SEE ALSO:** Widawski (1997b) collects a list of dBase files that are to be converted to SAS files. Several examples that are discussed by Yu (1998) use %SYSFUNC to read and write external files, and Chen (2003) discusses a number of methods that can be used to read external files.

### 13.1.2 Storing System Clock Values in Macro Variables

The automatic macro variables &SYSDATE and &SYSTIME reflect the time that the SAS session was invoked, and during long sessions they might not accurately reflect the actual time that a particular step executed. At a particular point in your program's execution, the current date and time can be captured and stored using the macro %UPDATE.

This macro can be used to grab either the current date value, the current time value, or both, and it refreshes or updates the macro variables &TIMESTR, &TODAYSTR, and &NOWSTR. You can specify any one of the three macro variables, or you can specify ALL to update each of the macro variables. Notice that these three macro variables reside in the global symbol table, so you might need to be selective about the names before using the macro in your system.

#### Program 13.1.2: Storing the Current Date and Time Values

```
%macro update(type=date);
  %global timestr datestr bothstr; ❶
  %let timestr=;
  %let datestr=;
  %let bothstr=;
  %let type = %lowcase(&type);
  %if &type=all %then %let type=both;

  %if &type=date or &type=both %then
    %let datestr = %left(%qsysfunc(today(),worddate.)); ❷
  %if &type=time or &type=both %then
    %let timestr = %left(%sysfunc(time(),HHMM.)); ❸

  %if &type=both %then %let bothstr=&timestr._&datestr;
%mend update;
%update(type=date)
```

- ❶ The macro variables that will hold the current date time values are loaded into the global symbol table.
- ❷ The %QSYSFUNC macro function is used to invoke the TODAY() function, which retrieves the current date, which is then formatted using the WORDDATE. format.
- ❸ The TIME() function is called using the %SYSFUNC macro function.

**MORE INFORMATION:** The %CURRDATE macro in Program 7.6.1d avoids the use of the global symbol table by building a macro function.

### 13.1.3 Executing a Series of SAS Programs

Execution of SAS programs from batch mode can sometimes have advantages even for users that usually execute interactively. Depending on how you would like to approach the problem, you might want to either execute the programs from within a single SAS job or by using a series of SAS executions—one per SAS program.

In Program 13.1.3a the macro %MAKERUNBAT, creates a batch file with a separate execution of SAS for each SAS program in the specified directory. This batch file can then be scheduled for execution at some later time.

#### Program 13.1.3a: Creating a Batch Program

```
%macro makerunbat(saspgmloc=c:\temp,batchloc=c:\);
  filename c1sas pipe "dir ""&saspgmloc\*.sas"" /o:n /b"; ①

  data _null_;
    length saspgm $40  excmd $350;

    * read the names of the SAS programs;
    infile c1sas truncover;
    input saspgm $40.; ②

    * Write out the batchfile into &batchloc;
    file "&batchloc\Program13.1.3a_runbat.bat"; ③

    * Build the executable command;
    excmd = catt('!', "%sysget(sasroot)\sas.exe", ④
                  ' -sysin ', "&saspgmloc", ⑤
                  '\', saspgm, '''); ⑥

    put excmd; ⑦
  run;

%mend makerunbat;
%makerunbat(saspgmloc=&path\chapter 1\SAS Programs,batchloc=c:\temp) ⑧
```

- ① A virtual list of the SAS programs at the &SASPGMLOC location is created and associated with the fileref C1SAS. Each of these programs is to be executed by a batch job. The DIR command switches (/o:n /b) are also used in Program 12.1.2d and are discussed there.
- ② The name of the SAS program is read from the input file and stored in the variable SASPGM.
- ③ A batch file is to be created and will be stored here.
- ④ The location of the SAS executable file is retrieved using the %SYSGET function.
- ⑤ The –SYSIN initialization option is used to identify the location and the name ⑥ of the SAS program to be executed.
- ⑥ The name of the program is appended onto the location.
- ⑦ The SAS executable command and file to be executed are written to the batch file ③.
- ⑧ The %MAKERUNBAT macro is called.

The resulting batch file "&batchloc\Program13.1.3a\_runbat.bat" can be immediately executed or even scheduled for execution at a later time. The file itself will contain one set of instructions for each program to be executed.

**Program 13.1.3a (Generated Batch Program): Batch Program Created by SAS**

```
"C:\Program Files\SASHome2\SASFoundation\9.4\sas.exe"
-sysin "C:\chapter 1\SAS Programs\Carpenter_17835TW_Program1.2.1.sas"

"C:\Program Files\SASHome2\SASFoundation\9.4\sas.exe"
-sysin "C:\chapter 1\SAS Programs\Carpenter_17835TW_Program1.4.sas"
```

Since there is a separate invocation of SAS for each program, there will also be a separate SAS Log and output for each program. While this might be an advantage, overall this approach might also have a disadvantage. If there are eight programs to be executed, SAS will be opened and closed eight times. It might be better to instead open SAS once and execute those same eight programs from within a single SAS session. The variation of %MAKERUNBAT in Program 13.1.3b will create a batch file that executes SAS only once while still executing all of the requested programs.

In this approach, rather than building a single batch file with a series of calls to SAS, two files are created. A temporary SAS program is written that uses an %INCLUDE statement to execute each program of interest. It is this program that is referenced in the batch program.

**Program 13.1.3b: Writing a Batch Program That Calls a Master Program**

```
%macro makerunbat2(saspgmloc=c:\temp,batchloc=c:\);
  filename c1sas pipe "dir ""&saspgmloc\*.sas"" /o:n /b";

  data _null_;
    length saspgm $40  excmd $350;

    * Build the master batch pgm;
    if n_=1 then do; ①
      * Write out the batchfile into &batchloc;
      file "&batchloc\Program13.1.3b_runbat.bat"; ②

      * Build the executable command;
      excmd = catt('','');
        "%sysget(sasroot)\sas.exe",
        " -sysin '", "&batchloc",
        '\Program13.1.3b_Masterpgm.sas'); ③
      put excmd;
    end;

    * read the names of the SAS programs;
    infile c1sas truncover;
    input saspgm $40.; ④
      * Write the INCLUDE statements into the master program;
      file "&batchloc\Program13.1.3b_Masterpgm.sas"; ⑤

      * Build the INCLUDE statement to be written to the master;
      excmd = cat('%include ',
                  quote(catt("&saspgmloc",
                  '\',
                  saspgm)));
      put excmd;
    run;
  %mend makerunbat2;
  %makerunbat2(saspgmloc=&path\chapter 1\SAS Programs,batchloc=c:\temp) ⑦
```

- ① Create a batch file that will contain a single statement to be executed.
- ② The batch file is named.
- ③ The statement that invokes SAS and executes the master program is written to the batch file. When the parameters at ⑦ are used, the following batch file is created.

```
"C:\Program Files\SASHome2\SASFoundation\9.4\sas.exe" -sysin  
"c:\temp\Program13.1.3b_Masterpgm.sas"
```

- ④ The name of the SAS program that is to be included is read into the SASPGM variable.
- ⑤ The master program, which is executed by the batch program ③, will contain an %INCLUDE statement for each program read in at ④.
- ⑥ The %INCLUDE is created and written to the master program. The QUOTE function is used to surround the name and path of the SAS program with double quotes. This allows for embedded spaces in the program name.
- ⑦ These parameters will be used by the DATA \_NULL\_ step to build the master program, which will have one %INCLUDE statement for each program in the specified directory (&SASPGMLOC). The master program contains:

```
%include "C:\chapter 1\SAS Programs\Carpenter_17835TW_Program1.2.1.sas"  
%include "C:\chapter 1\SAS Programs\Carpenter_17835TW_Program1.4.sas"
```

**MORE INFORMATION:** Section 8.1.1 uses the %SYSGET function to retrieve the location of the SAS executable file.

**SEE ALSO:** Chen and Gilbert (2002) show a macro that builds a batch file that executes all SAS programs from a set of directories using a series of executions of SAS. Lopez and Knowlton (2016) set up and execute SAS batch programs.

## 13.2 Working with the Output Delivery System

The Output Delivery System (ODS) can be used to produce and control graphs, charts, and reports. These can be redisplayed through a number of means, including the use of standard Internet browsers. In the production environment this often means the generation of a large number of graphs and reports. Of course, when the numbers are large, it becomes problematic to locate, name, point to, and redisplay all of these reports, graphs, and tables in such a way as to be fairly easy for the user and fairly automatic for the programmer.

Macros are often used to control the appearance and order of reports and listings. The examples in this section represent these types of macros. This is of course the forte of the macro language, which when used with ODS can be used not only to automate, but to manage the process as well.

Much of the code that is shown in Section 13.2 is taken from longer macros and programs, and is included here to demonstrate technique rather than complete examples.

**MORE INFORMATION:** Program 6.4.1b causes a series of procedures to be executed for each level of the BY variable rather than by cycling through the BY groups for each procedure.

**SEE ALSO:** Aboutaleb (1997a) discusses a macro that you can use to control output lines when you work with proportional fonts.

Several tips and techniques are suggested by Hayden (2002) that can be used to control titles, data selection, and report presentation. A macro used to control the labels in a trend chart is presented by Bessler and Pierri (2002).

Spotts (2002) uses a short macro to document his output, and Lund (2002) demonstrates a macro to build data dictionary tables.

A series of macros that can be used to enhance reports generated with a DATA \_NULL\_ step are presented by Ewing (2002). Automated column assignment in a DATA \_NULL\_ is discussed by Squire (1999).

Cheng (2000) adds standardized titles using a macro.

Maximizing the size of the available page for a PROC REPORT is discussed by Allen (2002).

Andresen (1997) presents a macro that will split and wrap a long text variable when creating a report using a DATA \_NULL\_ step. Chen (2001) uses the DATA step to form a Table of Contents. Larsen (2001) gives a quick description of a short macro that centers text in the DATA \_NULL\_ step through the use of the %LENGTH and %EVAL functions.

Felty and Nicholson (1999) discuss a macro that produces page numbers in a variety of styles and report types (including DATA \_NULL\_). Another macro that places page numbers is discussed by Peterson (1999).

Several macros that are used to generate styles and templates are presented by Bessler (2003).

### **13.2.1 Why You Might Need to Automate with Macros**

When you create ODS output, the graphs and reports must be named and placed in an accessible location. When programs are rerun and graphs are updated or regenerated, the naming conventions must be well enough defined so that the correct graphs are replaced. The current naming conventions make this easier to do; however, constraints such as limitations within the naming of graphical catalog entries must be dealt with.

Once created, the graphs and reports must then be placed in a location that is automatically determined within the production process, and if the location does not already exist, it must be created. This automated process requires a standardized and well thought out naming convention.

In Section 12.2.8, a series of survival analyses were conducted for a variety of different models. Because each of over 150 models will generate as many as 30 graphs and tables, the naming conventions and locations were determined, to a large extent, by a model designation code. The necessary codes and output names were designated in a metadata control file. Through the use of macros these filenames and locations could be coordinated for any number of tables.

**SEE ALSO:** A macro used to control the labels in a trend chart is presented by Bessler and Pierri (2002).

Lund (2002) coordinates data dictionaries in a drill-down HTML report. Carpenter and Smith (2002) discuss the use of macro variables and macro %DO loops to name, coordinate, and link listings and graphs.

The design of directory structures used to facilitate large and/or dynamic applications is presented in Carpenter and Smith (2001).

Leprince and Li (2003) use macros to coordinate tabular output with SAS/GPLOT plots. These are then written to PDF files using ODS.

SASHELP.VCOLUMN is used by Goddard (2003) in an example that writes RTF files using ODS.

---

### **13.2.2 Controlling Directories**

The physical location for the various files must be controlled as a part of an automated process. In a directory-based OS, the directories themselves are part of this structure. Often we need tools that will detect whether a directory exists as well as the ability to create directories when they have not yet been established.

You can determine whether a specific location (in this case a directory) exists through the use of the FILEEXIST function. Program 13.2.2a assumes that we are operating in the macro environment; therefore,

the macro function %SYSFUNC is used to call the FILEEXIST function, and if needed, %SYSEXEC is used to create the new directory. This macro function can be used directly in a FILENAME statement.

#### Program 13.2.2a: Checking for and Creating a Directory

```
%macro ChkDir(DirLoc=, DirName=);
  /* if the directory does not exist, make it;
  %if %sysfunc(fileexist("&dirloc\&dirname"))=0 %then %do; ①
    %put Create the directory: "&dirloc\&dirname";
    /* Make the directory;
    %sysexec md "&dirloc\&dirname"; ②
  %end;
  %else %put The directory "&dirloc\&dirname" already exists;
    &dirloc\&dirname ③
  %mend chkdir;

filename HRLoc "%chkdir(dirloc=c:\temp,dirname=hazardratios)"; ④
```

- ① The FILEEXIST function is used to check if the directory already exists. The function returns a nonzero value if it does.
- ② %SYSEXEC is used (instead of the X statement) to execute the DOS (Windows) make directory command. Both the X and %SYSEXEC statements open, but do not close the DOS command window. Use the NOXWAIT system option if you want this window to close automatically.
- ③ The name of the directory, which now exists, is passed back out of the macro.
- ④ Since the call to %CHKDIR resolves to an existing directory ③, it can be used as a part of the FILENAME statement.  
The *fileref* defined in ④ identifies only to the directory level; however, we can complete it by specifying a specific output file so that it can be referenced in an ODS statement.

```
filename HRLoc
"%chkdir(dirloc=c:\temp,dirname=hazardratios)\HRatio&model..pdf";

ods pdf file=hrloc;
```

For the HTML destination the PATH and BODY options could also be used, and the use of the FILENAME statement avoided altogether.

```
ods html path="%chkdir(dirloc=c:\temp,dirname=hazardratios)"
body="HRatio&Model..html";
```

You can avoid the use of the DOS MakeDirectory (MD) command by using the DCREATE function, as this function is OS independent. In Program 13.2.2b the DCREATE function replaces the %SYSEXEC statement and the MD command. This has the side benefit of not opening a sub-session window.

#### Program 13.2.2b: Using the DCREATE Function

```
%macro ChkDir2(DirLoc=, DirName=);
  /* if the directory does not exist, make it;
  %if %sysfunc(fileexist("&dirloc\&dirname"))=0 %then %do;
    %put Create the directory: "&dirloc\&dirname";
    /* Make the directory;
    %sysfunc(dcreate(&dirname,&dirloc)) ⑤
  %end;
  %else %do;
    %put The directory "&dirloc\&dirname" already exists;
    &dirloc\&dirname ⑥
  %end;
  %mend chkdir2;
```

- ⑤ The DCREATE function creates the directory named in the first argument and makes it a subdirectory of the structure named in the second argument. This function returns the completed path.
- ⑥ When the subdirectory already exists, the path is returned.

Regardless of whether the directory already exists, the %CHKDIR2 macro function returns the full path.

**MORE INFORMATION:** The %CHKDIR2 macro is used in Program 13.2.3.

**SEE ALSO:** It is also possible to create folders and subfolders using the LIBNAME statement and the DLCREATEDIR system option; however, this option might have some limitations that are avoided by using the above techniques. Consult the documentation and the companion associated with your OS for more information on DLCREATEDIR. Langston (2015b) demonstrates the use of DCREATE. Directories are created and deleted using a number of different techniques in Jia (2015).

### 13.2.3 Controlling Hyperlinks

There are a number of ways to create hyperlinks within reports and graphs. The difficulty is not in their creation, but rather in their coordination. If a link is created, it has to point to an existing location. The name of the various locations must be coordinated. Doing this for one or two links manually is not too bad, but as the number of cross links increases, so does the complexity. Enter the macro language.

Remember it is about control. This control can come from a specially created control file (see Chapter 11 for more on creating and using control files) or from the data itself. In the examples in this section the data itself is used to coordinate the various levels of linking. In these examples a series of reports are to be created, with each linked to the higher report through the title and to subservient tables through the data. As you read through the four macros used in this example, pay particular attention to how the reports are named and subsequently linked.

The examples that follow create a series of interlinked reports. The first is an index report that contains links to the reports specific to each region. The report for each region highlights the clinics within that region, and the reader can drill down through the clinic to the detail report for that clinic. Implementation is through four macros. This first builds macro variable lists and calls the macros that actually build the reports. The remaining three macros create the index report (%REGINDEX). The second macro creates the region reports (%REGRPT), and the third macro creates the reports for the individual clinics (%CLINRPT). As you examine these macros, one of the primary lessons is that the naming of the files is consistent and based on the values in the data.

#### Creating the Macro Lists (%CLINRPT)

The macro %CLINRPT is used to control the overall process. The incoming data is read and macro variable lists for regions and clinics are created in the local symbol table. The macros that generate the linked reports are called from within %CLINRPT. Since macro variables stored in the symbol table for %CLINRPT are available to the macros called from within %CLINRPT, these lists do not need to be passed to the interior macros.

#### Program 13.2.3 (%CLINRPT): Creating the Macro Variable Lists

```
%macro clinrpt(loc=,dir=);
  ods escapechar='~'; ①
  proc sort data=macro3.clinics
    out=clinics
    nodupkey;
  by region clinname;
  run;

  data _null_;
    set clinics end=eof;
    by region;
    if first.region then do;
```

```

rcnt+1;
      call symputx(catt('reg',region),region,'l'); ②
end;
ccnt+1;
call symputx(catt('creg',ccnt),region,'l'); ③
call symputx(catt('cnum',ccnt),clinnum,'l');
call symputx(catt('cnam',ccnt),clinname,'l');
if eof then do;
      call symputx('regcnt',rcnt,'l'); ④
      call symputx('clncnt',ccnt,'l'); ⑤
end;
run;

proc format; ⑥
  value $regname
    '1' = 'North East'
    '2' = 'New York'
    '3' = 'Central East'
    '4' = 'South East'
    '5' = 'Mid West'
    '6' = 'Texas'
    '7' = 'Central Plains'
    '8' = 'Rocky Mtn.'
    '9' = 'South Western'
    '10' = 'Pacific Northwest';
run;

%regindex ⑦
%regrpt ⑧
%clinicrpt ⑨
%mend clinrpt;

```

- ① Because inline formatting will be used to create some of the links, an ODS escape character is declared.
- ② A list of distinct regions is created.
- ③ Lists of the clinic numbers (CLINNUM), clinic names (CLINNAME), and the region (REGION) corresponding to each are created.
- ④ The total number of regions is saved.
- ⑤ The total number of clinics is saved
- ⑥ A format matching region codes with region names is created.
- ⑦ The %REGINDEX macro will create the master table of contents (the index) for the report. Items in this index will link to the individual region reports generated in ⑧.
- ⑧ Reports for the individual regions are generated by the %REGRPT macro. Each regional report has links to the individual clinic reports generated in ⑨.
- ⑨ One report is generated for each clinic by the %CLINICRPT macro. Because clinics are nested within regions, each of these reports are linked to from their associated region reports.

For the reports shown in this section, this macro will be executed using the following macro call:

```
%clinrpt(loc=c:\temp, dir=RegionReports)
```

Here the root portion of the path is stored in &LOC and the directory name is stored in &DIR. The directory will be created by the %REGINDEX macro, if it does not already exist.

### **Creating the Report That Serves as the Index (%REGINDEX)**

The table of contents for this series of reports (MASTER.PDF) is generated using a PROC PRINT. The data values in the data set contain the links.

**Program 13.2.3 (%REGINDEX): Creating the Report Index**

```
%macro regindex;
  %local i;
  * Create the Region Index;
  data masterlist; ①
    length region $2 RegName $100; ②
    %do i = 1 %to &regcnt; ③
      region="&&reg&i"; ④
      RegName=catt("~{style [url='", ⑤
                    "&loc\&dir\RegRpt_&&reg&i...pdf']} }",
                    put("&&reg&i", $regname.));
      output masterlist;
    %end;
    *put regname=;
  run;

  ods pdf file=%ttslit(%chkdir2(dirloc=&loc,dirname=&dir)\Master.pdf); ⑥
  title1 'Master Index to the Regional Reports';
  proc print data=masterlist noobs; ⑦
    var region regname;
  run;
  ods pdf close;
%mend regindex;
```

- ① The data set to be printed (WORK.MASTERLIST) is created manually. We could have also based this data set off of the WORK.CLINICS data set created in %CLINRPT.
- ② The attributes of the two variables are specified. The length of REGNAME must be great enough to accommodate the link information ⑤.
- ③ A %DO is used to generate a series of assignment statements for each region.
- ④ The value of the variable REGION is recovered from the macro variable list.
- ⑤ The variable REGNAME will contain not only the name of the region, which is returned by the format, but also the link that points to the report for this specific region. The first four observations of WORK.MASTERLIST are shown in Figure 13.2.3a.

**Figure 13.2.3a: Selected Observations from WORK.MASTERLIST**

 **VIEWTABLE: Work.Masterlist**

	region	RegName
1	1	~{style [url='c:\temp\RegionReports\RegRpt_1.pdf']}North East
2	2	~{style [url='c:\temp\RegionReports\RegRpt_2.pdf']}New York
3	3	~{style [url='c:\temp\RegionReports\RegRpt_3.pdf']}Central East
4	4	~{style [url='c:\temp\RegionReports\RegRpt_4.pdf']}South East

- ⑥ The report that is to be created is named. The %CHKDIR2 macro (see Program 13.2.2b) establishes the path. The autocall macro %TSLIT is used to generate quotes around the resultant string.
- ⑦ PROC PRINT is used to generate the index of links.

The resulting report has one row for each region.

**Figure 13.2.3b: The Index Report Linking to Individual Regions**

region	RegName
1	North East
2	New York
3	Central East
4	South East
5	Mid West
6	Texas
7	Central Plains
8	Rocky Mtn.
9	South Western
10	Pacific Northwest

### Creating the Individual Region Reports (%REGRPT)

The region reports that are named in the index (see Figure 13.2.3a) are generated by the %REGRPT macro. This macro is fairly straightforward with a %DO loop that cycles through each of the regions. Within each report, which highlights the individual clinics within that region, a link is constructed for each clinic that allows the reader to link to the detail report for each clinic.

#### Program 13.2.3 (%REGRPT): Creating Individual Reports for Each Region

```
%macro regrpt;
%local i;
* Create a preport for each Region;
%do i = 1 %to &regcnt;
  * Step through the individual regions;
  ods pdf file="&loc\&dir\RegRpt_&&reg&i...pdf"; ①
  title1 "Visit Counts for the ";
  "%qtrim(%qsysfunc(putc(&&reg&i, $regname.))) Region";
  title2 link="&loc\&dir\Master.pdf"
         "Return to the Master Index"; ②

  proc report data=macro3.clinics(where=(region="&&reg&i"));
    column clinnum clinname n;
    define clinnum / group 'Clinic Number';
    define clinname/ group 'Clinic Name';
    define n / 'Visit Count';
    compute clinname;
      link = catt("&loc\&dir\ClinicRpt_",
                  clinnum, ③
                  '.pdf');
    call define(_col_, 'url', link); ④
  endcomp;
  run;
  ods pdf close;
%end;
%mend regrpt;
```

- ❶ The PDF file is named using the same naming convention as was used for the individual regions in the master index report (%REGINDEX). The name of the file is controlled by using the same list of macro variables (&&REG&I).
- ❷ A link back to the next higher table is inserted into the TITLE statement using the LINK= option. When building a series of interconnected reports, it is almost always a good idea to provide a link back to the calling report.
- ❸ A link to each individual clinic is constructed in a compute block inside the PROC REPORT step. The link uses the clinic number to establish the unique file identifier. For clinic number 094789 the link becomes:  
C:\temp\RegionReports\ClinicRpt\_094789.pdf
- ❹ The CALL DEFINE routine is used to assign the name of the file to the URL attribute for this report column.

The report for each region highlights the visit counts for each clinic within the region. The clinic names form links that can be used to drill down to the details for the clinic itself. The clinic detail reports are generated by the %CLINICRPT macro.

**Figure 13.2.3c: Report for the South East Region**

Visit Counts for the South East Region		
<a href="#">Return to the Master Index</a>		
Clinic Number	Clinic Name	Visit Count
033476	Mississippi Health Center	4
043320	Miami Bay Medical Center	4
046789	Tampa Treatment Complex	2
049060	Atlanta General Hospital	4

### Creating the Detail Reports for Each Clinic (%CLINICRPT)

As the lowest (most detailed) report, this is the simplest to create. The only links are those that point back to the next higher table, the region report. We do need to be careful to make sure that the names of these reports, which are specified in the ODS FILE= option, match those specified in the links in the REPORT step that generates the individual region reports.

#### Program 13.2.3 (%CLINICRPT): Creating Individual Clinic Reports

```
%macro clinicrpt;
%local i;
* Create a preport for each clinic;
%do i = 1 %to &clnct;
  * Step through the individual regions;
  ods pdf file="&loc\&dir\ClinicRpt_&cnum&i...pdf"; ❶
  title1 "Clinic Visit Details for the &cnam&i Clinic";
  title2 link="&loc\&dir\RegRpt_&creg&i...pdf" ❷
    "Return to the Region Report for this Clinic";

  proc report data=macro3.clinics(where=(clinnum="&&cnum&i"));
    column lname fname dob exam symp;
    define lname / display 'Last Name';
    define fname / display 'First Name';
    define dob / display 'Date of Birth' f=date9.;
    define exam / display 'Exam Date' f=date9.;
    define symp / display 'Symptom Code';
  run;
  ods pdf close;
%end;
%mend clinicrpt;
```

- ❶ The file is named using the clinic number for the clinic in this report.
- ❷ The LINK= option is used to establish the link back to the calling report. Notice that the &&CREG&I list is used. This list contains the region for each clinic.
- ❸ The report for this clinic is established through the use of a WHERE clause that selects for this specific clinic number.

**Figure 13.2.3d: Detail Report for the Miami Bay Medical Center**

Clinic Visit Details for the Miami Bay Medical Center Clinic				
<a href="#">Return to the Region Report for this Clinic</a>				
Last Name	First Name	Date of Birth	Exam Date	Symptom Code
Halfner	John	02MAR1947	14SEP1985	02
Johnson	Randal	29AUG1956	.	.
Most	Mat	02MAR1947	14SEP1985	02
Jackson	Ted	29DEC1956	.	.

**SEE ALSO:** The generation of hyperlinks in PROC REPORT is discussed in Carpenter (2007).

More information about creating drill-down graphics can be found in Smith (2003) and in Carpenter and Smith (2002). Hadden (2003) creates drill-down maps using SAS/GPGRAPH.

## 13.3 Working with Data

Although the macro language is not generally used to read and write data directly, it is often used to create the DATA and PROC steps needed to do so. The examples in this section deal with the generation of code that works with data.

### 13.3.1 Selection of a Top Percentage of Observations

This macro creates a data set that is a subset of an original data set, and the subsetting criterion is based on a percentage of the largest values of a particular variable. You might use this macro if you want to regularly run a report on the top 10 percent of a data set that is constantly changing size, and you don't want to manually calculate and then edit the number of observations that you want to look at.

In Program 13.3.1 PROC SQL step is used to count the number of observations with distinct values of the ID variable (&IDVAR), and the requested fraction (&PCNT) of this number is then calculated and loaded into the macro variable &IDPCNT. The first &IDPCNT observations are then written to the new data set TOPITEMS using the OBS= data set option.

#### Program 13.3.1: Subsetting the Top N Percent

```
%macro toppcnt(dsn,idvar,pcnt);
*****;
* create table pcnt for indicating &pcnt of ids           *;
*****;

proc sql noprint;
  select count(distinct &idvar) *&pcnt ❶
    into :idpcnt ❷
    from &dsn;

*****;
*   sort on descending &idvar                                *;
```

```
*****;
proc sort data= &dsn out=items;
by descending &idvar;
run;

*****;
*   keep top &IDPCNT %                                *;
*****;

data topitems;
set items(obs=%sysevalf(&idpcnt,ceil)); ③
run;
%mend toppcnt;

%toppcnt(macro3.biomass,bmtotl,.25);
```

Source: Diane Goldschmidt

- ①** The number of distinct values of the specified variable is multiplied by the requested percentage.
- ②** The number of observations to read is stored in the macro variable &IDPCNT.
- ③** The number of observations to read is used in an OBS= option. Because of the way that &IDPCNT is calculated, it might contain a non-integer value. Since this would cause an error, the %SYSEVALF function with the CEIL option is used to return the next largest integer value.

The macro %TOPPCNT counts the distinct values of the variable named by &IDVAR. If each observation does not contain a unique value for that variable, the percentage of observations in TOPITEMS might not be accurate.

**MORE INFORMATION:** The %SYSEVALF function is introduced in Section 7.3.3.

**SEE ALSO:** Gerlach and Misra (2002) demonstrate a macro that will split a large data set into N subsets.

### 13.3.2 Selection of Top Percentage Using the POINT Option

You can also use the POINT and NOBS options in the SET statement to select data subsets. The macro in Program 13.3.2 uses many of the same naming conventions as the macro in Program 13.3.1. However, the NOBS= SET statement option is used to determine the number of observations.

#### Program 13.3.2: Subsetting Using the NOBS= and POINT= Options

```
%macro selpcnt(dsn=idvar,pcnt=);
* Sort the incoming data set in descending order;
proc sort data=&dsn ①
          out=items;
by descending &idvar;
run;

* Read the first IDPCNT observations from ITEMS;
data topNitems;
  idpcnt = ceil(nobs*&pcnt); ③
  do point = 1 to idpcnt;
    set items point=point nobs=nobs; ②
    output topnitems;
  end;
  stop;
run;
%mend selpcnt;

%selpcnt(dsn=macro3.biomass,idvar=bmtotl,pcnt=.25)
```

- ❶ The data are first sorted.
- ❷ The NOBS= option creates a variable on the PDV (NOBS) that is equal to the number of observations in the data set. The value of this variable is determined during the compilation phase of the DATA step.
- ❸ The calculation of the number of observations to read (IDPCNT) is made during the execution phase of the DATA step.

Unlike the macro %TOPPCNT in Program 13.3.1, the count is based on total observations, not total number of distinct values of the ID variable (&IDVAR).

Remember that, since the value for NOBS is established when the DATA step is compiled, the variable NOBS can be used in the assignment statement ❸, which is before the SET ❹ statement, which is where NOBS is declared.

### 13.3.3 Random Selection of Observations

A variety of routines have been written to create a data subset that is based on the random selection of a subset of observations. The use of the macro language is actually secondary to the process, which is fairly simple. You should find that more sophisticated subsampling methods will be available to you if you use PROC SURVEYSELECT, which is available with SAS/STAT®.

Random selection routines select either with or without replacement. WITH replacement selection means that a given observation is eligible for selection more than once. Each observation can be selected one time, at most, when using a WITHOUT replacement criteria.

The two macros presented here represent these two selection methods.

#### Selection without Replacement

The macro in Program 13.3.3a allows the user to select the data set from which the observations are to be selected, and the fraction of observations to select. This routine will result in a subset with approximately the requested fraction of observations. Other routines that are based on conditional probabilities have also been written, and these routines can result in a more precise number of observations in the subset.

#### Program 13.3.3a: Selection without Replacement (Approximate Sample Size)

```
%macro rand_wo(dsn=,pcnt=0);

* Randomly select an approximate percentage of
* observations from a data set.
*
* Sample WITHOUT replacement;
*      any given observation can be selected only once
*      all observations have equal probability of selection.
*;

* Randomly select observations from &DSN;
data rand_wo;
  set &dsn;
  if ranuni(0) le &pcnt then output rand_wo; ❶
  run;
%mend rand_wo;
%rand_wo(dsn=macro3.clinics,pcnt=.25) ❷
```

- ❶ The RANUNI function is used to return a random number between 0 and 1. The observation is only written to the new data set RAND\_WO if the returned value is less than the requested fraction (&PCNT).
- ❷ The call to %RAND\_WO requests a 25-percent subset.

The approach used in Program 13.3.3a allows any given observation to be selected only once; however, the number of observations selected need only be approximately correct. The version of %RAND\_WO in Program 13.3.3b selects the correct number of observations by using an ARRAY to store the observation numbers that have been selected.

#### Program 13.3.3b: Selection without Replacement (Requested Sample Size)

```
%macro rand_wo(dsn=,pcnt=100);
  %local obscnt;
  %let obscnt = %obscnt(&dsn); ③
  %put obs count is &obscnt;

  * Randomly select observations from &DSN;
  data rand_wo(drop=cnt totl);
    * Calculate the number of obs to read;
    totl = ceil(&pcnt*&obscnt); ④
    array obsno {&obscnt} _temporary_; ⑤

    do until(cnt=totl);
      point = ceil(ranuni(0)*&obscnt); ⑥
      if obsno{point} ne 1 then do; ⑦
        * This obs has not been selected before;
        set &dsn point=point; ⑧
        output rand_wo;
        obsno{point}=1; ⑨
        cnt+1;
      end;
    end;
    stop;
  run;
%mend rand_wo;
%rand_wo(dsn=macro3.clinics, pcnt=.25)
```

- ③ Determine the total number of observations in the data set (&DSN) using the %OBSCNT macro, which was described in Section 11.2.6.
- ④ Calculate the number of observations to read (TOTL).
- ⑤ A temporary array, with the observation number as the index, will be used to mark those observations that have been selected.
- ⑥ Randomly generate an observation number between 1 and &OBSCNT.
- ⑦ The number 1 is used to mark that a given observation has already been read.
- ⑧ Read the selected observation using the POINT= option on the SET statement.
- ⑨ Mark this observation as having been used.

#### Selection with Replacement

Sampling with replacement allows for more flexibility and a more interesting macro. When sampling with replacement, it is possible for the resulting data set to have more observations than the original. This can be very useful when you use statistical techniques, such as bootstrapping.

The following macro, %RAND\_W, enables the user to select either the number of desired observations or a percentage. Both might result in numbers that are larger than the number of original observations. The POINT and NOBS options in the SET statement are used to randomly select the observations that are to be included in the sample.

**Program 13.3.3c: Selecting Observations with Replacement**

```
%macro rand_w(dsn,numobs=0,pcnt=0); ①
  * Randomly select &NUMOBS observations from &DSN;
  data rand_w;
    retain numobs .;
    drop numobs i;

  * Create a variable (NUMOBS) to hold number of obs
  * to write to RAND_W;
  %if &pcnt ne 0 and &numobs=0 %then %do;
    * Use the percent to calculate a number of obs;
    numobs = round(nobs*&pcnt); ②
  %end;
  %else %do;
    numobs = &numobs; ③
  %end;

  * Loop through the SET statement NUMOBS times;
  do i = 1 to numobs;
    * Determine the next observation to read;
    point = ceil(ranuni(0)*nobs); ④

    * Read and output the selected observation;
    set &dsn point=point nobs=nobs ; ⑤
    output rand_w; ⑥
  end;
  stop;
  run;
%mend rand_w;
%rand_w(dsn=macro3.clinics,pcnt=.25) ⑦
%rand_w(dsn=macro3.clinics,numobs=150) ⑧
```

- ① To determine the subsetting method, the user selects one of these parameters:  
NUMOBS= is used when a specific number of observations is desired.
- PCNT= specifies a fraction (&PCNT can be greater than 1) of the observations to be selected.
- ② The variable NUMOBS will be used in a DO loop that controls the number of observations in the output data set. Here, NUMOBS is calculated based on the number of observations in the data set (NOBS) and the desired percent (&PCNT).
- ③ When a specific number of randomly selected observations has been requested, that value is transferred to the variable NUMOBS.
- ④ POINT will be a random integer that can range from 1 to NOBS. It is important that the CEIL function be used to create the integer. Functions such as INT, FLOOR, or ROUND will assign the incorrect probabilities to the first and last observations.
- ⑤ Read the observation indicated by the POINT variable.
- ⑥ Output the observation to the new data set and continue the loop.
- ⑦ This call requests a 25% subset.
- ⑧ A specific number of observations can be requested, and this number can exceed the number in the data set itself.

**SEE ALSO:** Additional subsampling techniques, including the use of PROC SURVEYSELECT, are discussed by Chapman (2001).

### 13.3.4 Building a WHERE Clause Dynamically

Often we need to build a WHERE clause based on information contained within the macro variables. The process is fairly straightforward in that we are simply building code; however, there are a couple things that are worth a special note.

The macro %MAKECSV in Program 13.3.4a was written to dump the contents of a SAS data set to a comma-separated flat file. Since not all of the original data set is desired, a WHERE clause is built based on the values of the macro parameters. This macro was written to work against a specific data set; however, the concepts could be generalized as long as the macro parameters contain the kind of building blocks that you will need to create your own WHERE clause. These building blocks might include the ability to work with:

- ❶ a list of values
- ❷ a specific value
- ❸ a logic flag.

#### Program 13.3.4a: Building a WHERE Clause Dynamically

```
%macro makecsv(dsn=,list=,reg=,miss=,chkvar=wt,no=no);
options &no.mprint &no.mlogic &no.symbolgen; ❶
%* ❶LIST one or more blank separated clinic numbers;
%*      Blank to get all clinics;
%* ❷REG Region of interest
%*      Blank to select all regions
%* ❸MISS are observations with missing weights allowed?
%*      ok      missing ok
%*      <other> nonmissing only;

%local qlist wclause; ❷

%* Quote the words in the list and separate
%* them with commas;
%let qlist = ❸
  %str('%')%sysfunc(tranwrd(&list,%str( ),%str(',')'))%str('%');

%* Build the WHERE clause;
%if &miss=ok %then %let wclause = &chkvar ge ._; ❹
%else %let wclause = &chkvar gt .z;
%if &reg ne %then %let wclause = &wclause & region="%reg"; ❺
%if %bquote(&list) ne %then
  %let wclause = &wclause & clinnum in(&qlist); ❻

data _null_;
  set &dsn(where=%unquote(&wclause)); ❾
  file "c:\temp\makecsv.csv" dlm=',';
  if _n_=1 then put "clinicnumber,clinicname,region,&chkvar";
  put clinnum clinicname region &chkvar;
  run;
%mend makecsv;
```

- ❶ Macro variables are declared as local. &WCLAUSE is created and then used in the WHERE statement.
- ❷ The list of values will be used as character values in an IN operator; therefore, they need to be quoted and comma separated. The TRANWRD function is used to convert embedded blanks to a comma. This macro assumes that the list is to be applied to a character variable. The ‘,’ that separates the words might trigger a warning concerning identifiers following a quoted string, and is an artifact of the parsing process.

- ⑥ &MISS is used to determine whether missing values of the variable to be checked (&CHKVAR) are to be included. In this macro the WHERE clause will always have this clause; of course &CHKVAR ge .\_ will eliminate nothing. The assumption is made that this variable is numeric.
- ⑦ If a value for the variable REGION is supplied, & region="&reg" is added to the clause.
- ⑧ When one or more values are supplied in &LIST the quoted values (&QLIST) are used with the IN operator, and & clinnum in (&qlist) is appended to the clause.
- ⑨ Because the %STR function was used to mask the single quotes, the %UNQUOTE function must be used so that the clause can be applied correctly. This is a timing issue, and the %UNQUOTE ensures that all macro processing is completed before the WHERE clause is compiled.
- ⑩ Debugging options can be turned on or off through the use of the &NO parameter. The default value, NO, turns them off.

Notice that the WHERE clause was built in a macro variable, which was then used in the WHERE= option. Because of timing issues between the macro facility and how the WHERE clause is applied, %IF-%THEN/%ELSE statements can cause problems when applied directly inside of what will become the WHERE clause ⑨.

The macro call is as follows:

```
%makecsv(dsn    = macro3.clinics,
         list   = 051345 057312,
         reg    = 5,
         miss   = ok,
         chkvar= death,
         no     = no) ⑩
```

It results in the following WHERE= option:

```
(where=(death ge ._ & region="5" & clinnum in('051345','057312')))
```

The macro %MAKECSV in Program 13.3.4a requires the user to fully specify the variables that are to be checked. In Program 13.3.4b the macro %FINDOUTLIERS is also used to find observations that meet a criteria, which has been passed into the macro through the macro parameters. However, unlike %MAKECSV, this macro is more general in that the user can select the variable names and the criteria for selection.

Macro parameters are as follows:

**dsn**

name of the data set to check

**prefix**

name or prefix letters of the variable(s) to use to determine the outliers

**value**

value of the variable to use as the determining criteria

**op**

comparison operator, for example “eq” or “=” (either the mnemonic or symbol may be used)

**logicop**

logic operator joining comparisons (may be either AND or OR)

The following call to %FINDOUTLIERS selects observations where any variable whose name starts with “BM” and has a value greater than or equal to 6:

```
%findoutliers(dsn=macro3.biomass,prefix=bm, value=6, op=ge,logicop=or)
```

The resulting WHERE clause becomes the following:

```
(where=(BMCRUS ge 6 or BMMOL ge 6 or BMOTHR ge 6 or BMPOLY ge 6 or BMTOTL  
ge 6))
```

All of the macro's parameters are named parameters; however, only &OP and &LOGICOP have default values.

#### Program 13.3.4b: Building the WHERE Using a Macro Variable List

```
%macro findoutliers(dsn=,prefix=,value=,op=ge, logicop=or);  
%local i wclause;  
proc contents data=&dsn noprint  
            out=contdsn; ①  
run;  
  
data _null_;  
set contdsn;  
if name =: %upcase("&prefix");  
cnt+1;  
call symputx(catt('var',cnt),trim(name),'l'); ②  
call symputx('varcnt',cnt,'l'); ③  
run;  
  
%if varcnt ge 1 %then %do;  
    %let wclause= &var1 &op &value; ④  
    %if varcnt gt 1 %then %do i = 2 %to &varcnt;  
        /* Build the where clause; ⑤  
        %let wclause = &wclause &logicop &&var&i &op &value;  
    %end;  
  
data outliers;  
set &dsn(where=(&wclause)); ⑥  
run;  
%end;  
%mend findoutliers;  
%findoutliers(dsn=macro3.biomass,prefix=bm, value=6, op=ge,logicop=or)
```

- ① PROC CONTENTS is used to create a list of the names of the variables in &DSN.
- ② The variable names that meet the criteria in &PREFIX are stored in the macro variables of the form &&VAR&i. The SYMPUTX routine forces these macro variables into the local symbol table.
- ③ The number of macro variables of interest is saved.
- ④ The first condition is assigned to the WHERE clause.
- ⑤ Additional conditions are attached to the WHERE clause as they are needed. Successive comparisons are chained together using the specified logical Boolean operator (&LOGICOP).
- ⑥ The WHERE clause is applied. Unlike the WHERE clause generated in Program 13.3.4a, no macro quoting was used to generate this clause; consequently, the %UNQUOTE is not needed.

**SEE ALSO:** Curtis Smith (1999) builds a WHERE clause with a LIKE operator.

# **Chapter 14: Miscellaneous Topics**

<b>14.1 More on Triple Ampersand Macro Variables .....</b>	<b>397</b>
14.1.1 Overview of Triple-Ampersand Macro Variables .....	398
14.1.2 Selecting Elements from Macro Arrays .....	398
<b>14.2 Doubly Subscripted Macro Arrays .....</b>	<b>399</b>
14.2.1 Subscript Resolution Issues for a Simple Case .....	400
14.2.2 Naming Row and Column Indicators .....	400
14.2.3 Using the &&&VAR&I Variable Form.....	402
14.2.4 Using the %SCAN Function to Identify Array Elements .....	404
<b>14.3 Programming Smarter .....</b>	<b>405</b>
14.3.1 Efficiency Issues .....	405
14.3.2 Programming with Style.....	407
14.3.3 Macro Programming Best Practices.....	409
14.3.4 Debugging Your Macros .....	411
14.3.5 Traps: DATA Step Code versus the Macro Language .....	412
14.3.6 Little Things with a Big Bite .....	417
<b>14.4 Understanding Recursion in the Macro Language .....</b>	<b>425</b>
<b>14.5 Determining Macro Variable Scopes.....</b>	<b>427</b>
14.5.1 Nested or Layered Symbol Tables .....	427
14.5.2 Macro Parameters .....	427
14.5.3 Macro Variables Created with %LET and %DO.....	428
14.5.4 Macro Variables Created with the SYMPUT and SYMPUTX Routines.....	428
14.5.5 Macro Variables Created in a PROC SQL Step Using the INTO: Operator .....	429
<b>14.6 Controlling System Initialization and Termination .....</b>	<b>429</b>
14.6.1 Controlling AUTOEXEC Execution .....	430
14.6.2 Saving the Global Symbol Table.....	431
14.6.3 Executing Initialization and Termination Statements .....	431
<b>14.7 Protecting Macros and Controlling Their Execution .....</b>	<b>432</b>

This chapter contains a number of eclectic topics. These include the use of arrays with more than one index, more on the use of triple ampersands, the use of recursion in the macro language, and a number of issues regarding programming techniques that you should be aware of as you begin to write more sophisticated macros.

For the examples in this chapter and indeed for all of the code examples throughout the book, if you want to execute these sample programs, then be sure to follow the setup instructions. Remember that all of the data sets and programs are available for download, so you do not need to retype either the code or the data. For instructions on accessing and setting up the programs and data, see the “Example Code and Data” section within this edition’s “About This Book” front matter.

---

## **14.1 More on Triple Ampersand Macro Variables**

There are a number of techniques that, although not used every day, are important to the macro programmer. Some of these techniques are specialized enough that when they are known to the programmer, they can save hours of programming effort.

**MORE INFORMATION:** The use of triple ampersand macro variables is described in Section 11.1.1. This macro variable form is also used in Program 11.2.4.

**SEE ALSO:** Matise (2015) gives a comprehensive discussion of multiple ampersand macro variable resolution.

### 14.1.1 Overview of Triple-Ampersand Macro Variables

The most common use of triple-ampersand macro variables is when you want to store the name of a macro variable in another macro variable. The examples in this section show, with increasing complexity, the use of the triple ampersand.

We have already seen in earlier sections of this book, numerous examples of the use of the `&&VAR&J` construct. Although the ampersands are not all adjacent, this is effectively still a triple-ampersand macro variable. Regardless of whether the ampersands are all adjacent as in the examples in this section, or whether they are indexed, we still have a macro variable that resolves to another macro variable that resolves to the value of ultimate interest.

If you are already comfortable with the use of the `&&VAR&J` form of the triple ampersand, remember that it references a list of numbered macro variables, and as such it mimics a macro array. If that array had only one element, you would not need the index variable, and the macro variable could be rewritten as `&&&VAR`. Essentially then the `&&&VAR` form of a macro variable is nothing more than an array of one. It is a placeholder that contains the name of the macro variable of interest.

**SEE ALSO:** Yindra (1997) has an example that uses a `&&&` macro variable in a `TITLE` statement. He then expands the topic with more examples in Yindra (1998).

The macro presented by Widawski (1997a) to create a list of files is generalized by using `&&&` macro variables.

Yarbrough (2000) combines the `&&&` construct with the `&&VAR&J` form with an example that uses the macro variable form of `&&&VAR&J`.

An example using six ampersands can be found in Gerlach (1997).

*SAS 9.4 Macro Language: Reference, Fourth Edition* gives an efficiency tip that uses triple ampersands (pp. 147-148).

### 14.1.2 Selecting Elements from Macro Arrays

The macro `%GETKEYS` shown in Program 14.1.2 is taken from an application that contains a series of data sets that exist with slight variations in different libraries. The data sets are similar enough so that many of the same processing programs can be used across libraries. Primarily, they differ in the names of the key (BY) variables. Globalized macro variable arrays are created that store the names of the data sets and the associated key variables for each *libref*. This tool allows the user to return the key variables from these macro variable lists, given the library and member name. This technique gives you the ability to access a specific member list without knowing the index value. Table 14.1.2 shows some typical macro variable lists, which might include:

**Table 14.1.2: Typical Macro Variable Lists**

libref	Number of data sets	Data set names	Key variables
final ❶	&finalent ❸	&finaldb1, &finaldb2, . . .	&keys1, &keys2, . . .

Given the data set name (for example, FINAL.DEMOG) you need to be able to retrieve the associated key variables. The macro `%GETKEYS` uses the *libref* (which for our purposes is also the root portion of the

macro variable list), and data set name to look up the index value of the data set and use that value to retrieve the associated list of key variables. This list is then stored in the macro variable &KEYVARS.

#### Program 14.1.2: Retrieving from a Macro Variable List without Knowing the Index Value

```
%macro getkeys(inlib①,indsn②);
/* getkeys.sas
* RSmith
* Macro to get the key variables for selected library & member.
* Assumes that the global macros for the databases & keys
* are created.
* Outputs a global macro variable KEYVARS.
*/
%global keyvars;
%do k = 1 %to &&&inlib.cnt; ③
    %if %upcase(&indsn②) = %upcase(&&&inlib.db&k④) %then
        %let keyvars = &&keys&k; ⑤
%end;
%mend getkeys;
```

Source: Richard O. Smith, Science Explorations

#### Program 14.1.2 (SAS Log): Showing a Call to %GETKEYS

```
49  %let finaldb1 = demog; %let keys1=subject;
50  %let finaldb2 = medhist; %let keys2=subject medhisno seqno;
51  %let finalcnt = 2; ③
52
53  %getkeys(final①,medhist②)
54  %put &keyvars;
KEYVARS=subject medhisno seqno ⑤
```

- ① The *libref*(FINAL) is also used as part of the name of the macro variable that counts the number of data sets in this library ③.
- ② &INDSN is used to store the name of the data set.
- ③ The list of data set names in this *libref* is stepped through one at a time. The number of data sets (elements in this pseudo array) is stored in a macro variable whose name is made up in part with the name of the *libref*. &&&inlib.cnt → &finalcnt → 2
- ④ The list of data set names is stored in macro variables that are formed using the *libref* and followed by DB and then a number, such as &FINALDB2. This array is indirectly referenced by using &&&inlib.db&k. For &K=2, &&&inlib.db&k → &FINALDB2 → MEDHIST
- ⑤ The key variables for this data set are assigned to the macro variable &KEYVARS.

## 14.2 Doubly Subscripted Macro Arrays

In numerous examples elsewhere in this book, a series of macro variables have been effectively used as a macro array. Although the macro language has no true macro arrays, you can create pseudo macro arrays by creating macro variables in the form of &&VAR&i (see Section 11.3 for introductory examples). These macro variables create a vector or what is essentially an array with a single subscript.

A *doubly subscripted array* describes a matrix of values or the rows and columns of a SAS data set or table. This enables you to store all of the values that are contained within a table in a single set of macro variables. You can approach the subscripting problem in several ways. Initially, one might try creating a logical extension of &&VAR&i as &&&&VAR&&i&j, where &i and &j indicate the desired row and column. Life, however, is never quite this simple, and there are complications with this approach. Fortunately, there are solutions (or this section would not have appeared in the book).

**SEE ALSO:** Noda, Kraemer, and Periyakoil (2000) use macro indexes to generate a pseudo matrix in the DATA step. Matise (2015) discusses several extensions involving multiple ampersands, including the resolution of the `&&&VAR&I` form of macro variable reference.

### 14.2.1 Subscript Resolution Issues for a Simple Case

Assume that you want the macro variable `&VAR34` to resolve to 5.62. That is, the fourth variable (column) in the third observation (row) has the value of 5.62. Assign the macro variables as follows:

```
%let i=3;
%let j=4;
%let var34 = 5.62;
```

The sequence of resolution is shown in Table 14.2.1a, however because the macro variable is improperly referenced, the resolution fails.

**Table 14.2.1a: Resolution of an Improperly Specified Reference**

Pass number	Macro Variable Reference	Resolves to
1	<code>&amp;&amp;&amp;&amp;var&amp;&amp;i&amp;j</code>	<code>&amp;&amp;var&amp;i4</code>
2	<code>&amp;&amp;var&amp;i4</code>	<code>&amp;var&amp;i4</code>

The resolved value of `&J` has become a part of the name of the macro variable `&I`. Because `&I4` is undefined, warnings are issued. In this example, you need to separate the `&I` from the 4. To do this, you can use a period to terminate the variable `&I`. The macro variable becomes `&&&&VAR&&I.&J`, and the sequence of resolution is shown in Table 14.2.1b.

**Table 14.2.1b: Removing the Resolution Ambiguity**

Pass number	Macro Variable Reference	Resolves to
1	<code>&amp;&amp;&amp;&amp;var&amp;&amp;i.&amp;j</code>	<code>&amp;&amp;var&amp;i.4</code>
2	<code>&amp;&amp;var&amp;i.4</code>	<code>&amp;var34</code>
3	<code>&amp;var34</code>	5.62

In this simple case, it would have been easier to write `&&&&VAR&&I.&J` as `&&VAR&I&J`, which resolves in only two passes.

This approach works fine as long as the number of rows and columns is fewer than ten. Larger numbers can result in naming conflicts. Does `&VAR345` refer to row 34 and column 5 or row 3 and column 45? For small tables, this is not a problem. Fortunately, a more robust method exists for larger tables.

**SEE ALSO:** Bryher (1997a) and Geary (1997) both use more sophisticated examples of doubly subscripted macro arrays. Geary uses the form `&&&&VAR&&J&K`, while Bryher uses `&&&VAR&J&K`.

### 14.2.2 Naming Row and Column Indicators

Rather than combining the row and column indicators into a single number, they can be kept separate. The macro variable `&R3C4` completely specifies the location that the value came from without conflict. Further, it can be referenced without using quadruple ampersands, for example, `&&R&I.C&J`. Notice that the dot (.) is still needed following the `&I`.

```
%let i=3;
%let j=4;
%let r3c4 = 5.62;
```

The sequence of resolution is shown in Table 14.2.2.

**Table 14.2.2: Resolution of a Macro Variable with Two Subscripts**

Pass number	Macro Variable Reference	Resolves to
1	&&r&i.c&j	&r3c4
2	&r3c4	5.62

The macro %BUILDMATRIX shown in Program 14.2.2 reads a SAS data set (table) and builds a macro variable for each numeric value. The number of numeric variables (and their names), as well as the number of data rows, is unknown. Each data value can be uniquely identified by a combination of the row and column number.

**Program 14.2.2: Building a Matrix of Data Values (Two-Dimensional Array)**

```
%macro buildmatrix(dsn=);
data _null_;
  set &dsn end=eof;
  array vlist {*} _numeric_; ①
  length name $18;
  i+1; ②
  if eof then call symputx('rowcnt',i,'l'); ③
  if i=1 then call symputx('colcnt',dim(vlist),'l');

  *** Store values for this row;
  * Build the base for the macro vars for this row;
  mbase = catt('r',i,'c'); ④

  * Step through the values for this observation;
  do j = 1 to dim(vlist); ⑤
    * Save the value for this row and column;
    call symputx(catt(mbase,j),vlist{j},'l'); ⑥

    * Save the variable name;
    if i=1 then do;
      call vname(vlist(j),name); ⑦
      call symputx(catt('vname',j),name,'l');
    end;
  end;
  run;

/* Show one element of the matrix; ⑧
%let rr = 4;
%let cc = 3;
%put Row &rr and col &cc (&&vname&cc) is %left(&&r&rr.c&cc);

%derval(maxrow=&rowcnt,maxcol=&colcnt) ⑨

%mend buildmatrix;
```

- ① The variables of interest are loaded into the array VLIST.
- ② The SUM statement is used to create a row (observation) counter.
- ③ &ROWCNT stores the number of observations, and &COLCNT stores the number of columns. These macro variables are often useful when stepping through the array in a %DO loop ⑨.
- ④ The variable MBASE is used to store the first portion of the macro variable name. For the third row this variable would take on the value of: 'r3c'.
- ⑤ A DO loop is used to step through the columns represented by the VLIST array.
- ⑥ The SYMPUTX routine is used to load the data value into the appropriate macro variable from within the DO loop. The j<sup>th</sup> variable (column) is stored and the value of J is appended onto MBASE forming the macro variable name that holds the data value.
- ⑦ For the first row (observation) the VNAME routine is used to capture the variable name that is associated with this column. The name is then stored in a macro variable (&VNAMEj).

- ❸ As a demonstration, the %PUT statement is used to access one of the values in the matrix of macro variables (row 4 and column 3). The %PUT statement causes the following line to be written to the SAS Log.

#### Program 14.2.2 (SAS Log): %PUT Statement Results

```
Row 4 and col 3 (Weight) is 102.5
```

- ❹ Usually, some type of loop is used to step through the macro array. The macro %DBVAL, which is a part of Program 14.2.2, displays each value in the table by stepping through the list of macro variables.

#### Program 14.2.2 (Continued): Stepping through the Doubly Subscripted Array

```
%macro dbval(maxrow=,maxcol=);
  %put row col Variable value;
  %do row = 1 %to &maxrow;
    %do col = 1 %to &maxcol;
      %put &row     &col     &&vname&col   &&r&row.c&col;
    %end;
  %end;
%mend dbval;
```

The macro call ❹ passes in the maximum number of rows and columns. These maximums are then used with two %DO loops. A %PUT statement writes out the individual values along with the variable name. The following are the first few lines that are written to the SAS Log by this %PUT:

row	col	Variable	value
1	1	Age	14
1	2	Height	69
1	3	Weight	112.5
2	1	Age	13
2	2	Height	56.5
2	3	Weight	84

**SEE ALSO:** Sun (1998) uses the form &&VAR&I&J to control two dimensional arrays, and Thornton (1999) uses the macro variable form &A&B&C&I&J.

---

### 14.2.3 Using the &&&VAR&I Variable Form

Rather than identifying both the row and column with numbers as was done in Sections 14.2.1 and 14.2.2, you can also use names to identify the columns. This approach can be especially helpful when there are a series of named and coordinated vectors of values. In this approach the &&&VAR&I form can be used to create an indirect reference to an array of macro variables. In the examples in Sections 14.2.1 and 14.2.2 two subscripts are used to identify the row and column associated with a value within a matrix. The approach shown here replaces either the row or column identifier (usually the column identifier) with a name.

In Table 11.3.1 the data set MACRO3.DBDIR is used as a control file that contains the name of a series of data sets and their associated BY variables. When used to build a series of macro variables, the corresponding lists can be used to drive a process. In Program 14.2.3a these lists are used to control a series of PROC PRINTS.

#### Program 14.2.3a: Controlling a Process with Macro Variable Lists

```
%macro PrtLists;
%local i;
/* Create lists;
proc sql noprint;
  select dsn, keyvar
  into :dsn1-, ❶
```

```

:keyvar1-
  from macro3.dbmdir;
%let dsncnt=&sqlobs;
quit;
%do i = 1 %to &dsncnt; ②
  %put %str( ③
    title1 "Showing Data Table &&dsn&i";
    proc print data=macro3.&&dsn&i; ④
      by &&keyvar&i;
      run;
  );
%end;

%mend Prtlists;

```

- ➊ Two macro variable lists are generated: one for the data set names and one for the list of BY variables associated with the corresponding data set.
- ➋ A %DO loop is used to step through the lists.
- ➌ A %PUT statement along with a %STR quoting function is used to test the generation of the PROC PRINT step.
- ➍ The data set name and its corresponding BY variables are specified using the macro list references.

In Program 14.2.3a the names of the variables in the control file (and therefore the names of the macro variable lists), as well as how many items there are within each list, is known to the programmer when writing the PROC PRINT step. Sometimes these names of the macro variable lists are not known until the execution of the %DO loop, and the programmer must use an indirect reference to the array itself. In the macro %MATRIXPRINT, the PROC PRINT is again to be executed; however, in this example the programmer of %MATRIXPRINT does not know the names of the macro variable lists that are to be used within the macro. To solve the naming issue, the names of the macro variables are passed rather than the values themselves.

#### Program 14.2.3b: Using Macro Lists of Unknown Names

```

%macro PrtLists;
  %local i;
  /* Create lists;
  proc sql noprint;
    select dsn, keyvar
      into :dsn1-,
        :keyvar1-
        from macro3.dbmdir;
  %let dsncnt=&sqlobs;
  quit;
  %matrixprint(dlist=dsn,vlist=keyvar,listcnt=dsncnt) ⑤
%mend Prtlists;
%macro matrixprint(DList=,VList=,ListCnt=);
  %local i;

  %do i = 1 %to &&listcnt; ⑥
    proc print data=macro3.&&&dlist&i; ⑦
      by &&&vlist&i; ⑧
      run;
  %end;

%mend matrixprint;

```

The writer of %MATRIXPRINT only knows that the macro will execute against macro variable lists, but not the names of those lists. Consequently, the parameters pass only the names of the lists not the lists themselves.

- ❸ The macro %MATRIXPRINT is called. The names of the macro variable lists are passed into the macro along with the name of the macro variable that contains the count of the list elements.
- ❹ &&&LISTCNT resolves to &DSNCNT, which resolves to the number of items in the lists.
- ❺ When &I=1, &&&DLIST&I resolves to &DSN1, which returns the name of the first data set DEMOG.
- ❻ The first two ampersands delay the resolution so that &VLIST can be resolved to the name of the list containing the BY variables. When &I=1, &&&VLIST&I resolves to &KEYVAR1, which then resolves to SUBJECT.

**SEE ALSO:** Indirect array references in the form of &&&VAR&J are used by Kunselman (2001) in a SAS/IntrNet example.

#### 14.2.4 Using the %SCAN Function to Identify Array Elements

When working with a list of names, the %SCAN function is often used to identify the names of the variables of interest. Each of these words is then saved as a macro variable. Rather than save the individual words of a macro variable as individual macro variables, you can instead continue to use the %SCAN function directly when you need the individual values. The concept of using the %SCAN function to replace a macro array can be applied to many of the examples where a macro array is formed (&&VAR&I).

When you start with a horizontal list of values, you will often use the %QSCAN function to parse the list. In a vertical list, you could refer to the third element by using &var3. Instead, for a horizontal list, you would reference the third element as %scan(&varlist, 3, %str( )). Obviously this is more typing, but it does avoid generating the vertical list of macro variables. This second approach is used in the macro %MATRIXPRINT in Program 14.2.4. In this macro a horizontal list of values is stored in a macro variable, and instead of passing this list of values, the name of the list is passed into %MATRIXPRINT.

##### Program 14.2.4: Passing the Name of a Horizontal List of Values

```
%macro PrtLists;
  %local i;
  /* Create lists;
  proc sql noprint;
    select dsn, keyvar
      into :dsnlst separated by '|', ❶
          :keylst separated by '|' ❷
        from macro3.dbmdir;
  %let dsncnt=&sqlobs; ❸
  quit;
  %matrixprint(dlist=dsnlst,vlist=keylst,listcnt=dsncnt) ❹
%mend Prtlists;
%macro matrixprint(DList=,VList=,ListCnt=);
  %local i;
  %do i = 1 %to &&&listcnt; ❺
    proc print data=macro3.%qscan(%unquote(&&&dlist,&i,%str(|))); ❻
      by %qscan(&&&vlist,&i,%str(|)); ❾
      run;
  %end;
%mend matrixprint;

%prtlsts
```

- ❶ A list of ‘|’ separated data set names is stored in &DSNLST.
- ❷ The list of key variables for each of the data sets is also stored in a horizontal macro variable list (&KEYLST).
- ❸ The number of items in each list is also stored.
- ❹ The macro %MATRIXPRINT is called by passing the names of the two lists and the number of items within each list.

- ➅ The %DO loop will cycle through items in the list. &&&LISTCNT resolves to &DSNCNT.
- ➆ The list of data set names is parsed inside of the %DO loop. The %QSCAN function retrieves the &I<sup>th</sup> word from the list of data set names (&&&DLIST), where &&&DLIST resolves to &DSNLST, which resolves to the list of data set names separated by a '|'.
- ➇ The list of BY variables associated with this data set is retrieved from &&&VLIST, which resolves to &KEYLST, which resolves to the '|' separated lists of variable names.

**SEE ALSO:** Flavin and Carpenter (2003) discuss the use of %SCAN as an array element in a SAS/GRAFH example.

## 14.3 Programming Smarter

As complex as the macro language is, there are any number of ways to solve most problems. Of course some of the solutions are better than others. Fortunately there are techniques and strategies that you can use to maximize your efficiencies as a macro programmer. These range from the style that you use to program, to your understanding of how the macro language interacts with the rest of SAS. Understanding these techniques will enable you to program smarter.

### 14.3.1 Efficiency Issues

When you use the macro language, it is easy to get carried away and to overuse it. This is especially a problem for SAS users who are new to macros. They tend to use macro language elements at the least excuse and often when they are not at all needed. Some SAS sites have restricted the use of the macro language for this reason. Restrictions should not be necessary if you program smarter and more efficiently. A number of papers have been written about various efficiency issues and programming techniques (Culp 1991, Norton 1991; O'Connor 1992; Tindall 1991, and Westerlund 1991).

Information in Chapter 11, “Writing Efficient and Portable Macros,” in *SAS 9.4 Macro Language: Reference, Fourth Edition* directly addresses these issues. This well-written chapter covers all of the major issues regarding macro efficiency and should be consulted for details not covered in the following summary.

Remember during the discussion in this section that there is more than one kind of efficiency. Usually, one thinks of efficient code in terms of its execution by SAS, but you should also consider programmer efficiency (time to code and time to maintain code). Macro code can be very difficult to maintain (especially if you are maintaining someone else’s code). A program that is down a week for maintenance can completely undo any savings realized by fast execution.

The following are some thoughts and tips on macro efficiency.

### Macro Language: To Use or Not to Use

The macro language is very useful and powerful for situations that require:

- conditional execution of blocks of code or procedures
- the storage of reusable or generalized code
- a block of constant code to be re-executed multiple times
- code to be written dynamically with parameters that depend on data values
- a series of repetitive tasks

The use of macros might be inappropriate or inefficient when any of the following occur:

- Macro code is used just because the language is fun to program.
- Macros are used to store constant text (see the macro %COMMENT in Section 3.1.2).
- Macros include extensive use of the %WINDOW and %DISPLAY macro statements to interact with the user. (SAS/AF screens are much more flexible and can be easier to develop).

## %MACRO versus %INCLUDE

Reusable code can be stored either within a macro or in a file that is retrieved using the %INCLUDE statement. In addition, the macros themselves can be stored within macro libraries. There are several considerations when choosing the best techniques for your particular needs.

When macro-level statements are required and macro parameters are not required, the retrieval of the code will probably be quicker with the use of the %INCLUDE statement. However, you should also take into consideration the programming effort needed to establish and maintain the necessary *fileref*s (as opposed to the use of an autocall library). From a practical point of view, I have not noticed a big difference between the two in terms of speed of execution.

If the code that is to be retrieved includes macro-level programming statements and particularly, macro definitions, the use of macro libraries offers some distinct advantages. Remember that macros retrieved through %INCLUDE will be compiled each time the %INCLUDE is executed, while using a macro library, such as an autocall library, allows the macro to be compiled once and then the compiled version can be reused.

## Nested Macro Definitions

A *nested macro definition* (see Section 5.1.5) is one that is defined inside of another macro. This programming *technique* is not as uncommon as it should be—it is rarely, if ever, necessary, and it is almost always inefficient. Defining macros this way has two efficiency problems:

- The macro definition might be difficult to find for maintenance.
- The nested macro will be recompiled **every** time the outer macro is called.

## Statement- and Command-Style Macros

Named-style macros always begin with a percent sign (%). This makes it easy for the macro processor (and the programmer) to find the names of the macros to be executed. Statement- and command-style macros (which are not discussed in this book) do not use the percent sign, and therefore additional resources are required to locate the macro calls. When they debug programs, few programmers will be looking for statement-style macros, and this can also lead to programmer confusion. This macro style is not discussed in this book.

## Multiple Ampersands

Programmers who use the macro language quickly become accustomed to seeing and using macro variable references of the form &VAR. Programmers who write dynamic code-building macros will become familiar (although sometimes begrudgingly) with &&VAR&I macro variables. When the code calls for three or more ampersands together, you might want to try to rethink your code, as there might be a better or easier way to handle the situation (see Section 14.2).

Macro variables of the form &&&VAR can, however, provide some efficiency savings in certain circumstances. This is because &&&VAR is an indirect macro variable reference (see Program 11.1.1c for a detailed explanation and example).

## Macro Quoting

The entire concept of macro quoting (see Section 7.1) is difficult for many macro programmers (I hesitate to say **most**, but I believe that **most** might be more accurate). Fortunately, macro quoting is not needed in most programming situations. When it is needed, ask yourself if you can code the task in a different way. If quoting is required, try to use one of the basic quoting functions such as %BQUOTE or %NRSTR.

## Macro System Options

Sections 3.3 and 8.4 discuss several system options that are associated with the macro facility. As a general rule, when system options are turned on, additional system resources are required. Some of these options are necessary; however, many options are situational. A summary of some of these system options follows:

**MACRO**

If you are not using any portion of the macro facility, it can be turned off. As a general rule, you should leave it on because some statements that are not thought of as part of the macro language actually need it to be available.

**MLOGIC, MPRINT, and SYMBOLGEN**

These options are used to display macro text. When a program is placed into production, there might no longer be any need to have these turned on. Because they can generate a great deal of text in the SAS Log, programs run more efficiently with these options turned off.

Often the MPRINT and NOMPRINT options are paired with the SOURCE and NOSOURCE options. For example, programs that have SOURCE turned off would probably also have MPRINT turned off.

**CMDMAC and IMPLMAC**

Used to turn on command- and statement-style macros, these should be off unless you really, really must use one of these styles of macros. These options can cause a large efficiency loss.

**MAUTOSOURCE, MRECALL, and SASAUTOS**

These options are used when macros are stored in AUTOCALL libraries. These libraries can be very convenient for the programmer, but some overhead is required to search the libraries. Turn off MAUTOSOURCE only if you do not want to use the AUTOCALL facility. *CAVEAT:* Turning off the AUTOCALL facility is rarely a good idea. When turned off, all of the AUTOCALL macros supplied by SAS, such as %LEFT, %TRIM, and %VERIFY, become unavailable as well.

**MSTORED and SASMSTORE**

When macros become permanent and are placed into production, a compiled version can be stored. Because these macros are only compiled once, you can realize a savings through the use of these libraries. If you are not using stored compiled macros, turn off MSTORED.

**Session Compiled and Compiled Stored Macros**

Production macros can be compiled and stored in a library as compiled stored macros. Once stored, these macros will not need to be compiled again. During a session, SAS will automatically store the compiled version (session compiled) of any called macro that is not already compiled. If you have macros in production that are not changing, they can be compiled and stored in a library. This is even more efficient than storing them uncompiled in an autocall library. See Section 10.3 for more on compiled stored macros.

**Use of the Autocall Facility**

Macros that are used by a number of programs or are in a state of flux can be stored in an autocall library (see Sections 3.3.4 and 10.4). This prevents the generation of multiple versions of the macro, and it removes the macro definition from the calling program itself. Macros defined in a calling program must be recompiled every time that program is executed, even if it has already been executed during that session. By moving the macro to an autocall library, it will only have to be compiled once in a session.

**14.3.2 Programming with Style**

There are a few things that you can do to make your programming life easier. The value of many of the comments in this section will be personal. Evaluate each against your programming needs, the way that you program, and your programming goals.

**Develop a Personal Pattern of Coding**

Try to build a personal pattern of coding. Determine which combinations of the following ideas work best for you, and apply them when developing your own style.

**Indentation**

Use indentation or tabs to indicate changes in types of statements. For example, indent the statements inside of a %DO block.

**Capitalization**

Develop a capitalization scheme. Consider capitalizing all macro statements and using lowercase for Base SAS language statements (Fehd, 2001).

**Naming conventions**

Think about and use naming conventions that match the flow of the program. Reusing patterns of names might make some macro programming tasks easier (Carpenter and Smith, 2001).

**Consistency**

Try to be consistent in naming conventions used both within and across macros. Consistency can help you remember what a macro does.

**Documentation**

Use comments liberally. Note each program modification in a comment header section such that the date on the modification note matches the date of the file (see Section 5.4.1). Within a macro the use of PL1 style comments /\* . . . \*/ is recommended (see SAS Usage Note 32684).

**Header Text**

Create a standard text header section in each program that at least notes the program name, author, purpose, inputs, and outputs (Carpenter, 2003).

**SEE ALSO:** Henderson and Carpenter (2012) has additional comments on macro coding style.

**Consider Program Organization**

Organize your program and your programming task by taking into consideration your programming needs and objectives.

- Plan out what you are going to do in your program (before you start coding).
- Develop a database and directory structure consistent with your programming goals (Carpenter and Smith, 2000 and 2001).
- When writing macros, look for and take advantage of patterns in your code.

**Build a Utility Tool Chest**

As you continue to write macros, start building a macro utility tool chest. Most programmers create programs that perform similar tasks to other programs.

- Turn standard programs into macro tools so that you can avoid redundancy (see Sections 7.5 and 7.6).
- Try to use a macro library to store your utility macros (see Chapter 10).

**Continue to Learn**

Always be on the lookout for new ways to push the boundaries of your SAS knowledge. Very often we get set in our ways. Once we learn how to do a task, we stop asking ourselves if it is the best or even the only solution. Consider the following techniques:

**Semicolon control**

Use %DO and %END to control semicolons (see Section 5.2.2).

**Named parameters**

Named or keyword macro parameters provide additional documentation for your macros and often make the macro easier to use (see Section 4.3.3).

**Program structure**

Minimize the number of program steps through the use of dynamic programming techniques (see Chapter 11).

**Use structured programming techniques**

Avoid the %GOTO and %label statements (see Section 8.2.2).

**Use %LOCAL**

The %LOCAL statement should be used to avoid macro variable collisions. Keep macro variables out of the global symbol table except when necessary (see Section 5.4.2).

**DATA Step Functions**

Learn the Base SAS language DATA step functions. Although not part of the macro language, through the use of %SYSFUNC many of these functions can be used very effectively with the macro language (see Section 7.4.2).

**Learn PROC SQL**

Use PROC SQL to build and maintain macro variables and macro variable lists. Often a single SQL step can accomplish the same thing as a number of other Base SAS language steps (see Section 6.2).

**SEE ALSO:** Levine (1989) discusses issues related to the efficient construction of macro-based systems in terms of standards and maintainability.

Fehd (2000 and 2001) has some great tips on writing good code and things to look for when writing macros. Cheng (1999) includes a number of tips on various aspects of SAS coding, including some tips on macro techniques.

A summary of things to think about before starting to write your macro can be found in Tangedal (2001).

Pochon and Burger (1998) discuss techniques that can be used to validate SAS macros.

Fehd (2014) discusses his thoughts on macro design theory and implementation.

Whitlock (2000a) discusses macro design considerations for a word-wrapping macro. Troxell (2001) discusses the use of complex macros to write code that is easier to validate. Methods of writing macros that are more resistant to user errors are presented by Troxell (2002).

Top-down programming techniques as they apply to macro programming are discussed by Heaton (2001).

Although primarily discussing non-macro issues, Levin (2001) makes a number of very good recommendations regarding SAS programming style.

---

### 14.3.3 Macro Programming Best Practices

There certainly is no one set of best practice guidelines for writing macros. While there is some consensus on some of the topics listed in this section, there is a fairly diverse set of opinions for each.

**Positional versus Named (Keyword) Parameters**

Most programmers prefer to use keyword or named parameters for most macros. The primary exception is for macro functions or macros that mimic functions. The overall feeling is that named parameters should be preferred unless there is a compelling reason to use positional parameters (see Section 4.4).

Some programmers base their decision on how many parameters are needed by the macro. For instance, positional parameters might be used when there are only one or two parameters, but named parameters are used if there are more than two. There is no consensus as to what that number should be.

**Local versus Global Macro Variables**

Macro programmers fall into two fairly evenly divided camps on this topic. Some prefer to use the global symbol table for most macro variables. Others tend to avoid the global table in preference to the use of the local symbol tables.

For those that choose to use the global symbol table, passing values between macros becomes a non-issue as all global macro variables are always available to all macros. And with the advent of the READONLY option on the %GLOBAL statement, it is now possible to protect macro variables in the global symbol table. Although it is possible to protect global macro variables, they cannot subsequently be altered. This type of approach is often best suited to a programming environment where the macro programmer has control over the macros that will be executed and the macro variables that will be created. Macro programmers that depend on macro variables in the global symbol table must *know* the names of those macro variables in order to use them. The resulting macro then becomes dependent on the global symbol table and cannot be used in another programming environment.

Macro programmers that prefer the use of local symbolic variables, tend to minimize the use of the global symbol table, and instead use the most local symbol table to store macro variables. Depending on how these local tables are nested, macros usually have more parameters that pass either the names of macro variables or the values themselves. Sometimes this results in values being concurrently stored in more than one symbol table at the same time; however it does help to control for the unexpected macro variable collision (see Section 5.4.2). Macros that do not depend on the existence of a higher symbol table tend to be more transportable.

### **Macro Variable Usage: With or without the Dot Suffix**

Some macro programmers try to always follow macro variable names with a dot (period) as a suffix. Usually this suffix is optional; however, it is occasionally needed.

Although the majority of the macro programmers tend to only include the dot when it is needed ("It saves a keystroke"), those that include the dot whether it is needed or not tend to feel quite strongly about their approach ("If you always include the dot, you will have it when it is needed"). Of course using a dot as a suffix is required when text is to be appended onto the end of a macro variable's resolved value, and required or not, when a macro variable is resolved, the dot is absorbed.

### **Macro Booleans**

There are no true Booleans in the macro language, so how should true/false and yes/no be indicated? Most macro programmers use 0/1 to indicate false/true rather than using the words. The advantage is that less information is required on the part of the programmer, who does not need to worry about checking for misspellings and miss-capitalizations.

The use of 0/1 values however does not increase the readability of your program in the same way that the words (false/true or no/yes) do. When making this decision, take your audience and the readability of your program into consideration.

### **Nested Macro Definitions**

There is strong agreement among macro programmers that it is just not a good idea to nest macro definitions. Nesting macro calls is another thing altogether. One should never need to nest the %MACRO to %MEND statements within another set of %MACRO to %MEND statements. Nesting definitions can cause a number of problems (see Section 5.1.3).

### **Macro Code Length**

There is no consensus among programmers that this is even a topic that is worthy of discussion, so I will just give you my opinion. I like macros that tend not to be much more than 50 or so lines of code (others have suggested up to 500 lines). My thinking is that macros should be modular, and should perform a single task. If they become longer, they are probably not very modular and should be broken up into more basic units. Consider using a driving macro that calls a series of more modular macros. Modular macros can be written to be more generalized, and therefore more useful in the long run.

### **Macro Libraries**

The preference among a majority of macro programmers seems to be for the use of autocall libraries as the primary storage. Suggestions for improving the autocall libraries include:

- Use different directories for generic macros versus more project-specific macros.
- Concatenate the libraries with the more specific project macro library listed first.
- Use the same hierarchy approach during the development and testing phases of the macro libraries.
- Name your files using lowercase. This makes them more transportable to UNIX and Linux operating systems.
- Each autocall file should ONLY have one macro definition per file.

Stored compiled macro libraries tend to be used for more specialized tasks, but only rarely for efficiency reasons—see Sun and Carpenter (2011).

**SEE ALSO:** Henderson and Carpenter (2012) led a discussion on macro programming best practices at the 2012 SAS Global Forum. The article is posted on sasCommunity.org at [http://www.sascommunity.org/wiki/Macro\\_Programming\\_Best\\_Practices:\\_Styles,\\_Guidelines\\_and\\_Conventions\\_Including\\_the\\_Rationale\\_Behind\\_Them?\\_sm\\_byp=iVVRn5Q2kmN11kIV](http://www.sascommunity.org/wiki/Macro_Programming_Best_Practices:_Styles,_Guidelines_and_Conventions_Including_the_Rationale_Behind_Them?_sm_byp=iVVRn5Q2kmN11kIV).

#### 14.3.4 Debugging Your Macros

As much as I hate to say it, your primary tool for debugging macros will be your own experience. In other words, when you need the most help, you will have the least experience. Later, when you are more experienced, debugging will be easier—of course by then you will be writing more complicated macros with more complicated errors. In the meantime consider some of the following thoughts on debugging your macros.

Debugging help can be found in the manual *SAS 9.4 Macro Language: Reference, Fourth Edition* under the appendix section entitled: “SAS Macro Facility Error and Warning Messages”. This section lists the macro errors you might receive along with possible causes and solutions.

#### Using System Options

The system options MPRINT, SYMBOLGEN, and MLOGIC (see Section 3.3.2) are especially useful during the debugging process. While these options will not make debugging easy, they can assist by providing valuable insights into the code that is being generated by the macros.

If it is not clear what the resultant code that has been generated by your macro looks like, then consider using the MFILE option to write the generated code to a file.

#### Using %PUT

The %PUT statement can be very useful to dump all or part of the various symbol tables to the SAS Log. This statement supports a number of options (see Section 2.4) that make it easy for you to display macro variables, their values, and the symbol tables in which they reside.

#### Addressing Unresolved Macro Calls

The WARNING “Apparent invocation of macro...” indicates that SAS has been unable to find the definition for the macro that has been called. An unresolved macro call results when this definition is not available. Sections 3.3.3 and 3.3.4 discuss how SAS searches for macro definitions and some of the particular issues to consider when you are using macro libraries.

Failure to find macros supplied by SAS such as %LEFT and %TRIM can often be traced to either a failure to include SASAUTOS in the autocall path or to having turned off the autocall facility altogether.

The misspelling of macro names can also contribute to this type of error, so be careful with naming conventions. This can be especially true with autocall libraries established under UNIX or Linux where the file name is case sensitive.

**SEE ALSO:** A number of conference presentations address various aspects of the debugging process. Delaney and Carpenter (2011) specifically covers a number of debugging situations.

Although some of these papers are based on earlier versions of SAS, they still provide good insight: Frankel and Kochanski (1991); Gilmore and Helwig (1990); O'Connor (1991); and Phillips, Walgamotte, and Drummond (1993).

Chapter 10, "Macro Facility Error Messages and Debugging," in *SAS 9.4 Macro Language: Reference, Fourth Edition* (pp. 119–141) both discuss a variety of topics that relate to the debugging and troubleshooting of macros.

O'Connor (1991), one of the primary developers of the macro language at SAS, provides some guidelines for tracking errors and for debugging your macro code.

Although written for Base SAS, Staum (2002) discusses a number of techniques and ideas that can be useful to the macro programmer as well.

Heaton (2001) discusses techniques that can assist with the debugging of macros.

Lafler (2016) has some basic debugging tips.

### 14.3.5 Traps: DATA Step Code versus the Macro Language

While there are a number of similarities between the Base SAS language, especially in the DATA step, and the macro language, there are enough differences that the unwary programmer can be either confused by subtle differences or caught unawares by surprising mistakes. A few of the things that have caused folks consternation are included here. Many of these points of confusion occur when the programmer has not fully assimilated the information contained in Figure 1.4d. A great deal of our understanding of the processes involved with the macro language is tied to the timing of these phases of SAS execution.

The remaining discussions in this section show some of these, sometimes surprising, timing issues and their impacts on the macro programmer.

#### Conditional Macro Statement Execution

In the DATA step, we cannot conditionally execute macros or macro statements (see Section 6.1). The following IF-THEN/ELSE statement attempts to assign a value to &GROUP based on the DATA set variable AGE.

```
if age > 55 then %let group=senior;
else %let group = ;
```

Not only will this code result in a syntax error, but you must always remember that the macro portions of these statements will be executed long before the IF-THEN/ELSE statement. In this case, although the DATA step will fail, &GROUP will always have a null value. After the macro statements have executed, the IF-THEN/ELSE statements become:

```
if age > 55 then
else
```

Of course macro variables can be conditionally specified when DATA step, rather than macro language, tools are used. The following, which uses the DATA step routine CALL SYMPUTX, will work correctly:

```
if age > 55 then call symputx('group', 'senior');
else call symputx('group', '');
```

## IF versus %IF in the DATA Step

It is not uncommon for programmers to attempt to interchange the use of the IF and the %IF statements. When in doubt as to which you should use, remember that the %IF can never evaluate the value of a DATA step variable. Never is a bit strong here, but you would have to go to great lengths to have the %IF evaluate the value of a DATA step variable residing on the Program Data Vector.

Macro statements are used to build code, and the %IF is used to evaluate the values of macro variables. The DATA step IF might use macro variables to build the names of variables or constants that are to be evaluated during DATA step execution, but it is primarily used to evaluate DATA step variables. In the following rather silly DATA step, regardless of the value of the variable AGE, the %IF evaluates to true and every observation from the incoming data set is written to NEW:

```
%macro tryit;
data new;
  set sashelp.class(keep=name age);
  %if age > 55 %then %do; output new; %end;
  run;
%mend tryit;
```

In the %IF statement the comparison is not between the variable AGE and 55, but rather between the letters a-g-e and 55. Remember when the %IF is evaluated, no data has been read and the variable AGE does not yet exist. Since lowercase letters (a-g-e) sort after numbers (in the ASCII collating sequence), the comparison is true. The DATA step code becomes:

```
data new;
  set sashelp.class(keep=name age);
  output new;
run;
```

The %IF is used incorrectly here in an attempt to compare a DATA step variable with a constant. This type of comparison should of course be made using the IF-THEN statement.

The following code executes without a syntax error; however, the IF statement will be executed for every observation in the incoming data set regardless of the value of &DSN. If the name of the requested data set is FEMALE, the conversion formula will be applied.

```
%macro smart(dsn);
data wt;
  set &dsn;
  if "&dsn"='FEMALE' then wt = wt*2.2;
  run;
%mend smart;
```

By using a %IF statement instead of an IF (no comparison is being made on a value in the data set), the assignment statement is only added to the DATA step when &DSN is FEMALE. This is much more efficient. Two semicolons are needed at the end of the %IF statement. The first terminates the %IF and the second completes the assignment statement (see Program 5.2.2c).

```
%macro smart(dsn);
data wt;
  set &dsn;
  %if &dsn=FEMALE %then wt = wt*2.2;;
  run;
%mend smart;
```

In the above two versions of %SMART, you can make the correct choice between %IF and IF by looking at what is being compared. In this case it is a macro variable—not a data set value; therefore a %IF is used.

## Mixing Functions

Some programmers have a tendency to mix DATA step and macro functions when in the DATA step. There are times when this is necessary; however, because the macro references are resolved first, the necessity is rare.

The following IF statement contains a number of embedded function calls. The author used macro functions (%TRIM and %LEFT) whenever the argument was a macro variable. In fact this was not necessary because the macro variable will be resolved to text prior to the application of the macro functions.

```
if upcase(reverse(substr(left(reverse(filename))
,1,length(%trim(%left("&TYP")))))=%trim(%left("&TYP"));
```

Also, it is possible that this usage will yield an incorrect result. Since the macro functions are outside of the quotes in the above example, they effectively do nothing. This is because the text "&typ" is always left-justified before applying the %LEFT function, regardless of the value of &TYP. This is because as far as the macro functions are concerned, the quote marks are just characters. This means that the quotation mark on the left is part of the text and it is already the leftmost character. The following could give a different result:

```
"%trim(%left(&TYP))"
```

There are times, however, when the macro function can be put to good use within non-macro code. In the following PROC PRINT (Stroupe, 2003) the DATA step UPCASE function could not be used.

```
title "Lowest Priced Hotels in the %upcase(&dsn) Data Set";
proc print data=&dsn;
run;
```

## Comparisons with Missing or Null Values

Unlike the DATA step, the macro language does not support the concept of a missing value (see Sections 5.2.1 and 7.1.2). This means that there is no automatic placeholder when making comparisons. In the DATA step, we might check to see whether the variable AGE is missing, with the following:

```
if age = . then do;
```

Of course if we try something similar in a macro %IF, we will be less successful. In this case we want to see whether the macro variable &AGE has a value:

```
%if &age = . %then %do;
```

The comparison will be true only when &AGE holds the value of a period (.), and this is not the same as a null value. Since, unlike DATA step variables, macro variables can take on a null value, the %IF comparison might be written like the following:

```
%if &age = %then %do;
```

Many programmers familiar with comparisons in the DATA step are uncomfortable without a value on the right side of the comparison operator. A common and fairly intuitive solution, but one that indicates a lack of understanding of how the macro language is working, is to use quotation marks to "mark" the right side of the comparison operator (see Sections 5.2.1 and Q5.9).

```
%if "&age" = "" %then %do;
```

Since in the macro language the quotation marks actually become part of what is compared, the first character on both sides of the equal sign is ". A solution that stays within the macro language, but still satisfies the desire to place something on the right side of the comparison operator, is to use a quoting function with no spaces between the parentheses (Section 7.1.2).

```
%if &age = %bquote() %then %do;
```

**MORE INFORMATION:** Additional examples on the detection of macro language null values can be found in Section 9.5.

**SEE ALSO:** Carpenter (2014c) and Chung and King (2009) explore a variety of ways to compare macro variables to null values.

## Using Quote Marks in the Macro Language

The third argument to the %SCAN function is used as the optional word delimiter. When used, it contains one or more characters that will be used to separate one word from the next. In the DATA step this argument is often specified as a quoted string.

```
wrd2 = scan(string,2,'');
```

In the macro language quotation marks are, of course, used quite differently; however, sometimes DATA step programming habits *accidentally* find their way into our macros. This has happened in the following %LET statement:

```
%let wrd2 = %qscan(&string,2,'');
```

The user expects the word delimiter to be a blank space. In fact three delimiters have been specified (a quote mark, a blank, and another quote mark). Using these delimiters, the following value of &STRING

```
AA B'C D'C
```

will have five words, not three:

```
AA B C D C
```

This could, of course, have unintended consequences. The appropriate code would use a quoting function to preserve the space:

```
%let wrd2 = %qscan(&string,2,%str( ));
```

Remember to be careful not to mix DATA step programming knowledge with the macro language.

## Using Implied Arithmetic Operations

In the DATA step, variables are known to be numeric or character and all numeric operations are applied to numeric variables and constants. In the macro language the distinction between numeric and character is not as clear-cut. Section 7.3.1 showed that the arithmetic in the following %LET statement will not be performed:

```
%let x = &y + 1;
```

However, this arithmetic operation is automatically done in the DATA step. The code fragment shown below also contains an arithmetic operation (addition) both in the second argument of the SCAN function and in a SUM function. In both cases the addition operation is performed as anticipated.

```
i=0;
do while((scan(xcode, i+1.0))>' ');
  i+1;
```

The second argument in the SCAN function adds 1 to the current value of the variable I as does the SUM statement. In the macro language this might be rewritten as:

#### **Program 14.3.5 (Partial): Arithmetic Operation with an Implied %EVAL**

```
%let i=0;
%do %while((%scan(&rcode, &i+1))>%bquote());
    %let i = %eval(&i+1);
```

It turns out that this %SCAN function will work correctly because there is an implied %EVAL function (Section 7.3.2) for the second argument (%SCAN knows that the second argument has got to be treated as a number, and because it detects an arithmetic operator, the plus sign, a %EVAL is used to perform the addition). It is as if the %DO %WHILE had been coded as:

```
%let i=0;
%do %while((%scan(&rcode, %eval(&i+1)))>%bquote());
    %let i = %eval(&i+1);
```

Because the %EVAL function (implied or explicit) only works with integers, you would be unable to code the second argument as:

```
%do %while((%scan(&rcode, &i+1.0))>%bquote());
```

Remember the macro language interprets any number with a decimal point (1.0 or even 1.) as non-integer.

#### **Creating and Using a Macro Variable in the Same Step**

Generally, when SAS programmers create and use a macro variable in the same DATA step, they are not taking advantage of the power of the DATA step. If you think about it, we are asking to take a DATA step value, write it to the symbol table, and then later retrieve it from the symbol table all in the same step. Why not use the Program Data Vector, which can do this exact same thing using the RETAIN statement without stepping out of the DATA step? Program 6.5.3c discusses a fairly practical example of using the macro variable in the same step that creates it.

That said, let's look at the problem that folks often have when trying to do this sort of thing. In the following DATA step, the first observation's value of the variable AGE is written to the symbol table using the SYMPUTX routine:

```
data new;
  set old;
  by age;
  if _n_=1 then call symputx('firstage',age);
run;
```

The youngest age for this patient will be stored in &FIRSTAGE. If all we wanted to do is save youngest or first age for use later within the DATA step, we could have written the step as:

```
data new;
  set old;
  by age;
  retain firstage .;
  if _n_=1 then firstage = age;
run;
```

For the sake of discussion, let's assume that the youngest age is to be used to subset the data. The DATA-step-only solution might be:

```
data ageerrors;
  set old;
  by age;
  retain firstage .;
```

```
if _n_=1 then firstage = age;
if age-firstage>10 then output;
run;
```

A common error, when using macro variables to do something similar, will be encountered in the following step:

```
data new;
  set old;
  by age;
  if _n_=1 then call symputx('firstage',age);
  if age-&firststage>10 then output;
run;
```

The macro facility will attempt to resolve &FIRSTAGE before a value has been assigned. Remember that this resolution will occur before *any* data has been read. This will, of course, cause syntax errors. Instead, you could use the SYMGET function to retrieve the macro variable during the execution of the DATA step.

```
data new;
  set old;
  by age;
  if _n_=1 then call symputx('firstage',age);
  if age-symget('firstage')>10 then output;
run;
```

By using the SYMGET function, this DATA step could be modified so that it could be used with more than one patient, without creating a macro variable for each patient.

```
data new;
  set old;
  by name age;
  if first.name then call symputx('firstage',age);
  if age-symget('firstage')>10 then output;
run;
```

Again, for this example the DATA-step-only solution is more practical, but it is important to understand the timing of the process nonetheless.

### 14.3.6 Little Things with a Big Bite

There are some problems that you could encounter in your macro programming career that you would not wish on your worst enemy, well maybe on your worst enemy, but no one else. These are the things that generate many hours if not days of frustration. They might not seem like much, but when they crop up, bad things can happen.

#### Macro Variables in the Wrong Symbol Table

There are a number of ways of creating macro variables, and how and when they are created will determine whether the macro variable will be written to a LOCAL or GLOBAL symbol table. The topic is complex enough to be discussed in more detail in Section 14.6.

An indication of a typical problem that occurs when variables go astray is the “undefined macro variable reference” error message in the SAS Log. When you get this message and you know that the variable should be defined, the variable might have been placed in the wrong symbol table.

Assuming that the macro variable really does exist, you might want to look for it in a symbol table that was not available when the message was generated. When you are sure that the variable has indeed been

defined, consider using a %PUT statement to identify the macro variables in the various symbol tables. The code in Program 14.3.6a creates the following:

- a global macro variable (&A)
- a macro variable in the symbol table for %TRYIT (&AT)
- a macro variable in the symbol table for %INSIDE (&B)

#### Program 14.3.6a: Showing Macro Variable Scopes

```
%let a = global_var; ①

%macro tryit;
  %let at = var_local_to_tryit; ②
  %inside
%mend tryit;

%macro inside;
  %let b=var_on_the_inside_table; ③
  %put **All Current Vars;
  %put _user_; ④
%mend inside;

%tryit
%put **** Open code Macro vars;
%put _user_; ④
```

- ① &A is placed on the global symbol table
- ② &AT is written to the local symbol table for %TRYIT
- ③ &B is written to the local symbol table for %INSIDE
- ④ For each macro variable the \_user\_ option on the %PUT statement writes to the SAS Log each of the following:
  - name of the symbol table
  - name of the macro variable
  - current value of the macro variable

After executing %TRYIT, the SAS Log for Program 14.3.6a clearly shows that the macro variables have been written to the correct tables.

```
33  %tryit
**All Current Vars
INSIDE B var_on_the_inside_table ②
TRYIT AT var_local_to_tryit ③
GLOBAL A global_var ①
34  %put **** Open code Macro vars;
**** Open code Macro vars
35  %put _user_;
GLOBAL A global_var ①
```

Macro variables can be written to the wrong table when:

- The programmer does not understand the rules that are used to place macro variables into symbol tables (see Section 14.6).
- The programmer does not notice a %GLOBAL or %LOCAL statement that names the offending macro variable.
- There is a macro variable collision (see next topic).

In Program 14.3.6b the macro variable &ININSIDE is created in the inner macro (%INSIDE) using a SYMPUTX routine. Because no symbol table has otherwise been created for %INSIDE, the macro variable &ININSIDE is written to the next higher table.

#### Program 14.3.6b: Writing to the Wrong Symbol Table

```
%macro tryit;
  %let at = var_local_to_tryit; ❸
  %inside
%mend tryit;

%macro inside;
data _null_;
  call symputx('ininside','inside'); ❹
run;
%put **All Current Vars;
%put _user_; ❺
%mend inside;

tryit
```

- ❸ The macro variable &AT is created and written to the local table for %TRYIT.
- ❹ A local table does not yet exist for %INSIDE, and the SYMPUTX routine will not create a local table; consequently, the macro variable &ININSIDE is written to the next higher table (%TRYIT). Using a %LET statement, as was done in Program 14.3.6a, is sufficient to create the symbol table for %INSIDE.
- ❺ The %PUT statement shows that both macro variables are in the %TRYIT symbol table.

```
**All Current Vars
TRYIT AT var_local_to_tryit
TRYIT ININSIDE inside
```

Although we might think that the macro variable &ININSIDE has been written to the wrong symbol table, in actuality the macro language has followed its own rules for determining where a macro variable should be written (see Section 14.6 for more on these assignment rules).

#### Macro Variable Collisions

In Program 14.3.6b a macro variable assignment was not made to the symbol table that we anticipated; however, since that macro variable did not already exist in *any* symbol table, there was no collision. A macro variable collision occurs when a value of one macro variable interferes with (replaces or changes) the value of a macro variable with the same name in a different (higher) symbol table.

A simple example can be used to illustrate the problem. In the following code the programmer creates a global macro variable (&DSN) to hold the name of the data set of interest. After his program was working his boss asked him to add a call to a macro (%DATSERNUM), which was independently developed and tested. Now whenever he uses the macro %DATSERNUM, the DATA step that creates NEW terminates with syntax errors. Although he does not yet know it, he has experienced a macro variable collision. Here is the portion of his program that produces the syntax error.

#### Program 14.3.6c (Partial): Demonstrating a Macro Variable Collision

```
/* Define the data set of interest;
%let dsn = clinics;

/* Determine the Data Serial Number;
%let sernum = %DatSerNum(adjust=5);

* Create the new data;
data new;
```

```
set &dsn; ❾
/*...code not shown...*/
run;
```

It turns out that the problem really resides in the macro %DATSERNUM, a portion of which is shown here:

#### Program 14.3.6c (Continued): This Instance of &DSN Overwrites the First Instance

```
%macro datsernum(adjust=0);
  %if &adjust= %then %let adjust=0;
  /* Generate the DSN (Dataset Serial Number);
  %let dsn = %eval(5 + &adjust); ❸
  &dsn
%mend datsernum;
```

When the variable &DSN is defined in the macro %DATSERNUM ❸ its value would normally be placed in the local symbol table for %DATSERNUM (which already exists); however, since &DSN already exists in a higher table, the value in that higher table will be replaced. In this case the name of the data set (CLINICS) is replaced by a number. It is this number that causes the syntax error when it is used as a data set name ❹ in the DATA step that creates NEW.

A more subtle example, and one that can cause horrid errors while leaving the programmer blissfully unaware, is contained in Program 14.3.6d. In this case a secondary macro is called from within a %DO loop. The %DO seems to work without error. But a closer inspection of the inner macro %CHKSURVEY reveals a hidden problem.

#### Program 14.3.6d: Collisions in the %DO Loop Index

```
%macro primary;
  /*...code not shown...*/
  %do i = 1 %to &dsncnt;
    %chksurvey(&&dsn&i)
  %end;
  /*...code not shown...*/
%mend primary;
%macro chksurvey(dset);
  %do i = 1 %to 5;
    /* ...code not shown...*/
  %end;
%mend chksurvey;
```

The %DO loop in %CHKSURVEY also uses &I as the index variable. Again, this variable would normally be local to %CHKSURVEY; however, since &I already exists in the higher table of the calling macro, %CHKSURVEY will modify the value of &I in that higher table. In the case shown above, &I will have a value of 6 after %CHKSURVEY has been executed. When &DSNCNT is less than 7, the %DO loop in %PRIMARY will terminate **after having executed only once!** If &DSNCNT is greater than 6, an infinite loop will have been established, since the &I in %PRIMARY will be reset to 6 over and over again. At least the programmer is likely to discover the latter problem.

The above macro variable collisions can be completely eliminated simply by using the %LOCAL statement to identify all macro variables that you intend to be local to the inner macro. %CHKSURVEY then becomes:

```
%macro chksurvey(dset);
  %local i;
  %do i = 1 %to 5;
    /* ...code not shown...*/
  %end;
%mend chksurvey;
```

**SEE ALSO:** Carpenter (2005) discusses macro variable collisions in more detail.

### Asterisk-Style Comments in Macros

There are three primary styles of comment statements that are available to SAS. Although all three can be used either in open code or within macros, they do not behave the same nor are they necessarily interchangeable. They behave differently in part because of when and how they are removed from the code (see Section 5.4.1 for more on the macro comment).

It is not unusual to write code that uses the asterisk-style comment to comment out macro statements. In the following open code statements, the %LET statement associated with the data type of interest is left uncommented.

```
*%let dattype = ae;
%let dattype = demog;
*%let dattype = meds;
**;

%put data type is &dattype;
```

In this case all three styles of comments will work equally well, and as we would expect, the %PUT writes to the SAS Log that the “data type is demog”.

When this same code is placed inside of a macro, however, the result is not the same. The SAS Log shows that &DATTTYPE has been set to ‘meds’ even though that %LET statement has been commented out.

```
55  %macro tryit;
56    *%let dattype = ae;
57    %let dattype = demog;
58    *%let dattype = meds;
59    **;
60
61    %put data type is &dattype;
62  %mend tryit;
63  %tryit
data type is meds
```

To understand the problem, we need to keep in mind how the code is being parsed. The parser breaks down the code into elemental units known as tokens. SAS statements start with an identifying set of characters known as a keyword. This keyword lets SAS know how to interpret the remaining characters through the end of the statement (semicolon).

In open code the asterisk is seen as keyword and all characters between the asterisk and the semicolon are commented out. This is even true for the %LET within the comment. Thus, in open code it is possible to comment out macro statements and macro calls using asterisk-style comments.

Inside a macro things are different. Once the %MACRO statement is encountered, all the code between the %MACRO and %MEND is passed to the macro facility. Here, when the code is parsed, SAS looks for macro statements and generally ignores most of the things that otherwise have meaning in the Base SAS language (for example, asterisks are not seen as tokens and are therefore not interpreted as the keyword that starts a comment). This means that for the macro %TRYIT shown here, each of the macro statements (bolded) will be executed.

```
%macro tryit;
*%let dattype = ae;
%let dattype = demog;
*%let dattype = meds;
**;

%put data type is &dattype;
%mend tryit;
```

All three %LET statements are executed, the last of which assigns the value of MEDS to &DATATYPE. Remember that, although macro statements such as %LET cannot be commented with an asterisk, a macro call is not seen as a macro statement and can still be commented in this manner (see example below). After the macro statements have been executed, the asterisks remain. Essentially, the code that is passed out of %TRYIT becomes the four remaining asterisks.

```
*  
*  
**;
```

As a cautionary comment on what just happened, notice that the string of asterisks end up forming a comment that starts with the very first asterisk. If this string of asterisks had not been in the macro, the first two asterisks would have been left behind without a semicolon. This could easily cause a problem by inadvertently creating a comment where it was not wanted. This is also a clue to answering the questions posed in the following example.

**QUIZLET:** In order to determine your level of understanding of the use of comments within a macro, consider the macro %ABC in Program 14.3.6e, which contains a single %PUT statement that the programmer has attempted to comment. This macro is called twice in the macro %DOIT. Both times an attempt has been made to comment out the macro call and the %LET statement that contains the second macro call.

- Will the %PUT ① be executed if %ABC is called?
- When %DOIT is executed, how many times will %ABC ② be called?
- During the execution of %DOIT, will the %LET be executed and if so, what value will be assigned to &X? ③
- What, if anything, is the resolved (non-macro) code returned by %DOIT? ④

#### Program 14.3.6e: Understanding Asterisk Comments in a Macro

```
%macro abc;  
    *%put in abc; ①  
%mend abc;  
  
%macro doit;  
    *%abc ②  
    %put here;  
    *%let x = %abc; ②③  
        %put value of x is &x;  
%mend doit;  
%doit * run doit; ④
```

The SAS Log shows the following:

```
35  %doit * run doit;  
here  
in abc ②  
value of x is * ③
```

- ① When %ABC is called the asterisk will *not* form a comment and the %PUT will be executed. This leaves the \* behind as the resolved text from the macro %ABC.
- ② The first call to %ABC is commented; however, neither the %PUT nor the %LET is commented. Before a value can be assigned to &X, macro %ABC is executed, and this in turn executes the %PUT that is in %ABC.
- ③ The only non-macro text in %ABC (the asterisk) is passed back as the value for &X.
- ④ The non-macro text in %DOIT will be \*%abc\*.

**EXTRA CREDIT:** What would the SAS Log show if you called the macro %DOIT with the following %PUT statement? (See Appendix 1 for the answer.)

```
%put %doit;
```

This example highlights the necessity to use comments correctly within a macro. SAS Institute recommends the use of PL1 style comments within a macro.

**SEE ALSO:** SAS Usage Note 32684 describes the use of the different styles of comments within a macro.

## Macro Syntax

Debugging macros can sometimes be problematic. The error messages, even when using MPRINT, MLOGIC, and SYMBOLGEN, can occasionally be a bit cryptic. The problem can be exacerbated when your code is missing pieces of the macro language.

A simple but non-trivial example occurs when the programmer forgets to use the %MEND statement to close the macro definition. When programming interactively, this is an especially irritating problem as the system seems to freeze (it is actually waiting patiently for the %MEND). Resubmitting the program only makes things worse as SAS is then waiting for two %MEND statements.

Often you can clear these pending macro definitions by submitting a series of %MEND statements. You might need to submit several. You will know that you need no additional %MEND statements when the following error message is displayed in the SAS Log:

```
ERROR: No matching %MACRO statement for this %MEND statement.
```

Related problems occur when the % sign is left off of the %MEND or when the name of the macro does not match the name on the %MEND statement.

The error in the following example also wipes out the %MEND statement, but by a very different mechanism—a missing parenthesis. Parentheses of course come in pairs and when macro functions, especially embedded macro functions, do not have the correct pairs of parentheses, the error messages point in every direction except to the parentheses. The following macro function counts the number of words in a list using the %SCAN and %STR functions.

### Program 14.3.6f: Counting Words in a List of Words

```
%macro wrdcnt(string);
%let cnt=0;
%do %while(%scan(&string,&cnt+1,%str( )) ne %str());
    %let cnt = %eval(&cnt+1);
%end;
&cnt
%mend wrdcnt;
```

If we “accidentally” leave out one of the closing parentheses after the first use of %STR ①, the code becomes:

```
%macro wrdcnt(string);
%let cnt=0;
%do %while(%scan(&string,&cnt+1,%str( )① ne %str());
    %let cnt = %eval(&cnt+1);
%end;
&cnt
%mend wrdcnt;
```

When this version of %WRDCNT is executed, the SAS Log shows the following:

```

124  %macro wrdcnt(string);
125  %let cnt=0;
126  %do %while(%scan(&string,&cnt+1,%str( ) ne %str()));
127    %let cnt = %eval(&cnt+1);
ERROR: Macro keyword LET appears as text. A semicolon or other delimiter
may be missing.
128  %end;
ERROR: Macro keyword END appears as text. A semicolon or other delimiter
may be missing.
129  &cnt
130  %mend wrdcnt;
ERROR: Macro keyword MEND appears as text. A semicolon or other delimiter
may be missing.
131
132  %put the count is %wrdcnt(a b c d);
ERROR: Macro keyword PUT appears as text. A semicolon or other delimiter
may be missing.
ERROR: Expected semicolon not found after WHILE clause. A dummy macro will
be be compiled.
WARNING: Missing %MEND statement.

```

As you can see, just about every statement is flagged except the one with the missing parentheses. Especially note that since the %MEND was also missing, an interactive session will be waiting for a %MEND statement. When you are using nested functions and you get odd messages, in addition to missing semicolons, also look for mismatched parentheses.

### Non-integer Comparisons

When making numeric comparisons one of the first things to remember is that by default the macro language will only perform a numeric comparison for integer values (for example, 9 < 10). The presence of decimal values or even of the decimal point will invoke the use of a character or alphabetical comparison (for example, 10. < 9.). More details on these comparisons, the defaults, and how to override them are discussed in Section 7.3.

It of course remains your responsibility to ensure that the type of comparison is what you expect it to be. If you anticipate that the values are to be integers, the use of the implicit %EVAL is sufficient. But if you do make that assumption, a comparison of non-integer values could then result in incorrect conclusions some of the time.

### Numeric Range Comparisons

The macro %CHECKIT performs a simple range check on the value passed into the macro. The intent is to identify all values that are either in the range of -5 to 0 or in the range of 1 to 5 (the value of &VAL is assumed to be an integer).

```

%macro checkit(val);
  %if -5 le &val le 0 %then %put &val is in neg range (-5 to 0);
  %if 1 le &val le 5 %then %put &val is in pos range (1 to 5);
%mend checkit;

```

The SAS Log shows the following:

```

150  %checkit(-10)
-10 is in neg range (-5 to 0)
-10 is in pos range (1 to 5)
151  %checkit(-2)
-2 is in pos range (1 to 5)
152  %checkit(2)
2 is in pos range (1 to 5)
153  %checkit(10)
10 is in pos range (1 to 5)

```

Virtually every determination that this macro makes is incorrect! What has happened? In the DATA step this type of range check is very common and the following IF will work correctly:

```
if -5 le val le 0 then do;
```

The DATA step interprets the range comparison as if it had been coded as:

```
if -5 le val and val le 0 then do;
```

The macro language, however, does not interpret the expression in the same way. The first %IF in %CHECKIT:

```
%if -5 le &val le 0 %then ...
```

is interpreted as:

```
%if (-5 le &val) le 0 %then ...
```

when &VAL is -10 the %IF becomes:

```
%if (-5 le -10) le 0 %then ...
```

since the comparison that checks to see whether -5 was less than -10 is false, a zero is returned and the expression becomes:

```
%if 0 le 0 %then ...
```

and this is of course true. This means that when you want to do a range check in the macro language, you must use a compound expression. In order for it to be correctly evaluated, the previous %IF becomes:

```
%if -5 le &val and &val le 0 %then ...
```

## 14.4 Understanding Recursion in the Macro Language

Recursion occurs when an element of the macro language calls or references itself. Depending on the circumstances, this could be useful or it could cause errors.

Usually you will encounter recursion in the form of an error—you didn't do it on purpose. Often this is the result of a syntax error and the most likely culprit will be a missing semicolon. In the following statements, the user has attempted to use the %LET to assign a value to &DSN. This value is then to be displayed using a %PUT.

```
%let dsn = clinics  
%put data set is &dsn;
```

Since the semicolon is missing from the %LET statement, the macro processor will see the %PUT as part of the value to be assigned to &DSN, but the %PUT must be a keyword, not part of a statement, and the SAS Log will show:

```
211 %let dsn = clinics  
212 %put data set is &dsn;  
ERROR: Open code statement recursion detected.
```

This error is NOT being caused because the &DSN appears on both sides of the equal sign, but because the %PUT is out of place. There is nothing wrong with statements like:

```
%let dsn = &dsn clinics;
```

In fact this is a simple example of an acceptable recursion. Other examples include macros that call themselves. The macro %FACT in Program 14.4 calculates and returns a factorial value by calling itself recursively.

The factorial for 4 is defined as  $4*3*2*1$ , which is equal to 24. The statement:

```
put "The factorial of 4 is %fact(4);
```

becomes:

```
put "The factorial of 4 is 24";
```

Although the macro itself is fairly simple, there are other ways to calculate a factorial (see Program 7.6.2a or the DATA step's FACT function). This macro can, however, be used to help explain what happens when a macro calls itself.

#### Program 14.4: Using Recursion to Have a Macro Call Itself

```
%macro fact(n);①
  %if &n > 1 %then %eval(&n * %fact(%eval(&n-1))); ②
  %else 1; ③
%mend fact;
```

- ① The parameter (&N) is the number whose factorial is to be calculated.
- ② The %EVAL function is used to multiply the current value to the factorial of one less than the current number. This macro takes advantage of the fact that  $4!=4*3!$ , which in turn is  $4! = 4*3!=4*3*2!$ , and so on. The “and so on” is the recursion.
- ③ Always be very careful when a macro calls itself. Unless you have an escape set up, you can easily generate an infinite loop.

For %FACT(4) the %EVAL ② becomes:

```
%eval(4 * %fact(3))
```

which becomes:

```
%eval(4 * %eval(3 * %fact(2)))
```

which becomes:

```
%eval(4 * %eval(3 * %eval(2 * %fact(1))))
```

which becomes:

```
%eval(4 * %eval(3 * %eval(2 * 1)))
```

which becomes:

```
%eval(4 * %eval(3 * 2))
```

which becomes:

```
%eval(4 * 6)
```

which becomes:

```
24
```

Now that we have established that a macro *can* call itself, we should also ask **should** a macro call itself? The nesting of macros can cause problems as the number of nested calls becomes large. “Large” of course varies by a number of factors including the operating system and the version of SAS. Just be careful.

**SEE ALSO:** Ward (2001) discusses the use of recursion within the macro language as well as within SCL. A type of recursion is demonstrated by Rhoades (2001) when he re-enters and re-reads data multiple times. Adams (2003) discusses and uses recursion in a macro that builds a list of files including those in subdirectories. Benjamin (1999) demonstrates a macro that simulates recursion through the use of arrays.

## 14.5 Determining Macro Variable Scopes

The primary factor in determining the scope of a macro variable is whether it is being defined in open code or from within a macro. Secondarily, the decision tree varies according to the method by which the variable is created.

Macro variables created in open code, regardless of the method of creation, will be stored in the global symbol table. This should be fairly obvious since in open code there are no local symbol tables, or scopes, defined. When macro variables are defined from within a macro, they will generally be placed on the symbol table local to that macro. There are exceptions, however, and these exceptions can be somewhat arcane. The decision trees discussed in the sections below assume that the macro variable is being defined within a macro.

The current scope of all macro variables can be determined by using the \_USER\_ keyword on the %PUT statement and it is also stored in the view SASHELP.VMACRO and the DICTIONARY.MACROS table.

You will sometimes hear the term “empty symbol table”. There has been some discussion regarding this topic. It is likely that there are no empty symbol tables and instead the table itself is not created until the first macro variable is written to it.

**MORE INFORMATION:** Section 1.4 briefly introduces the concepts of referencing environments, and Section 5.4.2 shows, through the use of the %LOCAL and %GLOBAL statements, how to override and control the default behaviors discussed in the sections below.

**SEE ALSO:** Chapter 5 in the *SAS 9.4 Macro Language: Reference, Fourth Edition* does a good job of discussing the decision trees described in these sections. Similar decision trees can also be found in Burlew (2014, Chapter 5). In a blog Russ Tyndall, a SAS macro language developer, gives a very nice summary of the process to determine macro variable scopes (see <http://blogs.sas.com/content/sgf/2015/02/13/sas-macro-variables-how-to-determine-scope/>).

### 14.5.1 Nested or Layered Symbol Tables

It is not at all unusual to have macros that call macros, and since there is likely to be a symbol table associated with most if not all of these macros, the symbol tables are essentially nested or layered. The symbol table associated with the calling macro is said to be the “next higher” symbol table or scope.

When a macro variable is created it must be assigned to a symbol table, and when nested symbol tables exist the process of deciding which symbol table to use often includes a check of any higher symbol tables. SAS automatically keeps track of the number of levels and of the macro variables associated with each symbol table. You can determine which variables are in which symbol tables by browsing SASHELP.VMACRO or DICTIONARY.MACROS.

### 14.5.2 Macro Parameters

Macro variables (positional or keyword) created through the use of macro statement parameters will always have a scope that is local to that macro. This will be true even if the macro variable already exists in a higher symbol table, including the global symbol table.

Other than the decision rule for determining the scope of macro variables created in open code, this is probably the easiest of the assignment rules to understand and remember.

### 14.5.3 Macro Variables Created with %LET and %DO

Macro variables created with either the %LET or %DO statements follow the same rules. These rules are applied in the following order:

1. For each assignment of a macro variable value, the local table is checked first to see whether the variable already exists on the local table. If it does exist, the assignment is made to the local table.
2. If the variable is not already on the local table, or the local table does not already exist, each higher table is checked successively up to and including the global table. If the variable is found to exist on any of these tables, the assignment is made to the first table (the lowest) that already contains the variable.
3. If the variable is not found on any higher table, the assignment is made to the local table.

### 14.5.4 Macro Variables Created with the SYMPUT and SYMPUTX Routines

The default rules for the placement of macro variables created by the CALL SYMPUT and CALL SYMPUTX DATA step routines are the same. However, the optional third argument for SYMPUTX allows the programmer to force the macro variable into either the global symbol table or into the current local symbol table. The default (the third argument is 'f') follows the rules described below.

The assignment rules for these routines are not as easily anticipated as are those for the %LET statement. In part this is because these routines are not macro language elements. Usually they write the macro variable to the local symbol table (whether it already exists on the table or not) and does not check higher tables (as the %LET statement does).

A special case rule comes into play if the local table has not yet been created (no macro variables have been assigned to the local table). When this happens, both SYMPUT and SYMPUTX will write to the first higher table that is not empty. Since this rule is a rare exception to the usual case, it is probably only fitting that it too should have an even rarer exception.

This last, rarely invoked, rule is that these routines will write to the local table, or create it if necessary, when either of two conditions apply:

1. The macro containing the DATA step creates &SYSPBUFF when the macro is called.  
(Technically &SYSPBUFF will already be on the local table, so the local table must therefore already exist. This exception really isn't an exception as much as a reminder.)
2. The macro contains a computed %GOTO statement.

In the following example the CALL SYMPUTX seems to violate these rules. The DATA step is used to create a macro variable CNTMALES, which contains the number of male patients.

#### Program 14.5.4: Symbol Table Assignment Using SYMPUTX

```
%macro cntmales;
  %local cntmales; ①
  data malesonly;
    set macro3.clinics end=eof;
    if sex='M' then do;
      cnt+1;
      output malesonly;
    end;
    if eof then call symputx('cntmales',cnt);
  %mend cntmales;

%cntmales
title "Number of Males is &cntmales"; ②
proc print data=malesonly(obs=5);
  var lname fname sex;
  run;
```

Since &CNTMALES is declared as local ❶, it should come as no surprise that the use of &CNTMALES in the TITLE statement ❷, which is outside of the macro, produces an uninitialized macro variable message.

However, when the TITLE statement is moved inside the PROC step ❸, the macro variable &CNTMALES is no longer uninitialized and therefore seems not to be local to the macro %CNTMALES.

```
%cntmales
proc print data=malesonly(obs=5);
  var lname fname sex;
  title "Number of Males is &cntmales"; ❸
  run;
```

What has happened? Moving the TITLE statement won't make a difference as to which symbol table contains &CNTMALES. In fact, the local version of &CNTMALES was never assigned a value. The key here is determining the step boundaries. Since the DATA step does not have a RUN statement (I failed to point this out), the DATA step is not terminated, and therefore not executed, until the PROC statement is encountered. In this case the DATA step is not executed until after %CNTMALES has completed. Effectively, the DATA step is in open code and the SYMPUTX necessarily writes to the GLOBAL symbol table. The symbol table for %CNTMALES no longer even exists.

Placing the TITLE statement before the PROC statement ❷ causes the uninitialized macro variable error, because as a global statement (TITLE statements are not seen as step boundaries), it does not trigger the execution of the DATA step. As a global statement the TITLE statement is executed when it is encountered, and as it is before the step boundary, the TITLE statement is executed before the DATA step. Like a RUN statement, the PROC statement is a step boundary, and does signify that the DATA step has been fully specified, and the DATA step is executed when the PROC statement is encountered.

With the TITLE statement inside the PROC step ❸, the DATA step must have been executed and the macro variable &CNTMALES made available by the time the TITLE statement is encountered.

#### 14.5.5 Macro Variables Created in a PROC SQL Step Using the INTO: Operator

Like the SYMPUT and SYMPUTX routines, the INTO: operator in an SQL step is not a macro language element. However, its rules for the assignment of macro variable values are most similar to those of the %LET statement. In part this is due to the fact that, unlike DATA steps that contain a CALL SYMPUT, the local table will always exist and will never be empty. This is because the SQL step itself will automatically generate a minimum set of local macro variables (&SQLOBS, &SQLOOPS, &SQLXOBS, and &SQLRC).

---

### 14.6 Controlling System Initialization and Termination

Whenever SAS is started, you can direct it to automatically execute a SAS program that you can design to set up your environment. Although you can call this initialization program anything that you want, by default the name is AUTOEXEC.SAS. The file can be executed through the use of the AUTOEXEC initialization option. The use of the AUTOEXEC.SAS is especially useful to macro programmers as it gives you a way to automatically set up a customized environment. This can include global macro variables and macro libraries.

Sometimes we want to initialize our global symbol table to values saved from a previous SAS session, and this can be done by transferring the information to a SAS data set.

You can also cause specific statements to be executed at the start and end of your SAS session through the use of the -INITSTMT and -TERMSTMT system options.

**MORE INFORMATION:** The purpose and utility of the AUTOEXEC.SAS program is described in Section 11.9.3. The automatic macro variable &SYSPARM is used in the initialization of a SAS session in Section 8.3.1.

**SEE ALSO:** Consult the SAS Companion for your operating system for specific details on the default locations and naming conventions for the AUTOEXEC.SAS program. Wang (2003) has an example that customizes a Windows shortcut using the AUTOEXEC. Eberhardt (2016) discusses initialization options associated with the use of the autoexec and configuration files.

### 14.6.1 Controlling AUTOEXEC Execution

When SAS is first initialized, options can be applied that will direct SAS to execute a specific AUTOEXEC file. This means that the file can take on any name, and it does not need to be AUTOEXEC.SAS.

Initialization options are preceded by a dash when specified, and the -AUTOEXEC initialization option specifies the file to be executed.

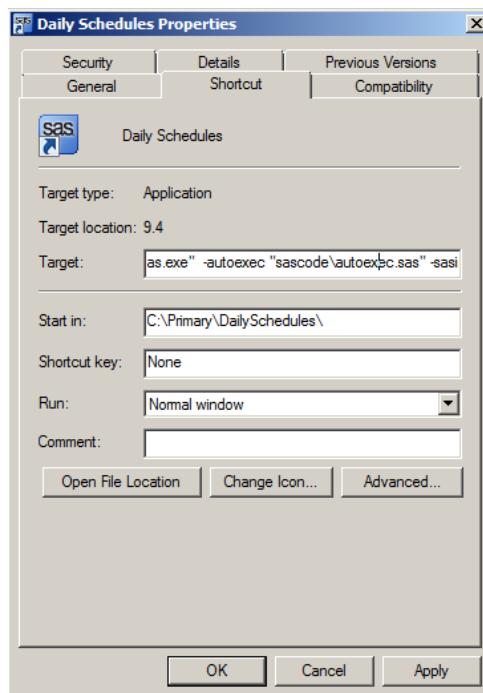
Under MVS and z/OS this option can be specified on the //SYSIN line.

When executing SAS from batch, the UNIX option might look something like this:

```
sas -autoexec /home/justina/autoexec.sas
```

Under Windows it is specified by editing the properties of the SAS shortcut, as shown in Figure 14.6.1.

**Figure 14.6.1: Windows Shortcut Using the -AUTOEXEC Initialization Option**



**SEE ALSO:** Carpenter (2012) Section 14.1.1 discusses initialization options in more detail, and Section 11.9.3 discusses the use of the AUTOEXEC program.

Fehd (2001) has three examples of AUTOEXEC programs and discusses their differences. Jennifer Price (1998) gives some general guidelines on setting up an AUTOEXEC.

## 14.6.2 Saving the Global Symbol Table

The global symbol table is by default cleared at the end of the SAS session. There are times however when you might wish to preserve all or some of the macro variables from one session and use their values to reinitialize them at the start of a subsequent SAS session.

In Program 14.6.2 a DATA step in the macro %GLOBALSAVE is used to write the contents of selected portions of the view SASHELP.VMACRO to a data set. The macro %GLOBALRETRIEVE can then be used in a subsequent SAS session to restore these same macro variables.

### Program 14.6.2: Saving and Recovering Global Macro Variable Values across SAS Sessions

```
%macro GlobalSave(scope=global,macdsn=);
/* Save global macro variables
 * scope    global      user defined global macro variables (default)
 *          automatic   system generated automatic macro variables
 * macdsn   Data set to contain the macro variable values
 * *****;
data &macdsn; ①
  set sashelp.vmacro(where=(scope="%upcase(&scope)")); ②
  run;
%mend globalsave;
%globalsave(scope=global,macdsn=macro3.GlobalMacroVars) ③

%macro GlobalRetrieve(scope=global, macdsn=);
data _null_;
  set &macdsn(where=(scope="%upcase(&scope)")); ④
  call symputx(name,value,'g'); ⑤
  run;
%mend globalretrieve;
%globalretrieve(scope=global, macdsn=macro3.globalmacrovars) ⑥
```

- ① The macro variables will be saved to a SAS data set named when the macro is called. This, of course, should be a permanent data set, but this pair of macros can also be used to save macro variable values that might become altered during the course of the execution of an application. This data set could have also been created with an SQL step using the DICTIONARY.MACROS; however, the automatic macro variables created by the SQL step will not yet have values that they will have after the step is completed.
- ② Generally only user-defined global macro variables will be saved, but the user could also save automatic macro variables instead.
- ③ For this call to the %GLOBALSAVE macro, the values of the currently defined global macro variables will be saved in the MACRO3.GLOBALMACROVARS data set.
- ④ The %GLOBALRETRIEVE macro will restore the macro variables stored in the data set named in &MACDSN macro variable with the scope of GLOBAL.
- ⑤ The SYMPUTX routine is used to place the macro variables named by NAME with the value stored in the data set variable VALUE. The third argument forces these macro variables into the global symbol table.
- ⑥ This macro call to %GLOBALRETRIEVE restores the macro variables stored in the data set MACRO3.GLOBALMACROVARS.

## 14.6.3 Executing Initialization and Termination Statements

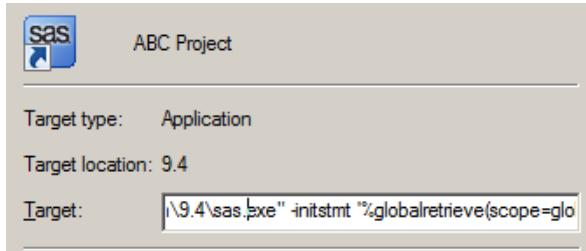
When you want to execute a predefined program at SAS invocation, you can use the -AUTOEXEC initialization option (see Section 14.6.1), or you can use the -INITSTMT option to supply one or more statements directly. These can be macro statements, a %INCLUDE, or even a call to a macro. Whatever is specified on the -INITSTMT option will be executed immediately after the AUTOEXEC.SAS program.

Like other system options that are used at SAS invocation, the -INITSTMT option is specified along with the script that starts the execution of SAS.

```
sas -initstmt '%put *** Initialization Complete ***;'
```

In the Windows environment the -INITSTMT option is used on the target line in the properties of the SAS shortcut. Figure 14.6.3 shows a portion of the shortcut properties. In this figure, the %GLOBALRETRIEVE macro (see Section 14.6.2) is being called to restore saved global symbol table values.

**Figure 14.6.3: Target Dialogue Showing the -INITSTMT Option**



When executing in the batch environment, you can also execute statements at the end of the execution of your program through the use of the -TERMSTMT. Like the -INITSTMT option, this option is specified at the invocation of SAS; however, it is not used until SAS has finished executing all of your program statements. Interactive users can only take advantage of the -TERMSTMT option if they terminate their program with the ENDSAS statement.

## 14.7 Protecting Macros and Controlling Their Execution

One of the huge advantages of SAS is the level of control that the programmer can exert. The macro language adds to that control; however, sometimes as macro programmers we just want a bit more control over the creation and execution of the macros that we have written. When control-related issues are discussed by experienced macro programmers, common topics include the following:

- What version of a macro is being executed?
- How can we control for the correct version?
- How do we avoid macro variable collisions?

How do we protect our macro variables, compiled macros, and the source code?

Some of these issues become the roadblocks that tend to discourage application developers. On the other hand, these same challenges have spawned a number of SAS conference papers, with a number of innovative solutions.

In Sun and Carpenter (2011) many of these points are discussed in detail and will not be repeated in this book. Topics in that paper include the following:

- How are macros compiled and stored?
- How is Macro (library) search order defined, and how can it be protected?
- What is a macro variable collision, and how are they prevented?
- How can the version of the macro be managed and secured?
- How can we avoid macro reverse engineering?

Since that paper was written, the READONLY options on the %LOCAL and %GLOBAL statements have been introduced into the macro language (see Section 8.2.6). However, because of the limitations of this option on these statements, macro programmers must still understand about macro variable collisions and how to protect their macro variables.

# Appendix 1: Exercise Solutions

<b>Chapter 2 .....</b>	<b>433</b>
<b>Chapter 3 .....</b>	<b>435</b>
<b>Chapter 4 .....</b>	<b>436</b>
<b>Chapter 5 .....</b>	<b>437</b>
<b>Chapter 6 .....</b>	<b>440</b>
<b>Chapter 7 .....</b>	<b>444</b>
<b>Section 14.3.6 Quizlette.....</b>	<b>447</b>

Several of the chapters in this book contain a section titled “Testing Your Knowledge with Chapter Exercises.” These problems are designed to help you assess your understanding of the material that is contained within that chapter. Some of the exercises require the writing of SAS macros or code, and many of the questions have “solutions” that are included in this appendix.

Many of the solutions to exercises, especially those that require programming, should be considered *a* solution and not *the* solution. As the saying goes, “If it can be done one way, in SAS it can be done three ways.” Of course, the cynic then adds, “And the fourth way (*my way*) is the best.”

For the examples and solutions in this appendix and indeed for all of the code examples throughout the book, if you want to execute these sample programs, then be sure to follow the setup instructions. Remember that all of the data sets and programs are available for download, so you do not need to retype either the code or the data. For instructions on accessing and setting up the programs and data, see the “Example Code and Data” section within this edition’s “About This Book” front matter. You will find that not every macro language element is explained in every example. You can, however, find an explanation for macro language elements elsewhere in the book (see Appendix 4 to locate examples and explanations of various macro language elements).

---

## Chapter 2

1. Selected portions of text have been replaced with macro variables.

### Program A1.2.1: Chapter 2, Exercise 1 Solution

```
* Program A1.2.1
* Chapter 2 Question 1 solution;

%let dsn = clinics;
%let var1 = edu;
%let var2 = sex;

title1 'Counts of Gender vs. Years of Education';
proc tabulate data=macro3.&dsn;
    class &var2 &var1;
    table &var2=' ',&var1*n=' '/box=&var2;
    run;

title1 'Statistics on Heights for Each Gender';
```

```
proc univariate data=macro3.&dsn;
  class &var2;
  var ht;
run;
```

2. False.
3. You can use the following %PUT statements to test your answers.

#### **Program A1.2.3: Testing Macro Variable Resolution**

```
%let dsn=clinic;
%let lib=sasuser;
%let i=3;
%let dsn3 = studydrg;
%let b=dsn;

%put '&lib&dsn' &lib&dsn;
%put '&lib.&dsn' &lib.&dsn;
%put '&lib..&dsn' &lib..&dsn;

%put '&dsn&i' &dsn&i;
%put '&&dsn&i' &&dsn&i;
%put '&dsn.&i' &dsn.&i;

%put '&&bb' &&bb;
%put '&&&b' &&&b;

* Extra credit;
%put '&dsn..&&dsn&i' &dsn..&&dsn&i;
```

These produce the following SAS Log entry:

```
196 * Program A1.2.3
197 * Chapter 2 Question 3 solution;
198
199 %let dsn=clinic;
200 %let lib=sasuser;
201 %let i=3;
202 %let dsn3 = studydrg;
203 %let b=dsn;
204
205 %put '&lib&dsn' &lib&dsn;
'&lib&dsn' sasuserclinic
206 %put '&lib.&dsn' &lib.&dsn;
'&lib.&dsn' sasuserclinic
207 %put '&lib..&dsn' &lib..&dsn;
'&lib..&dsn' sasuser.clinic
208
209 %put '&dsn&i' &dsn&i;
'&dsn&i' clinic3
210 %put '&&dsn&i' &&dsn&i;
'&&dsn&i' studydrg
211 %put '&dsn.&i' &dsn.&i;
'&dsn.&i' clinic3
212
213 %put '&&bb' &&bb;
WARNING: Apparent symbolic reference BB not resolved.
'&&bb' &bb
214 %put '&&&b' &&&b;
'&&&b' clinic
215
216 * Extra credit;
217 %put '&dsn..&&dsn&i' &dsn..&&dsn&i;
'&dsn..&&dsn&i' clinic.studydrg
```

Notice the Warning for the %PUT in line 213. &&BB resolves to &BB, not dsnb.

The triple ampersand (line 214 in the SAS Log) resolves in two passes. On the first pass, && resolves to & and &B resolves to dsn leaving &DSN, which on the second pass is resolved to clinic.

4. Automatic macro variables are like any other macro variable except that they are initialized and defined by SAS.
5. &SYSDATE9 contains the date, in DATE9. form, such as 24oct2014, that the current SAS session was started. It will typically be the current date; however, if a session spans multiple days (a day boundary, midnight, is crossed) the date in &SYSDATE9 will not be updated and will not be current.

## Chapter 3

1. The macro %REPORTIT executes the desired code.

### Program A1.3.1: Generalized Macro

```
%let dsn = clinics;
%let var1 = edu;
%let var2 = sex;

%macro reportit;
title1 'Counts of Gender vs. Years of Education';
proc tabulate data=macro3.&dsn;
    class &var2 &var1;
    table &var2=' ',&var1*n=' '/box=&var2;
run;

title1 'Statistics on Heights for Each Gender';
proc univariate data=macro3.&dsn;
    class &var2;
    var ht;
    run;
%mend reportit;

%reportit
```

2. You should have at least tried the MPRINT, MLOGIC, and SYMBOLGEN system options. The MFILE option can also be very helpful.
3. True.
4. The definition of the %REPORTIT macro has been saved in the solutions folder.

### Program A1.3.4: Creating an Autocall Library

```
* Creating an autocall library;
* Assumes that the definition of the macro REPORTIT has been
* written to the location in the fileref MACLIB;

filename maclib "&path\Appendix 1 Exercise Solutions\SAS Programs";
options mautosource
        sasautos=(maclib, sasautos);

%let dsn = clinics;
%let var1 = edu;
%let var2 = sex;

%reportit
```

If the macro already exists in the WORK.SASMACR catalog, then you can delete it by using the %SYSMACDELETE statement (see Section 8.2.5).

```
%sysmacdelete reportit / nowarn;
```

## Chapter 4

1. POSITIONAL and KEYWORD (also called NAMED).
2. True—they can be specified in any order, but they must follow *all* positional parameters.
3. The following macro passes the data set name and two analysis variables.

### Program A1.4.3a: Positional Parameters

```
%macro reportit(dsn,var1,var2);
title1 'Counts of Gender vs. Years of Education';
proc tabulate data=macro3.&dsn;
  class &var2 &var1;
  table &var2=' ',&var1*n=' '/box=&var2;
  run;

title1 'Statistics on Heights for Each Gender';
proc univariate data=macro3.&dsn;
  class &var2;
  var ht;
  run;
%mend reportit;

%reportit(clinics,edu,sex)
```

The %REPORTIT macro in Program A1.4.3a requires only very minor modification to use keyword parameters. In this program only &VAR1 has a specified default value (EDU), while the other two parameters have a default of a null value.

### Program A1.4.3b: Using Keyword Parameters

```
%macro reportit(dsn=,var1=edu,var2=);
. . . . code not shown . . .
%mend reportit;

%reportit(dsn=clinics,var2=sex)
```

In the macro call, &VAR1 is not specified and will take on its default value.

4. Two positional parameters and one keyword parameter. The semicolon is required on the %MACRO statement, but not on the macro call.
5. For positional parameters, the order and placement of position determines the values to be passed. It is not possible to determine this placement when positional parameters are mixed with keyword parameters.
6. There are two syntax corrections required:

### Program A1.4.6: Corrected Definition of %MYCOPY

```
①%macro mycopy;

  proc copy in=work  out=master;
    select patients;
    run;

%mend mycopy;②
```

- ❶ % is a required part of the %MACRO statement.
- ❷ The macro name must be consistent on both the %MACRO and %MEND statements.

## Chapter 5

1. The expected macro call with mixed positional and named parameters would require that the positional parameter be specified first.

### Program A1.5.1: Mixed Parameter Types

```
%printt(macro3.clinics,proc=print)
```

Because positional parameters can be called using a named field, either of the following will also work – and could easily be considered to be preferable.

```
%printt(dsins=macro3.clinics,proc=print)
%printt(proc=print,dsins=macro3.clinics)
```

2. The three %DO loop forms are: iterative %DO, %DO %WHILE, and %DO %UNTIL.

### Program A1.5.2: Generating %DO Loops

```
%macro loop1;
  /* Using the iterative %DO;
  %do cnt = 1 %to 10;
    %put This is Test &cnt;
  %end;
%mend loop1;

%macro loop2;
  /* Using the %DO %WHILE;
  %let cnt=1;
  %do %while(&cnt <= 10);
    %put This is Test &cnt;
    %let cnt = %eval(&cnt + 1);
  %end;
%mend loop2;

%macro loop3;
  /* Using the %DO %UNTIL;
  %let cnt=1;
  %do %until(&cnt > 10);
    %put This is Test &cnt;
    %let cnt = %eval(&cnt + 1);
  %end;
%mend loop3;

%put loop1; %loop1
%put loop2; %loop2
%put loop3; %loop3
```

3. True.
4. True.
5. Which statement(s) below is (are) syntactically incorrect? Why?
  - A. %RETURN ABEND;

The %RETURN statement has no options. The ABEND option is considered to be extraneous text and is ignored. The %RETURN executes.

B. %GLOBLE SAVE;

Most likely %GLOBAL has been misspelled as %GLOBLE, but must this be an error? What if a macro %GLOBLE was defined? Would the execution of the %GLOBLE macro cause a syntax error when combined with the SAVE; ? Probably, but at this point we do not know for sure. The point is this: as macro programmers, our product is code, not the result of the execution of that code. %GLOBLE might generate code that is compatible with the SAVE;. We don't know. But probably it is an error.

C. %IF &SEX = M THEN %PUT MALE;

Either there is an extra % on the IF or a % is missing from the THEN. Which is it?

Could this be a DATA step IF? For sure not, because a DATA step IF could **NEVER** be used to conditionally execute a macro language element; therefore, this has to be a macro %IF and the % is missing from the %THEN.

6. Macro %GENPROC passes in the name of the data set as well as the name of the procedure and analysis variables.

#### Program A1.5.6: Generic PROC Step

```
%macro genproc(proc=,dsn=,varlst=);
  title1 "&proc Procedure for &dsn";
  proc &proc data=&dsn;
    var &varlst;
    run;
%mend genproc;
%genproc(proc=means,dsn=macro3.clinics, varlst=edu ht wt)
%genproc(proc=univariate,dsn=macro3.clinics, varlst=edu ht wt)
```

Because commas are used to separate parameters, the list of variables (separated by spaces) can be passed in a single macro variable &VARLST.

7. The macro %MYMEANS contains a PROC MEANS that can produce printed output or an output data set.

#### Program A1.5.7: Generalize a PROC MEANS Step

```
%macro mymeans(dsn=,      varlst=, statlst= mean max,
               outdsn=, print=noprint);
proc means data=&dsn &statlst ①
  %if &outdsn = %then print; ②
  %else &print; ③
  ; ④

  var &varlst;
  %if &outdsn ne %then %do; ⑤
    output out=&outdsn mean= max= / autoname; ⑥
  %end;
  run;
%mend mymeans;

* print selected stats (no output data set);
%mymeans(dsn=macro3.clinics,
          varlst=ht wt,
          statlst=mean stderr) ⑦

* no printed stats (output data set only);
%mymeans(dsn=macro3.clinics,
          varlst=ht wt,
          outdsn=outstat) ⑧
proc print data=outstat;
```

```

run;

* printed stats & an output data set ;
%mymeans(dsn=macro3.clinics,
    varlst=ht wt,
    statlst=sum max,
    outdsn=outstat,
    print=print) ⑨

```

- ➊ &STATLST will contain the list of statistics to be printed.
  - ➋ When an output data set is not named in &OUTDSN, printed statistics are requested using the PRINT option.
  - ➌ Printed statistics can be requested even if an output data set is requested by passing PRINT into &PRINT.
  - ➍ This semicolon closes the PROC statement.
  - ➎ An OUTPUT statement is generated if an output data set is named in &OUTDSN.
  - ➏ The AUTONAME option generates unique names for each statistic and analysis variable combination.
  - ➐ This call to %MYMEANS will result in only printed output because the parameter &OUTDSN is not specified.
  - ➑ This call to %MYMEANS produces an output data set (OUTSTAT) and no printed output.
  - ➒ This call to %MYMEANS produces both an output data set (OUTSTAT) and the printed statistics (SUM and MAX).
8. Macro variables that are passed into a macro are necessarily local to that macro – no %LOCAL statement is required. The macro will operate correctly. It would not be a bad idea, however, to add the %LOCAL statement to %REGTEST.
  9. The SAS Log shows that, when included, the quotes become part of the comparison.

#### Program A1.5.9 (SAS Log): Showing How Quotes Become Part of the Comparison

```

73  %let a = AAA;
74  %macro try;
75      %put &a;
76      %if &a      =  AAA  %then %put no quotes;
77      %if '&a'   =  'AAA' %then %put single quotes;
78      %if 'AAA'   =  'AAA' %then %put exact strings;
79      %if "&a"   =  "AAA" %then %put double quotes;
80      %if "&a"   =  'AAA' %then %put mixed quotes;
81      %if "&a"   =  AAA  %then %put quotes on one side only;
82  %mend try;
83  %try
A=AAA
no quotes
exact strings
double quotes

```

10. Because we are now dealing with four-digit years, very little has to be done with this code. Notice that the more efficient RETAIN statement has replaced the assignment statement for YEAR. In the Chapter 7 exercises, this program is again rewritten to accommodate two-digit years that cross the century boundary.

#### Program A1.5.10: Accommodating Four-Digit Years

```
%macro allyr(start=2004,stop=2005);
  %do year = &start %to &stop;
    data temp;
      set yr&year;
      retain year &year;
      run;
    proc datasets lib=work nolist;
      append base=allyear data=temp;
    quit;
  %end;
%mend allyr;
%allyr(start=1999, stop=2002)
```

## Chapter 6

1. One will be right-justified and the other left-justified and trimmed. The %PUT statements write the values to the SAS Log:

```
404 %put |&mwt|;
| 50.5|
405 %put |&mwt1|;
|50.5|
```

2. In the CLINICS data set, there are ten regions that are numbered from '1' to '10'. The following solution would not work if the region values were not consecutive, starting at one.

#### Program A1.6.2: Breaking Up a Data Set into Parts

```
%macro RegionRpt;
  * Build a macro variable for each level of REGION;
  proc sql noprint;
    select distinct region
    into :reg1 - ❶
    from macro3.clinics;
  %let total = &sqlobs; ❷
  quit;

  * Break up the data set into one per region;
  data
    %* Build the names of the new data sets;
    %do i = 1 %to &total;
      reg&i ❸
    %end;
    ;
    set macro3.clinics;
    %do i = 1 %to &total;
      %* Build the output statements;
      %if &i^=1 %then else; ❹
        if region="&i" then output reg&i; ❺
      %end;
    run;

  * Separate analyses for each level of REGION;
  %do i=1 %to &total; ❻
    title1 "Region: &&reg&i";
```

```

proc print data=reg&i(obs=10);
  var clinname sex edu ht wt;
  run;
proc report data=reg&i
  nowd;
  column clinname ht wt;
  define clinname / group;
  define ht / mean;
  define wt / mean;
  run;
proc tabulate data=reg&i;
  class sex edu;
  var ht wt;
  table edu, sex*(ht wt)*(n mean stderr);
  run;
%end;
%mend regionrpt;
%regionrpt

```

- ❶ The value of the  $I^{\text{th}}$  region is stored in &REG<sub>i</sub>.
- ❷ The total number of regions is stored in &TOTAL.
- ❸ The DATA statement is built dynamically using the %DO loop. Each data set will have the name REG1, REG2, and so on.
- ❹ The ELSE is added to all IF-THEN statements except for the first one.
- ❺ The IF-THEN statements are built dynamically for each region. These statements depend on the values of the variable REGION being consecutive and starting at one. This is a limitation of this solution.
- ❻ This loop executes a separate series of procedure steps for each region.

The limitations of this approach are removed in the solution for Question 3.

3. The names of the individual regions are now reflected in the names of the data subsets. The data is used as the control file to build the list of regions. This solution does not require that the values of REGION take on any particular sequence or set of values.

#### Program A1.6.3: Using the Data Itself to Name the Data Subsets

```

%macro RegionRpt(dsn=macro3.clinics);
  %local i;
  * Build a macro variable for each level of REGION;
  proc sql noprint;
    select distinct region
    into :reg1 -
      from &dsn;
    %let total = &sqllobs;
    quit;

  * Break up the data set into one per region;
  data
    %* Build the names of the new data sets;
    %do i = 1 %to &total;
      reg_&&reg&i ❶
    %end;
    ;
    set &dsn;
    %do i = 1 %to &total;
      %* Build the output statements;
      %if &i^=1 %then else;
      if region="&reg&i" then output reg_&&reg&i; ❷
    %end;
    run;

```

```

* Separate analyses for each level of REGION;
%do i=1 %to &total;
  title1 "Region: &reg&i";
  proc print data=reg_&reg&i(obs=10);
    var clinname sex edu ht wt;
    run;
  proc report data=reg_&reg&i
    nowd;
    column clinname ht wt;
    define clinname / group;
    define ht / mean;
    define wt / mean;
    run;
  proc tabulate data=reg_&reg&i;
    class sex edu;
    var ht wt;
    table edu, sex*(ht wt)*(n mean stderr);
    run;
%end;
%mend regionrpt;

```

- ❶ The data sets created will be named REG\_xxx, where xxx is the value of the variable REGION.
- ❷ &&REG&I contains the value of the variable REGION for the &I<sup>th</sup> region.

Suppose the variable REGION took on the values '1', '23Q', and '42'.

I	REGION	REG_&&REG&I
1	1	REG_1
2	23Q	REG_23Q
3	42	REG_42

**MORE INFORMATION:** Some discussion of this macro can be found in Section 11.1.8.

**SEE ALSO:** Sridharma (2003) splits a data set into  $N$  data sets based on the number of observations.

4. The macro %ALLCHAR converts all numeric variables to character, including the use of any available formats.

#### Program A1.6.4: Changing All Numeric Variables to Character

```

%macro allchar(dsn=);
%local i;

* Determine the numeric vars in &dsn;
proc contents data=&dsn
  out=cont(where=(type=1)) ❶
  keep=name type format formatl formatd label)
  noprint;
run;

* Create the macro variables for each numeric var;
data _null_;
  set cont end=eof;
  length fmt $15;

```

```

* Count the numeric vars and save the total number;
ii+1;
ii=left(put(i,3.));
if eof then call symputx('n',ii,'l'); ②

* create a format string;
fmt = 'best.';
if format ne ' ' then fmt = catt(format,formatl,'.',formatd);
call symputx('fmt'||ii,fmt,'l'); ③

* Save the variable name;
call symputx('name'||ii,name); ④

* Save the label for this variable;
if label = ' ' then label = name;
call symputx('label'||ii,label); ⑤
run;

* Establish a data set with only character variables;
* &n      number of numeric variables in &dsn;
* __aa&i   temporary var to hold numeric values;
* &&name&i name of the variable to convert from numeric;
*
* The numeric value of &name1 is stored in __aa1
* by renaming the variable in the SET statement. __aa1
* is then converted to character and stored in the
* 'new' variable &name1 in the data set CHARONLY.
* ;
data charonly (drop=
  %* Drop the temp. vars used to hold numeric values;
  %do i=1 %to &n;
    __aa&i ⑥
  %end;
  );
length
  %* Establish the vars as character;
  %do i=1 %to &n;
    &&name&i
  %end;
  $8;

set &dsn (rename=(

  %* Rename the incoming numeric var to a temp name;
  %* This allows the reuse of the variables name;
  %do i=1 %to &n;
    &&name&i=__aa&i ⑦
  %end;
  ));

  * Convert the numeric values to character;
  %do i=1 %to &n;
    &&name&i = left(put(__aa&i,&&fmt&i)); ⑧
    label &&name&i = "&&label&i";
  %end;
run;

proc contents data=charonly;
proc print data=charonly;
run;
%mend allchar;

%allchar(dsn=macro3.biomass)

```

- ❶ Numeric variables have TYPE=1.
- ❷ Save the number of numeric variables.
- ❸ Save the format for this numeric variable.
- ❹ Save the name of the variable.
- ❺ Save the label of the variable.
- ❻ Incoming numeric variables will be temporarily renamed using the RENAME= data set option ❷, and these temporary names need to be dropped. This solution assumes that there are no incoming variables named \_AA1, \_AA2, \_AA3, and so on.
- ❼ The RENAME= option is built dynamically inside of the %DO loop.
- ❽ Use the PUT function to convert the numeric variable that is temporarily stored in \_AA&I to character using the appropriate format (&&FMT&I), and the name that is stored in &&NAME&I.

**SEE ALSO:** Roberts (1997) uses an analogous approach to converting all character variables to uppercase. Varney (2016) uses macro variable lists to build variables with predetermined lengths.

5. The CALL EXECUTE routine can be used to directly execute a macro from within a DATA step.

#### Program A1.6.5: Using CALL EXECUTE to Call a Macro

```
data splist;
  infile datalines truncover;
  input species $9.;
  datalines;
Parkki
Perch
Pike
run;
%macro splist(sp=);
title1 "Average Weight of &sp";
proc means data=sashelp.fish n mean;
  where species=&sp;
  var weight;
  run;
%mend splist;
data _null_;
  set splist; ❶
  call execute(catt('%splist(sp=',species,')'));
run;
```

The CALL EXECUTE would generate three macro calls, one for each species of fish.

```
%splist(sp=Parkki)
%splist(sp=Perch)
%splist(sp=Pike)
```

---

## Chapter 7

1. A, B, C, and D. Give yourself half credit if you selected F. Technically, F (All of the above) includes E, which is clearly false.
2. True.
3. Mostly true. It is a compilation time function, but some of the special characters, such as quotes or parentheses, are not masked.

4. The following macro can be used to count the number of words in a string. An optional second parameter (&PARM) can be used as a word delimiter.

#### Program A1.7.4: Counting Words in a String

```
%macro count(string,parm);
  %local count word; ①
  %if &parm= %then %let parm = %str( ); ②
  %let count=0;
  %let word = %qscan(&string,&count+1,&parm); ③
  %do %while(&word ne);
    %let count = %eval(&count+1);
    %let word = %qscan(&string,&count+1,&parm); ③
  %end;
  &count ④
%mend count;
```

- ① &COUNT and &WORD will be local.
- ② When no parameter is passed in, use a blank to separate words.
- ③ Retrieve the word designated by &COUNT+1. &PARM holds the word separator.
- ④ &COUNT contains the number of words.

The use of the %COUNT macro can be demonstrated by calling it with different word lists. The SAS Log shows:

```
28  %put %count( );
0
29  %put %count(this is a short string);
5
30  %put %count(this is a short string, %str( ));
5
31  %put %count(%nrstr(this*&is*a*string),%str(*));
4
32  %put
%count(%nrstr(this,&is,a,comma,separated,string),%bquote(,));
6
```

**SEE ALSO:** Abdurazak (2002) has a similar word counting macro.

5. The data set name follows the period that separates the *libref* from the data set name. Determine the location of the period and use %SUBSTR to grab the name.

#### Program A1.7.5: Extracting the Name Portion of a Permanent Data Set Name

```
%macro nameonly(dsn);
  %local len col name;

  %let len = %length(&dsn);
  %let col = %index(&dsn,.);
  %let name = %substr(&dsn,%eval(&col+1),%eval(&len-&col));
  /*%let name = %substr(&dsn,%eval(&col+1));
  &name
  %mend nameonly;
```

#### Extra Credit:

If you take advantage of the period and use it as a word delimiter, the %SCAN function can be used to separate the data set name:

```
%macro nameonly(dsn);
  %scan(&dsn,2,.)
%mend nameonly;
```

A stronger version, which reads from right to left, does not require the presence of a *libref*:

```
%macro nameonly(dsn);
  %scan(&dsn,-1,.)
%mend nameonly;
```

6. Remove the check for the first occurrence. &WNUM will be zero until the selected word is detected. Without this check, &WNUM will be updated for each occurrence of the selected word (&WORD) and will hold the word number of the last occurrence when exiting the loop.

#### Program A1.7.6: Return the Word Number for the Last Occurrence of a Word

```
%macro revscan(list, word);
  %local wcnt wnum;
  %let wcnt=0;
  %let wnum=0;
  /* Determine the word number in a list of words;
  %do %while(%scan(&list,%eval(&wcnt+1),%str( )) ne %str()
    /*& &wnum=0*/);
  %let wcnt = %eval(&wcnt+1);
  %if %upcase(%scan(&list,&wcnt,%str( )))=%upcase(&word) %then
    %let wnum=&wcnt;
  %end;
  &wnum
%mend revscan;
%put %revscan(aa a bb cc a dd,a);
```

7. Care must be taken to control the year when incrementing across the century boundary. When incrementing using two-digit years, care must be taken to ensure that you have leading zeros for values less than 10.

#### Program A1.7.7: Looping across the Century Boundary with Two-Digit Years

```
%macro allyr(start=04,stop=05); ①
/*   /* assume a two digit yearcutoff of 1920; */
/*   /* (1920 is the default for SAS9.3) */
/*   /* (1926 is the default for SAS9.4)*/
/*   %if &start <= 20 %then %let first = 2000 + &start; */ ②
/*   %else %let first = 1900 + &start; */
/*   %if &stop <= 20 %then %let last = 2000 + &stop; */
/*   %else %let last = 1900 + &stop; */
%local first last year yr;
/* Use the current setting of the YEARCUTOFF option;
%let first = %sysfunc(year(%sysfunc(mdy(1,1,&start)))); ③
%let last = %sysfunc(year(%sysfunc(mdy(1,1,&stop)))); ④
%do year = &first %to &last; ⑤
  /* Create a two digit year;
  %let yr = %sysfunc(mod(&year,100),z2.); ⑥
/*   /* Test values; */
/*   %put &=first &=last &=year &=yr; */ ⑦
  data temp;
    set yr&yr; ⑧
    year = &year;
    run;
  proc datasets lib=work nolist;
    append base=allyear data=temp;
    quit;
%end;
```

```
%mend allyr;
%allyr(start=98, stop=11) ⑨
```

- ❶ The first and last two digit years are incoming parameters.
- ❷ The four-digit year could be constructed using logic and a hardcoded definition of the YEARCUTOFF value.
- ❸ Use the incoming two digit year to build a four-digit year. By using the MDY function to construct a date, we automatically use the current definition of the YEARCUTOFF option.
- ❹ Loop across the years of interest using the four-digit years
- ❺ Extract the last two digits from the four-digit year. A leading zero is added for values of &YR less than 10 through the use of the Z2. format.
- ❻ It is always a good idea to test the creation of the macro variables before using them.
- ❼ The two-digit year (&YR), with a leading zero if needed, is used in the data set name.
- ❽ The four-digit year is assigned to YEAR.
- ❾ The macro call uses two-digit years, and it is assumed that 98 is before 11

### Section 14.3.6 Quizlette

The discussion surrounding Program 14.3.6 demonstrates how it is a bad idea to use asterisk style comments to mask macro language elements within a macro definition. That discussion centered on the result of a call to the %DOIT macro, and how the macro language elements were not commented.

#### Program 14.3.6e: Using the Asterisk to Mask Macro Language Elements

```
%macro abc;
  *%put in abc;
%mend abc;

%macro doit;
  *%abc
  %put here;
  *%let x = %abc;
  %put value of x is &x;
%mend doit;
```

The question posed at the end of that discussion was what would happen if the macro call was instead made from within a %PUT statement.

```
%PUT %DOIT;
```

The %PUT allows all the generated text, including the asterisks, to be surfaced. This gives us a better understanding of what is going on behind the scenes when %DOIT is executed.

The SAS Log will contain the following:

```
in abc
here
in abc
value of x is *
**      *
```

Notice how the generated asterisks (the ones intended to comment out the macro code) are written at the bottom of the resultant text.

Notice that the %ABC macro is executed twice, not just once, and that the %PUT writes out the non-macro asterisks into the SAS Log. To see where these asterisks come from, let's replace a couple of them with the letters A and B. The macro becomes

```
%macro abc;
  A%put in abc;
%mend abc;

%macro doit;
  *%abc ①
  %put here; ②
  B%let x = %abc; ③
  %put value of x is &x;
%mend doit;
```

The SAS Log shows the following:

```
59  %put %doit;
in abc ①
here ②
in abc
value of x is A ③
*A      B
```

The %PUT writes \*A B to the SAS Log.

- ① \*%ABC is not seen as a comment so the asterisk is left behind (written by the %PUT to the SAS Log) and %ABC is executed. %ABC writes in abc to the SAS Log and adds an A to the text that will be written by %PUT.
- ② This %PUT shows that %ABC has already been executed.
- ③ The %LET also executes %ABC, but this time the letter A is stored in &X. The B is then added to the text that will be written by the final %PUT. The spaces between the A and the B are a result of the blank lines and extra spaces in %ABC and %DOIT.

## **Appendix 2: Using the Macro Language with Compiled Programs**

<b>A2.1 The Problem: Macro Variable Resolution during Compilation .....</b>	<b>450</b>
<b>A2.2 Using Macro Variables.....</b>	<b>451</b>
A2.2.1 Defining Macro Variables.....	451
A2.2.2 Macro Variables in SCL SUBMIT Blocks .....	452
A2.2.3 Using Macro Variables in SCL.....	453
A2.2.4 Passing Macro Values between SCL Entries .....	453
A2.2.5 Using &&VAR&I Macro Arrays in SCL Programs .....	454
<b>A2.3 Calling Macros from within Compiled Programs .....</b>	<b>454</b>
A2.3.1 Run-Time Macros .....	454
A2.3.2 Compile-Time Macros .....	455
<b>A2.4 Using the Macro Language with FCMP Functions .....</b>	<b>457</b>
A2.4.1 Compile-Time Execution .....	457
A2.4.2 Executing a Macro during Function Execution.....	457

Generally SAS code is compiled and immediately executed. This is not always the case however. There are a number of situations where the code is compiled and the compiled code is saved for execution later. This includes the use of compiled views, compiled DATA steps, SAS Component Language (SCL — formerly known as Screen Control Language) programs, and PROC FCMP functions. The macro facility is available in each of these situations; however, you have to be careful how the macro language elements are addressed. Because the compilation process resolves and executes macro references, you might need to adjust your perception of the order and sequence of macro-related events.

During the compilation of language elements like FCMP functions, all macro references are resolved to their current values. A macro call, for instance, will be executed during the compilation phase and not when the element is executed. This is a different way of thinking about macro references, and this appendix is designed to help you deal with these differences.

These are not common issues for most macro programmers, which is why this is a topic relegated to this appendix. When it is a problem it most commonly appears in SCL programs; consequently the bulk of these examples are centered around SCL. However, much of the SCL discussion applies equally to other types of compiled programs. Section A2.4 addresses topics specific to PROC FCMP.

**MORE INFORMATION:** Although not discussed in this appendix, similar issues can be encountered when working with macro windows (see Section 8.2.3 for more on the %WINDOW statement).

**SEE ALSO:** *Beyond the Obvious with SAS Screen Control Language* (Stanley, 1994) contains a summary of warnings and tips on the use of macros and macro variables in SCL programs (pp. 40–43).

“Using Macro Variables in SCL Programs,” in *SAS Component Language Reference 9.4: Reference, Third Edition* discusses macro variables, and the substitution of text in SUBMIT blocks.

<http://support.sas.com/documentation/cdl/en/sceref/67564/HTML/default/viewer.htm#n1mjbf5azcquann1oe605j0rcfq4.htm>

Norton (1991) contrasts the use of the macro language with SCL and suggests that SCL be used as an alternative to the macro language in some situations.

Davis (1997) shows how you can use macros to provide consistency among SCL frame entries.

Ward (1999) uses macros with frame entries to document and manage programs.

## A2.1 The Problem: Macro Variable Resolution during Compilation

For compiled programs, external references, such as macro variables and macro calls, are resolved during compilation, not during execution. The examples shown in this section are from a portion of an SCL program, but the concepts apply equally to the other types of compiled programs such as functions created using PROC FCMP, compiled views, compiled DATA steps, macro windows created with %WINDOW, and remote submittals.

The writer of the following SCL INIT section would like to open the data set named in the macro variable &DSN when the INIT section executes:

```
init:
  * open the data set &dsn;
  dsnid = open("&dsn",'i');
  if dsnid=0 then put "unable to open &dsn";
return;
```

This code will fail even to compile unless &DSN is currently defined at the time of compilation. If &DSN is defined at compile time to be `clinics`, then the code that is compiled will be:

```
init:
  * open the data set clinics;
  dsnid = open("clinics",'i');
  if dsnid=0 then put "unable to open clinics";
return;
```

Subsequent changes to the macro variable &DSN will have no effect on what data set will appear in the OPEN function.

A similar problem exists for macro calls inside of compiled programs. The following code is supposed to execute an error-check macro (%ERRCHK) when the file named by the SCL variable DSNAME cannot be opened:

```
init:
  * open the data set &dsn;
  dsnid = open(dsname,'i');
  if dsnid=0 then %errchk(dsname);
return;
```

The macro %ERRCHK is supposed to write the name of the missing data set (which is passed into the macro as a parameter) to the SAS Log. The %ERRCHK macro has previously been defined as:

```
%macro errchk(d);
  %put "data set not found &d";
%mend errchk;
```

Regardless of the name of the unknown data set, the SAS Log will contain the following:

```
"data set not found dsname"
```

The macro call is resolved and the macro executed when the SCL program containing the macro call is compiled. At compile time, DSNAME in the macro call is seen as text, not as an SCL variable. DSNAME

is, therefore, passed unresolved to the macro. Here, &D will take on the value of dsname and the %PUT is executed. This example highlights the major disadvantages of using macro calls in compiled programs:

- The macro must be defined before SCL compilation.
- SCL variable values cannot be passed into the macro through a macro call.
- Changes to the macro will not be reflected in the SCL program until the SCL program is recompiled.

Despite these caveats, there are very definite uses for macro variables and macro calls within compiled programs. The following sections in this appendix cover these topics in more detail.

## A2.2 Using Macro Variables

Macro variables are not necessarily part of any particular application or data set, and they are not associated with a particular screen, method, or program. This makes them ideal for passing information through various portions of your program or application. Once defined in the global symbol table the values of the macro variables are available throughout the SAS session.

For SCL applications macro variables are especially useful when passing information between entries in your application. Values that are set once and then repeated or held constant across screens or data sets are prime candidates for use with macro variables. Macro variables are often used to hold values for:

- the name of a SAS data set to be opened or the file identifier of an open data set
- the external filename to be opened or its file identifier once it is opened
- constant text such as the current date
- values passed between applications
- values passed between programs within an application.

In compiled programs macro variables must be mentioned obliquely if you want the execution time value to be used. Outside of the SUBMIT block, it is unlikely that you will use the ampersand much, if at all, in SCL programs. When they are used directly, it is usually to build code.

### A2.2.1 Defining Macro Variables

You can assign values to macro variables in compiled programs in much the same way as you would in other SAS programs. However, you will probably depend more on the CALL SYMPUTX routine and less on the %LET statement. Like %LET statements in non-SCL programs, the characters on the right of the equal sign are taken to be text. Because %LET is executed when the SCL program is compiled, SCL variables are unresolvable. The following %LET statement creates the macro variable DSN, which will contain the text string dsname:

```
%let dsn = dsname;
```

Even if DSNAME is an SCL variable (screen or nonscreen), the value of DSN is the **text string** dsname.

Rather than using the macro language to define the macro variable, the SYMPUTX routine is used **at run time** to create macro variables. This routine is used the same in SCL as it is in a DATA step (see Section 6.1 for more information on the SYMPUTX routine and its usage). If DSNAME is a character SCL variable that contains the string `macro3.clinics`, the following statement assigns the value taken on by DSNAME, that is, `macro3.clinics`, to the macro variable DSN:

```
call symputx('dsn',dsname);
```

If you use SYMPUT instead of SYMPUTX, the second argument of SYMPUT is expected to be a character string or the name of a character variable. SYMPUTX overcomes this limitation, and SCL also

has the SYMPUTN routine, which performs similarly. Both of these routines can also be used to create macro variables from numeric SCL variables. In the following example, the numeric SCL variable OBS contains the observation number of interest:

```
call symputn('obsnum', obs);
```

Often the macro variables that are created within an application are globalized so that they will be available throughout the application. Unless the macro variables have been previously defined as local, the SCL SYMPUT and SYMPUTN routines always create globalized macro variables.

### A2.2.2 Macro Variables in SCL SUBMIT Blocks

SUBMIT blocks are used in SCL programs to pass statements to SAS for execution. Statements that are contained in SUBMIT blocks are not SCL statements, but are standard SAS language statements. However, these statements can contain references to both SCL and macro variables.

Inside of a SUBMIT block, SCL variables are identified by preceding them with an ampersand. Since this is the same nomenclature as is used to identify macro variables, there is the possibility of confusion. In the following section of SCL, a SUBMIT block is used to execute the PRINT procedure:

```
if modified(rpt) then submit;
  title 'Data Listing for &dsname';
  proc print data=&dsname(obs=10);
    run;
endsubmit;
```

A couple of things are worth noting in this code. At compilation the &DSNAME is not seen as a macro variable reference because it is inside of a submit block. At the execution of the SCL program and before the SUBMIT block is passed to the SAS processor for execution, the reference &DSNAME is first checked against the list of SCL variables in the SCL Data Vector (SDV) and resolved if found. If it is not found on the SDV, it is then passed unresolved (still containing the &) for execution, where it will be treated as a macro variable reference. Because of this behavior, when passing values into SUBMIT blocks it is generally not a good idea to use the same name for both macro variables and SCL variables.

You might also have noted in the previous example that the title is enclosed by single quotation marks. In SCL SUBMIT blocks, single quotes will **not** mask the meaning of the ampersand as they will in Base SAS language statements. However, if the &DSNAME is passed unresolved, in single quotes, to Base SAS for processing, the macro variable reference will remain quoted. In both the previous and in the following example, if DSNAME is not an SCL variable, it will remain unresolved in the title.

If you do need to specify a macro variable in a SUBMIT block and an SCL variable exists with the same name, you can use a double ampersand to prevent its resolution as an SCL variable. In the following example, the &&VARLST will not resolve to the SCL variable VARLST even if VARLST exists on the SDV:

```
if modified(rpt) then submit;
  title 'Data Listing for &dsname';
  proc print data=&dsname(obs=10);
    var &&varlst;
    run;
endsubmit;
```

The double ampersand is resolved to one ampersand, as would happen within the macro facility; however, a second scanning pass is not made on the submit block code until it is executed. It is at that time that &VARLST will be interpreted as a macro variable.

### A2.2.3 Using Macro Variables in SCL

The automatic macro variables that are discussed in Section 2.6 are all available within SCL programs. As was noted in Section A2.2.1 however, you need to be careful how you use them. The following SCL code attempts to open the most recently modified data set:

```
dsname = "&syslast";
dsid = open(dsname);
```

The code will not do what the programmer wants because &SYSLAST will be resolved when the SCL program is compiled. Upon execution the program will always try to open the same data set regardless of the value of &SYSLAST at the time of execution.

The problem is solved by using the SYMGET function. Because SYMGET is not a macro language element, it will be executed along with the rest of the SCL code. In the following revised SCL code, the value of &SYSLAST is not retrieved until the SCL program is executed:

```
dsname = symget('syslast');
dsid = open(dsname);
```

When you want to create a numeric SCL variable from a macro variable, you should use the SYMGETN function. SYMGETN returns a numeric value rather than the character value that is returned by SYMGET.

### A2.2.4 Passing Macro Values between SCL Entries

Passing macro variable values between SCL entries is very straightforward. Macro variables are defined using the SYMPUT, SYMPUTX, or SYMPUTN routines, and these macro variables are then available throughout the application. A subsequent SCL entry (or even code within a SUBMIT block) can retrieve the values by using SYMGET and SYMGETN.

The following INIT section both retrieves macro variables, and creates them for later use:

```
INIT:
  * Specify a macro var used for SCL in edit screens;
  call symputx('scrntryp','DE'); ①

  * Create a libref for the log used
  * by this Data Entry userid;
  userid = symget('userid'); ②
  tst = symget('tst'); ②
  path = compress('h:\studyx\phase2\' 
    ||tst||'datprep\d_entry\' 
    ||userid);
  call libname('delog',path);

  control enter;
  cursor subject;
return;
```

- ① The macro variable SCRNTYPE is initialized to DE.
- ② The macro variables USERID and TST (both of which must have been created earlier in the application) are retrieved from the symbol table and placed in SCL variables of the same name. Remember, using the same name for macro and SCL variables is acceptable and causes problems only when used carelessly in SUBMIT blocks.

Macro variables often might not be the best method for passing values between entries within an application or even between applications. Built into SAS Component Language is the concept of an SCL LIST. Analogous to an ARRAY list, entries can be loaded from files, saved as files, and passed from one entry to another. Like macro variables, LISTS can be global or local. But because LISTS are designed to be an integral part of SCL (macro variables just coexist with SCL), they work more smoothly and have

additional support functions. It is likely that the use of SCL LIST functions will be quicker than SYMGET and SYMPUTX.

### A2.2.5 Using &&VAR&I Macro Arrays in SCL Programs

Since it is not possible to resolve the &&VAR&I macro variable form within an SCL program during program execution, macro variable lists of this form are accessed by using the SYMGET and SYMGETN functions to create SCL variables.

Like in the macro language, when you want to step through a series of macro variables that have been created with a subscript, a DO loop is used. This time, of course, it will be an SCL DO loop rather than a %DO loop, and the resulting SCL variable (I) will be used as the index to identify the specific macro variable.

In the following example a series of data set names have been stored in the macro variables &LIVEDB1, &LIVEDB2, and so on, and we would now like to step through the list of data sets from within an SCL program.

```
 . . . portions of the code not shown . . .
* Step through the list of data sets;
cnt = symgetn('livecnt');
do i=1 to cnt; ①
    ii = left(put(i,3.)); ②
    * Get the data set name and open it. ;
    dsn = 'datamgt.'||left(symget('livedb'||ii ③)); ④
    dsid = open(dsn);
. . . portions of the code not shown . . .
```

- ① Specify the SCL DO loop (rather than a %DO loop).
- ② The SCL loop creates a numeric index variable, which is converted to character.
- ③ This index is then appended to the root name of the macro variable series.
- ④ The concatenated value is retrieved using SYMGET or SYMGETN. The conversion at ② could have been avoided by using the CAT or CATT functions when building DSN. The assignment statement becomes:

```
dsn = catt('datamgt.',left(symget(catt('livedb',i))));
```

---

### A2.3 Calling Macros from within Compiled Programs

The brief example using %ERRCHK in Section A2.1 illustrates the problem associated with calling macros from within compiled programs. Remember, macros are executed when the SCL program is compiled. This means that when the macro changes, programs that use that macro will need to be recompiled. Despite this problem, macro calls can have a definite place in compiled programs.

Macros called in compiled programs fall into two classes, and these are determined by when the macro is to be executed. Macros in compiled programs can be executed either when the program is compiled (compile-time) or when it is executed (run-time).

---

#### A2.3.1 Run-Time Macros

*Run-time* macros are macros that execute when the compiled program executes. Unlike macros in the base language, which are all run-time, this concept has very little meaning in SCL programs. The exception is found in SUBMIT blocks. Because SAS does not resolve macro references in SUBMIT blocks until the block itself is executed, you can safely call macros here. Because the block of code is essentially set aside, the called macro does not need to exist until the SUBMIT block is actually executed.

Run-time macros avoid the SCL compilation issues that are noted in Section A2.1 and behave as other base system macros behave.

### A2.3.2 Compile-Time Macros

*Compile-time* macros are most often used to write code. These macros will execute when the SCL program is compiled—long before values for the SCL variables are available. If the macro contains reusable code but is not actually generating SCL, as in the example shown later in this section, consider using a METHOD instead of a macro. There is an informative discussion on the advantages and disadvantages of METHODS and compile-time macros in *Beyond the Obvious with SAS Screen Control Language* by Don Stanley (1994, pp. 42–43).

The example shown in this section illustrates the use of a macro to generate SCL code. A series of over 20 FSEDIT screens were to have similar (but not quite the same) SCL that was used to initialize protected variables. Rather than develop a series of parallel programs that would be difficult to maintain, the entire SCL program for each screen was placed within a single generalized macro. The source screen for each FSEDIT consisted only of the call to the macro %DATASTMP, with the macro parameters completely specifying the specifics for each screen definition:

```
%datastmp(subject,ptid)
```

For this usage of %DATASTMP the macro arguments are names of SCL variables that will apply to a specific data set and screen. Remember, the *names* of the variables are being passed to the macro not the variable values, which are not yet available.

```
%macro datastmp(var1,var2,var3,var4);

* determine the number of vars;
%do i = 1 %to 4; ①
  %if &&var&i ne %then %let varcnt = &i;
%end;

fseinit:
  scrntrtype=symget('scrntrtype');
  if scrntrtype in ('CLN', 'PED') then do;
    control enter;

  ...ordinary SCL not shown...
return;

init:
  if scrntrtype='DE' or word(1)='ADD' then do;
    %do i = 1 %to &varcnt; ②
      unprotect &&var&i; ③
      &&var&i = symget("&&var&i"); ④
      protect &&var&i;
    %end;
    end;
return;

  ...ordinary SCL not shown...

%mend datastmp;
```

- ① The number of non-null parameters is counted. This counter is used in the %DO loop that writes the protection and assignments for the specified variables in the INIT section. The count is stored in &VARCOUNT.
- ② The variable &VARCOUNT is used as the upper bound for the %DO loop.

- ③ Each variable is to be unprotected so that it can receive a value from a macro variable of the same name. Because this %DO loop executes during the compilation phase, the macro variable form &&VAR&I can be used, and will resolve to the name of an SCL variable.
- ④ SYMGET pulls the value from the macro variable and loads it into an SCL variable of the same name. In both instances, &&VAR&I resolves to the name of a variable.

When the compile process starts to compile the SCL that contains the macro call, the macro executes. First the macro generates SCL code, and then that code is compiled.

The following discussion assumes that this call to %DATASTMP is used:

```
%datastmp(subject,ptid)
```

This call to %DATASTMP has two specified positional parameters (SUBJECT and PTID). The %DO loop in the INIT section will execute twice. The INIT section that follows is generated when the SCL that contains the call to %DATASTMP (as previously shown) is compiled (after the macro executes):

```
init:
  if scrnype='DE' or word(1)='ADD' then do;
    unprotect subject; ③ ⑤
    subject = symget("subject"); ④ ⑥
    protect subject; ⑦
    unprotect ptid;
    ptid = symget("ptid");
    protect ptid;
  end;
return;
```

- ⑤ When this INIT section executes, the screen variable SUBJECT will be unprotected.
- ⑥ A value for the screen variable is retrieved from a macro variable of the same name.
- ⑦ The screen variable is re-protected.

The %DATASTMP macro has been used to generate SCL code that has SCL variables with the same name as the macro variable values.

As a general rule, you should have a compelling reason to generate SCL code this way, especially for FSEDIT screens. The problem is that the SCL source code is not integral to the SCREEN. If you lose the macro source code, you will be unable to regenerate the SCL code. From a practical point of view, the process itself can be cumbersome. If you need to change the SCL code you will need to:

- make the change in the macro source code.
- if running interactively, make sure that the current version of WORK.SASMACR contains the correct version of the macro. Either delete the catalog entry (using %SYSMACDELETE), or recompile the macro.
- start FSEDIT and use modify.
- compile the SCL. If there are compile errors, go back to the start of this list.

**SEE ALSO:** Bryher (1997b) uses a macro to build SCL code.

## A2.4 Using the Macro Language with FCMP Functions

Functions and routines defined through the use of PROC FCMP are compiled, and as such are subject to many of the same limitations as other compiled entities, like SCL programs, within SAS. The general concepts for the use of the macro language within compiled SCL programs also apply to the use of macro language elements within an FCMP function.

- Macro language elements are executed during the compilation time of the function (see Section A2.1)
- %LET statements cannot be used to create macro variables during the execution of the function (see Section A2.2.1)
- Macro variable resolution cannot be triggered with an ampersand (see Section A2.2.3) during the execution of the function
- Macro calls triggered by a percent sign cannot be executed during the function execution (see Section A2.3)

In SCL a macro can be executed from within a SUBMIT block (see Section A2.2.2); however, although there is no SUBMIT block within a PROC FCMP step, it is still possible to execute a macro during the execution of the function by using the RUN\_MACRO routine (see Section A2.4.2).

### A2.4.1 Compile-Time Execution

Macro language elements are executed during the compilation phase of the function (See Section A2.3.2). As such these elements would be used to create the FCMP step code. This technique is even less commonly used with FCMP than it is with SCL. Primarily this is because FCMP functions tend to be shorter, more generalized, and more static. If you do want to use the macro language to help with the writing of your FCMP step code, the techniques described in Section A2.3.2 should give you the necessary help.

**SEE ALSO:** Duling (2015) uses a macro call to generate FCMP code.

### A2.4.2 Executing a Macro during Function Execution

Macro language elements can be used with compiled functions; however, like with other compiled elements (views, compiled DATA steps, SCL programs, and such), one must be careful to understand the interaction of the macro language with the compilation process. For PROC FCMP you need to specifically use the RUN\_MACRO routine to execute a macro during function execution.

For the purposes of this discussion we want to execute the macro PRINTIT from within an FCMP subroutine. Macro PRINTIT is a simple PROC PRINT with some TITLE statements, and is defined in Program A2.4.2a.

#### Program A2.4.2a: Simple Macro to Print a Selected Number of Observations

```
%macro printit(lib, dsn, num);
  %if &num= %then %let num=max;
  title2 "&lib..&dsn";
  title3 "First &num Observations";
  proc print data=&lib..&dsn(obs=&num);
    run;
%mend printit;
```

We would like to pass the subroutine PRINTN three arguments and then in turn pass those arguments as parameters to the macro %PRINTIT. Unfortunately THIS DOES NOT WORK!! We must remember that the function is first compiled and later (perhaps even next week) it is executed. A macro call such as the one shown in Program A2.4.2b is executed during the compilation of the function and NOT during its execution.

**Program A2.4.2b: Incorrect Usage of a Macro Call in an FCMP routine**

```
proc fcmp outlib=macro3.functions.utilities;
  subroutine printN(lib $, dsn $,num);
    * This will NOT work! The macro is
    * executed when the function is compiled;
    %printit(lib,dsn,num)
  endsub;
  run;
```

It is conceivable that you actually would want to execute a macro during the compilation of the function. Remember the macro language is primarily a code generator. So if your macro is used to write the code used by the function, then a macro call within the function definition could be appropriate (see Section A2.3.2).

When you want a function or subroutine to execute a macro when the function executes, you will need to use the special RUN\_MACRO routine. This special routine allows you to name a macro to be executed and to list its parameters. The RUN\_MACRO routine and the way it interfaces with the macro does have some limitations, but they are not too severe. First the character parameter values passed into the macro from the function will be surrounded by quotes, which will almost certainly need to be removed. Secondly the macro itself is defined without parameters – as these are supplied by the function. As a result macros written to be called by FCMP functions will have limited utility elsewhere.

Here the PRINTN subroutine has been rewritten to use the RUN\_MACRO routine. The macro name is quoted, as it is a constant in this example. And the macro name is followed by the parameter values.

**Program A2.4.2c: Using RUN\_MACRO**

```
proc fcmp outlib=macro3.functions.utilities;
  subroutine printN(lib $, dsn $,num);
    rc=run_macro('printit',lib,dsn,num);
  endsub;
  run;
```

Macros called through the use of the RUN\_MACRO routine will tend to be coded with specific requirements of the RUN\_MACRO routine in mind. In Program A2.4.2d the %PRINTIT macro (shown in Program A2.4.2a) has been adapted for use with RUN\_MACRO.

**Program A2.4.2d: %PRINTIT Adapted for Use with the RUN\_MACRO Routine**

```
%macro printit(); ①
  %put Executing PRINTIT for &lib &dsn &num; ②
  %let lib = %sysfunc(dequote(&lib)); ③
  %let dsn = %sysfunc(dequote(&dsn));
  %if &num= %then %let num=max; ④
  title2 "&lib..&dsn"; ⑤
  title3 "First &num Observations";
  proc print data=&lib..&dsn(obs=&num);
    run;
%mend printit;
```

- ① The macro itself is defined without parameters. The names of the arguments in the subroutine are the same as the parameters in the macro, and they are passed into the macro directly. Inclusion of the parentheses is optional.
- ② This %PUT statement is included here merely to demonstrate that the values of character parameters are surrounded by quotes.
- ③ Because the process of calling a macro using RUN\_MACRO inserts quotes around character parameter values, these quote marks will need to be removed. Here the DEQUOTE function is used to remove the quotes from the parameters.

- ❸ This check should not be needed as the RUN\_MACRO routine does not accept null parameter values.
- ❹ The names of the macro parameters are the same as the names of the variables in the RUN\_MACRO call.

We can show that the macro is called through a DATA step call to the PRINTN routine:

```
data _null_;  
put '***** Before routine call';  
call printn('macro3', 'clinics', 3); ❺  
put '***** After routine call';  
run;
```

The SAS Log shows the following:

```
***** Before routine call  
Executing PRINTIT for 'macro3' 'clinics' 3 ❻  
***** After routine call  
NOTE: DATA statement used (Total process time):
```

**SEE ALSO:** Carpenter (2013) introduces the FCMP procedure and includes a short discussion of the RUN\_MACRO routine, and most of that discussion is repeated in Section A2.4 (with the author's permission).

## **Appendix 3: Utilities and Examples Locator**

<b>Data Set / File Manipulation .....</b>	<b>461</b>
<b>Data Variable Manipulation .....</b>	<b>461</b>
<b>Data Value Manipulation.....</b>	<b>461</b>
<b>Date / Time .....</b>	<b>462</b>
<b>Library / Directory Tools .....</b>	<b>462</b>
<b>Macro Techniques .....</b>	<b>462</b>
<b>Macro Variable Tools.....</b>	<b>462</b>
<b>SAS Execution .....</b>	<b>462</b>
<b>SAS/GRAFH Tools.....</b>	<b>462</b>
<b>System and Environment.....</b>	<b>463</b>
<b>Text Manipulation.....</b>	<b>463</b>

This appendix is designed to assist you with locating examples of certain types of macro utilities in this book. Obviously, this static list is a brief and incomplete compilation of the many fine utilities developed by various users of the SAS macro language. For sources other than those in this book, consult this book's bibliography.

---

### **Data Set / File Manipulation**

Appending unknown data sets	12.1.2
Copy an unknown number of catalogs	12.1.1
Counting observations	11.2.6a
Creating empty data sets	11.7.4
Creating data subsets	6.4.1b, 11.27a, 11.2.7b, 12.4.1, 13.3.1, 13.3.2, 13.3.3, 13.3.4, A1.6.2, A1.6.3
Does the data set exist?	7.1.3a, 7.5.1, 7.5.2
Dumping or listing flat files	13.1.1a, 13.1.1b
Implementing field checks	11.7.5

---

### **Data Variable Manipulation**

Changing character variables to numeric	A1.6.4
Collecting variable names	11.9.1c, 12.4.2, A1.6.4
Does the variable exist?	12.4.6
Random number generation	7.4.1b, 13.3.3
Variable list (building)	12.4.2

---

### **Data Value Manipulation**

Factorial / combinatorial calculation	7.6.1b, 7.6.1c, 7.6.2a, 7.6.2b
Number rounding	7.4.3

---

## Date / Time

Consolidating dates	7.6.3b
Loading dates and times into macro variables	7.3.3c, 7.4.2b, 13.1.2
Generating date text strings	7.4.2, 7.6.1d
Incrementing dates	7.6.3a
Working with two-digit years	A1.7.7

---

## Library / Directory Tools

Deleting files	7.4.2d, 7.4.2e
Establishing and checking directories	8.2.4, 13.2.2
Establishing and checking libraries	7.4.3a, 12.3.3

---

## Macro Techniques

Commenting blocks of code	3.1.2
Comments within a macro	14.3.6, Program 14.3.6e
Debugging	8.3.2, 14.3.4
Doubly subscripted arrays	14.2
Dynamic macro programming	Chapters 11 & 12
Hashing	9.3
Macro libraries	10.6.9

---

## Macro Variable Tools

Copy macro source code	10.3.3c
Deleting macro variables	8.2.1
Determine macro variable scope	8.1.5, 8.5.3
Determine the word number	12.6.2c
Does the macro variable exist?	8.1.5, 9.2.1b
Last word in a list	7.6.1k, 7.6.1l, 12.4.3a
Remove duplicate words from a list	12.4.7
Search for a word in a list	8.3.3c

---

## SAS Execution

Batch file creation and execution	13.1.3
Compiled programs	Appendix 2
Executable file location (SAS.EXE)	8.1.1
Product availability check	8.1.4
Remote server execution	9.7
Returning the path of the executing program	8.1.1c, 8.1.1d
Session initialization and termination	14.6
Storing and retrieving system options	12.3.1

---

## SAS/GRAFTH Tools

Generating gray-scale PATTERN statements	7.4.3b
Create RGB color specifications	10.6.8

---

## System and Environment

Accessing system options	11.2.4
Checking for Write access (copy success)	8.3.2
Control of titles and footnotes	12.2.1a, 12.2.1b
Creating and working with directories	12.1.2f
Detecting operation errors	8.3.2
Establishing directories	7.4.2d, 8.2.4
Executing batch SAS programs	13.1.3
Format maintenance	12.3.2
ODS style display	12.2.2
Option control	8.1.1a
Output hyperlink control	13.2.3
Reading system variables	8.1.1
Suspending SAS operations	7.6.1g, 7.6.1h, 7.6.1i, 7.6.1j
System initialization (AUTOEXEC)	14.6

---

## Text Manipulation

Adding commas between words	12.4.4
Adding quotation marks to words	12.4.5
Counting words in a list	7.3.2d, 8.3.3, 11.4.4a, 11.4.4b, A1.7.4
Compare strings of unequal length	7.6.2e
Removing duplicate words from a list	12.4.7
Repeat a list of characters	7.6.2d
Retrieving the last word in a list	7.6.1k, 7.6.1l, 12.4.3a
Return a word number from a list	7.6.2c
Return the word number for the last occurrence	A1.7.6

## **Appendix 4: Code Sample Locator**

<b>A4.1 Macro Variable Constructs .....</b>	<b>465</b>
<b>A4.2 Macro Language Statements, Functions, and Autocall Macros .....</b>	<b>466</b>
<b>A4.3 %MACRO Statement Options .....</b>	<b>469</b>
<b>A4.4 Automatic Macro Variables .....</b>	<b>469</b>
<b>A4.5 DATA Step and Other Non-Macro-Language Elements .....</b>	<b>470</b>
<b>A4.6 SASHELP Views and DICTIONARY Tables .....</b>	<b>473</b>

Very often we learn best through the use of examples; however, it is not always easy to find an example of a specific language element. This appendix is designed to assist you with locating examples of certain topics or statements. Each table shows an alphabetical listing of SAS language elements, the sections in this book where they are introduced or discussed, and a list of additional sections that contain examples that pertain to that element.

---

### **A4.1 Macro Variable Constructs**

The use of the &VAR construct is so ubiquitous in this book that it is not worthy of mentioning here; however, less common macro variable references are also discussed throughout the book. The primary alternate forms of macro variable references can be found in Table A4.1.

**Table A4.1: Macro Variable Constructs**

<b>Macro Variable Construct</b>	<b>Initial or Primary Discussion</b>	<b>Used in these Sections and Programs</b>
&&&var	6.4.2	11.1.1c, 11.2.4, 14.1.2, 14.2.3b
&&var&i	2.5.3, 11.3	6.1.4, 6.4.1, 9.1.1b, 11.1.1, 11.2.2b, 11.2.7a, 11.7.4, 11.7.5, 12.1.1, 12.1.2, 12.2.2, 12.4.3c, 12.4.8a, 13.2.3, A1.6.3, A1.6.4, A2.2.5, A2.3.2, and others throughout the book
&&&var&i	14.2.3	

## A4.2 Macro Language Statements, Functions, and Autocall Macros

Macro language elements include statements, functions, and autocall macros that behave like functions. Table A4.2 lists those elements described in this book. Automatic macro variables are listed in Table A4.3.

**Table A4.2 Macro Language Elements**

Macro Language Element	Initial or Primary Discussion	Used in these Sections and Programs
%* (macro comment) and comments within macros	5.4.1	11.7.5, 12.4.1, A1.7.7, and throughout the book
%ABORT	5.4.4	8.3.2a
%BQUOTE	7.1.1	7.1.3, 7.1.4, 7.2.8, 7.6.1e, 8.3.2b, 9.8.2, 9.8.4, 12.1.2e, 14.3.5
%NRBQUOTE		
%CMPRES %QCMPRES	10.6.3	12.3.1, 12.4.4b, 12.4.5b
%COMPSTOR	10.6.7	
%COPY	10.3.3	
%DATATYP	10.6.6	
%DISPLAY	8.2.3	
%DO (block)	5.3.1	7.1.3, 11.1.5, 11.2.6a
%DO (iterative)	5.3.2	6.1.4, 6.4.1, 8.1.2, 8.3.3c, 9.1.1b, 10.6.4, 10.6.9, 11.1.1a, 11.1.1d, 11.1.8, 11.2.2b, 11.2.7a, 11.2.7b, 11.4.2, 11.7.4, 11.7.5, 12.1.1, 12.1.2a, 12.1.2b, 12.2.2, 12.4.8a, 13.1.1a, 13.2.3, 14.1.2, 14.2.2, 14.2.3a, 14.2.3b, 14.2.4, A1.5.2, A1.5.10, A1.6.2, A1.6.3, A1.6.4, A1.7.7, A2.3.2
%DO %UNTIL	5.3.3	7.2.3, A1.5.2
%DO %WHILE	5.3.4	7.6.2c, 8.3.3d, 14.3.5, 14.3.6, 10.6.3, 11.4.4a, 12.4.4a, 12.4.5a, 12.4.6, 12.4.7, 14.3.5, A1.5.2, A1.7.4, A1.7.6
%END	5.3	In each example with a %DO
%EVAL	7.3.1, 7.3.2	5.2.3, 5.3.3, 5.3.4, 7.2.3, 7.6.1f, 7.6.2c, 8.1.5, 8.3.3c, 8.3.3d, 11.2.1, 12.2.1a, 12.2.1b, 12.4.1, 12.4.3c, 12.4.4a, 12.4.5a, 12.4.7, 14.3.5, 14.4, A1.7.4, A1.7.6
%GLOBAL	5.4.2, 8.2.6	7.5.1, 8.5.3, 11.9.3, 13.1.2, 14.1.2

<b>Macro Language Element</b>	<b>Initial or Primary Discussion</b>	<b>Used in these Sections and Programs</b>
%GOTO %label	8.2.2	10.6.5, 11.2.4
%IF - %THEN / %ELSE	5.2	5.3.1, 7.1.3, 7.2.4, 7.2.5, 7.5.1, 7.6.2d, 8.3.2b, 8.3.2c, 8.3.3d, 10.3.3c, 11.2.4, 11.2.6a, 11.4.3, 11.7.5, 12.2.1a, 12.3.2, 13.2.2b, 13.3.3c, 13.3.4a, A1.5.7
%INDEX	7.2.1	10.6.3, 10.6.4, 12.3.2
%LEFT %QLEFT	7.2.6, 10.6.2	7.1.4, 7.6.1d, 9.8.1, 9.8.2, 9.8.3, 9.8.4, 10.6.3, 13.1.2
%LENGTH	7.2.2	8.3.3c, 9.5, 10.6.2, 10.6.4, 10.6.5, 11.2.4, 11.4.4b, 12.3.2
%LET	2.2, 2.3	Throughout the remainder of the book
%LOCAL	5.4.2, 8.2.6, 11.7.3	6.1.4, 7.4.2, 8.1.2, 8.1.5, 8.3.3c, 8.3.3d, 8.5.3, 9.2.1b, 9.2.2d, 10.6.5, 10.6.9, 11.1.1d, 11.2.1, 11.2.7a, 11.2.7b, 11.4.3, 11.4.4a, 11.7.5, 12.1.2b, 12.1.2f, 12.2.1a, 12.4.1, 12.4.3c, 12.4.4a, 12.4.5a, 12.4.6, 12.4.7, 13.1.1a, 13.3.3b, 13.3.4a, 14.2.3a, 14.2.3b, 14.2.4, A1.7.4, A1.7.6, A1.7.7
%LOWCASE %QLOWCASE	7.2.7, 10.6.4	13.1.2
%MACRO	3.1	Chapter 4 and throughout the book
%MEND		
%PUT	2.4	2.6.2, 6.1, 6.2.2, 6.5.3, 7.1.4, 7.1.6, 7.2.3, 7.2.4, 7.2.7, 7.2.8, 7.4.2, 8.1.3, 10.3.3c, 11.7.3, 13.3.3b, 14.2.2
%QUOTE %NRQUOTE	7.1.5	7.1.9
%RETURN	5.4.5	8.2.2e, 12.4.8b
%SCAN %QSCAN	7.2.3, 11.4.3	7.6.1k, 7.6.11, 7.6.2c, 8.3.3c, 8.3.3d, 10.6.9, 11.1.1d, 11.4.2, 11.4.3, 12.1.2f, 12.4.3a, 12.4.3c, 12.4.4a, 12.4.5a, 12.4.7, 14.3.5, A1.7.4, A1.7.5, A1.7.6

<b>Macro Language Element</b>	<b>Initial or Primary Discussion</b>	<b>Used in these Sections and Programs</b>
%STR	7.1.2	7.1.3, 7.1.9, 7.6.2b, 7.6.2c, 7.6.2e, 9.5,
%NRSTR		9.8.1, 9.8.2, 9.8.3, 9.8.4, 10.6.2,
		10.6.5, 10.6.9, 11.1.1d, 11.4.3,
		12.4.5a, A1.7.4
%SUBSTR	7.2.4	7.6.2e, 8.3.3c, 10.6.1, 10.6.2, 10.6.3,
%QSUBSTR		10.6.4, 10.6.5, 11.2.4, 12.1.2f, 12.3.2
%SUPERQ	7.1.7	9.5, 9.8.3, 9.8.4
%SYMDEL	2.7	8.2.1
%SYMEXIST	8.1.5	9.2.1a
%SYMGLOBL	8.1.5	9.2.1a
%SYMLOCAL	8.1.5	9.2.1a
%SYSCALL	7.4.1	
%QSYSCALL		
%SYSEVALF	7.3.3	7.1.7, 7.4.3b, 7.6.1j, 7.6.2b, 9.5, 12.4.1
%SYSEXEC	5.4.3, 8.2.4	10.3.3c, 13.2.2a
%SYSFUNC	7.4.2	7.5.2a, 7.5.2c, 7.6.1, 7.6.2, 8.3.3c,
%QSYSFUNC		9.2.1b, 10.6.9, 11.1.1d, 11.2.4,
		11.2.6a, 11.4.4b, 12.1.2d, 12.1.2f,
		12.3.2, 12.3.3, 12.4.1, 12.4.2d,
		12.4.5b, 12.4.6, 12.4.7, 13.1.2,
		13.2.2a, 13.2.2b, A1.7.7
%SYSGET	8.1.1	13.1.3
%SYSLPUT	9.7	
%SYSMACDELETE	8.2.5	10.3.5
%SYSMACEEXEC	8.1.3	
%SYSMACEEXIST	8.1.3	
%SYSMEXECDEPTH	8.1.2	
%SYSMEXECNAME	8.1.2	
%SYSMSTORECLEAR	10.3.6	
%SYSPROD	8.1.4	
%SYSRPUT	9.7	
%TRIM	7.2.8, 10.6.5	7.6.1d, 10.6.3, 13.2.3
%QTRIM		

<b>Macro Language Element</b>	<b>Initial or Primary Discussion</b>	<b>Used in these Sections and Programs</b>
%TSLIT		12.1.2e, 13.2.3
%UNQUOTE	7.1.6	7.2.8, 7.6.1e, 11.4.3, 12.1.2e
%UPCASE %QUPCASE	7.2.5	5.2.3, 7.6.1i, 7.6.2c, 7.6.2e, 8.1.3, 8.3.2b, 10.3.3c, 11.4.3, 11.7.5, 12.1.2, 12.1.2f, 12.3.1, 14.1.2, A1.7.6
%VERIFY %KVERIFY	10.6.1	10.6.2
%WINDOW	8.2.3	

---

### A4.3 %MACRO Statement Options

The options shown in Table A4.3 can appear on the %MACRO statement.

**Table A4.3: %MACRO Statement Options**

<b>%MACRO Statement Option</b>	<b>Initial or Primary Discussion</b>	<b>Used in these Sections and Programs</b>
/DES	3.1.3	
/PARMBUFF /PBUFF	8.3.3	
/SECURE	10.3.4	
/SOURCE	10.3.3	
/STORE	10.3.2	10.3.3, 10.3.4, 10.3.6

---

### A4.4 Automatic Macro Variables

Automatic macro variables are created and generally maintained by the macro facility. The primary ones discussed in this book can be found in Table A4.4. Initial discussion of this topic can be found in Section 2.6 and additional general discussion in Section 8.3.

**Table A4.4: Automatic Macro Variables**

<b>Automatic Macro Variable</b>	<b>Initial or Primary Discussion</b>	<b>Used in these Sections and Programs</b>
&SQLOBS	6.2.2b	Throughout the book.
&SQLRC	8.3.2e	
&SYSBUFFER		7.1.7
&SYSSCC	2.6.3, 8.3.2	
&SYSDATE	2.6.1	7.4.2a, 7.4.2b, 7.4.2c

Automatic Macro Variable	Initial or Primary Discussion	Used in these Sections and Programs
&SYSDATE9		
&SYSDAY	2.6.1	
&SYSDSN	2.6.2	
&SYSERR	2.6.3, 8.3.2	
&SYSERRORTEXT	8.3.2	
&SYSFILRC	8.3.6	
&SYSINFO	8.3.2f	
&SYSJOBID		13.1.1a
&SYSLAST	2.6.2	A2.2.3
&SYSLIBRC	8.3.6	
&SYSMACRONAME	2.6.6, 8.3.5	
&SYSNOBS	8.3.4	9.2.2c
&SYSPARM	8.3.1	
&SYSPBUFF	8.3.3	
%SYSPROD	8.1.4	
&SYSRC	2.6.4	
&SYSSCP &SYSSCPL	2.6.5	
&SYSSITE	2.6.5	
&SYSTIME	2.6.1	
&SYSUSERID	2.6.5	
&SYSWARNINGTEXT	8.3.2	

---

## A4.5 DATA Step and Other Non-Macro-Language Elements

The macro language has the capability to take advantage of DATA step functions, routines, Base SAS statements, and other non-macro language elements. The primary elements that are discussed or used in this book are highlighted in Table A4.5, the majority of the functions highlighted in this table have been used with %SYSFUNC or %QSYSFUNC.

**Table A4.5: DATA Step and Other Non-Macro-Language Elements**

<b>DATA Step and Non-macro Language Element</b>	<b>Initial or Primary Discussion</b>	<b>Used in these Sections and Programs</b>
ATTRN function	9.2.2d	11.2.6a, 12.4.1, 12.4.2d
CLOSE function	9.2.1b	9.2.2d, 11.2.6a, 12.4.2d, 12.4.6
COMB function	7.6.1c	
COMPRESS function	11.2.7b	
COUNTW function	8.3.3	11.1.1d, 11.4.4b, 12.4.3b
DATE function	7.4.2c	
DATETIME function	7.6.1j	
DCLOSE function	12.1.2f	
DCREATE function	13.2.2b	
DNUM function	12.1.2f	
DOSUBL function	8.5.1	
DOPEN function	12.1.2f	
DREAD function	12.1.2f	
EXECUTE routine	6.5, 11.5	A1.6.5
EXIST function	7.5.2	7.5.3, 7.6.1a
FACT function	7.6.1b	
FCMP procedure	A2.4.1	
FDELETE function	7.4.2d	7.4.2e
FETCH function	9.2.1b	
FETCHOBS function	9.2.2d	
FEXIST function	7.4.2e	
FILEEXIST function	7.4.2d	8.2.4, 10.3.3c, 13.2.2a, 13.2.2b
FILENAME function	7.4.2d	7.4.2e, 12.1.2f
GETOPTION function	11.2.4	10.3.3c , 12.3.2
INDEXW function	7.6.1m	12.4.7
INTNX function	7.6.3	
LIBNAME function	8.3.2d	12.3.3, A2.2.4

<b>DATA Step and Non-macro Language Element</b>	<b>Initial or Primary Discussion</b>	<b>Used in these Sections and Programs</b>
LIBREF function	12.3.3	
MIN function	7.6.2e	12.4.1
MOD	A1.7.7	
OPEN function	9.2.1b, 11.2.6	9.2.2d, 11.2.6a, 12.4.1, 12.4.2d, 12.4.6, A2.2.3
PATHNAME function	7.4.3a	10.6.7, 12.1.2d, 12.1.2e, 12.1.2f, 12.3.3
PERM function	7.6.2a	7.6.2b
PUTN function	7.4.2b	7.4.3b, 7.6.1n
RANUNI function	7.4.1b	13.3.3a
REPEAT function	7.6.2d	
RESOLVE function	6.3.3	6.3.4, 9.6
REVERSE function	7.6.1k	
ROUND function	7.4.3	7.6.1n
RSUBMIT statement	9.7	
RUN_MACRO	A2.2.4	
SET function	9.2.2d	
SLEEP function	7.6.1g	7.6.1h, 7.6.1i, 7.6.1j
SQL INTO clause	6.2, 11.3.1, 11.4.1	9.1.1b, 10.4.2d, 11.1.2a, 11.2.1, 11.2.2, 11.4.1, 12.1.1, 11.2.2b, 11.2.7a, 11.4.2, 11.7.3, 11.7.4, 12.1.1, 12.2.1a, 12.2.2, 12.4.2c, 12.4.8a, 13.3.1, 14.2.3b, 14.2.4, A1.6.2, A1.6.3
SYMDEL routine	8.5.2	
SYMEXIST function	8.5.3	9.3a
SYMGET function	6.3.2	6.3.3, 6.3.4, 6.3.5, 6.4.2c, 14.3.5, A2.2.3, A2.2.4, A2.3.2
SYMGETN function	6.3.2, A2.2.3	A2.2.5
SYMGLOBAL function	8.5.3	
SYMLOCAL function	8.5.3	

<b>DATA Step and Non-macro Language Element</b>	<b>Initial or Primary Discussion</b>	<b>Used in these Sections and Programs</b>
SYMPUT routine	6.1.2	7.2.4, 7.2.6, 7.2.8
SYMPUTN routine	A2.2.1	
SYMPUTX routine	6.1, 14.5.4	6.3.3, 6.3.5, 6.4.1, 6.4.2, 6.5.3, 7.1.3, 7.1.7, 7.2.6, 9.2.2b, 9.2.2c, 9.3b, 11.1.2a, 11.2.6b, 11.2.8a, 11.9.1c, 12.2.1b, 12.4.2b, 13.2.3, 14.2.2, 14.6.2, A1.6.4, A2.2.1, A2.2.4
SYSMSG function	8.3.2d	11.2.6a, 12.3.3
SYSTEM routine	7.4.1a	
TIME function		13.1.2
TODAY function		13.1.2
TRANWRD function		12.4.5b
VARNAME function	12.4.1	12.4.2d
VARNUM function	12.4.6	
X statement		12.1.2d

---

## A4.6 SASHELP Views and DICTIONARY Tables

SASHELP views and DICTIONARY tables provide an excellent source of information for the macro programmer. Table A4.6 shows where these tables are discussed and used within this book.

**Table A4.6: Using SASHELP Views and DICTIONARY Tables**

<b>Tables and Views</b>	<b>Initial or Primary Discussion</b>	<b>Used in these Sections and Programs</b>
DICTIONARY tables	11.2.2	12.1.2b, 12.2.1a, 12.4.2c
SASHELP views	11.2.1	6.2.2, 6.6.1, 6.6.5, 8.1.1a, 8.2.1e, 8.3.2f, 9.2.1b, 12.1.1, 12.1.2, 12.2.1b, 12.2.2, 12.3.1, 12.4.2a, 14.6.2

## Appendix 5: Glossary

### **autocall macro library**

a facility that enables the user to create a library of macro definitions that can be automatically recalled when they are to be executed.

### **automatic macro variables**

a special-purpose *macro variable* that is automatically defined and provided by the SAS System. The variable name should be considered to be reserved (see Section 2.6).

### **dynamic code**

SAS code generated during program execution by *macro statements* that are often dependent on either the data being processed or on a control file (see Sections 1.2.3, 6.4, and Chapter 11).

### **explicit specification**

when you override default behaviors or options or when you specifically call functions, it is said to be an *explicit specification*. See also *implicit specification*. In the following %SCAN function the second argument contains an **explicit** %EVAL even though it would be implied if it were not actually specified:

```
%scan(&list,%eval(&i+1),%str(-))
```

### **global macro variable**

a global macro variable has a value that is available to all macros within the current session or program. *Macro variables* that are defined outside of any macro will be global (see Section 1.5). See also *local macro variable, referencing environment (scope)*.

### **implicit specification**

when an operation is implied by your code without actually being specified, it is said to be an implicit specification (see also *explicit specification*). One of the most common implicit specifications is with the %EVAL function (see Section 7.3.2). In the following %SCAN function there is an implied %EVAL in the second argument:

```
%scan(&list,&i+1,%str(-))
```

### **literal text**

text enclosed in single or double quotation marks.

### **local macro variable**

a local macro variable has a value that is available only within the context of the macro in which it is defined. Nested symbol tables may contain multiple (or nested) local definitions of a macro variable at one time (see Section 1.5). See also *global macro variable, referencing environment (scope)*.

### **macro**

stored text that contains SAS statements and macro language statements (see Chapter 3).

### **macro execution**

macro definitions are compiled and stored before they can be used or executed. Macros are called or executed by preceding their name with a percent sign (%) (see Chapter 3).

### **macro expression**

one or more *macro variable* names, *text*, or *macro functions* that are combined together by the use of one or more operators or parentheses or both. *Macro expressions* are very analogous to the expressions used in standard SAS programming (see Section 5.2).

**macro facility**

the tool within Base SAS software that contains the essential elements that enable you to use macros. It is the overall collection of SAS software processing utilities that control the processing of the *macro language* (see Sections 1.1 and 1.4).

**macro function**

predefined routines for processing text in macros and in macro variables. Many *macro functions* are similar to functions that are used in the DATA step (see Chapter 7).

**macro language**

the elements and tools that provide you with the means to communicate with the *macro processor* and the *macro facility* (see Section 1.1).

**macro processor**

software within the *macro facility* that translates macro code into both statements and *text* that can be used by SAS (see Section 1.1).

**macro program statement**

a statement that controls what actions take place during the macro execution. It is always preceded by a percent sign (%) and is often syntactically similar to statements used in the DATA step (see Tables 5.4a and 5.4b).

**macro reference**

*text* that results in a call to a macro language element is said to contain a *macro language reference*. This call will contain macro facility *triggers* that are generally one or both of the special symbols: ampersand (&) and percent sign (%) (see Section 1.4).

**macro system options**

system options that directly deal with the way SAS deals with macro code and how the results of macro processing are displayed (see Sections 3.3 and 8.4).

**macro tokens**

see *tokens*.

**macro triggers**

see *triggers*.

**macro variables**

macro, or symbolic, variables are often used to store *text*. The value of a macro variable is stored in a *symbol table*, and when used, the names of macro variables are almost always preceded by an ampersand (&) (see Chapter 2).

**open code**

SAS program statements that exist outside of any macro definition.

**operators**

symbols that are used for comparisons, logical operation, or arithmetic calculations. The operators are primarily the same ones used in the DATA step (see Section 5.2).

**persistence**

related to *scope*. The duration of a definition or value.

**referencing environment (scope)**

each *macro variable*'s definition in the *symbol table* is also associated with a referencing environment or scope that is determined by where and how the macro variable is defined. Scopes can be either *global* or *local* (Section 1.4). Local environments can be nested and might have multiple layers (see Section 5.4.2). See also *global macro variable*, *local macro variable*.

**resolving macro references**

when elements of the macro language (or *macro references*) are replaced with *text* during the resolution process (see Section 2.5).

**scope**

see referencing environment (scope).

**symbol table**

storage location in memory that is used to store current values of *macro variables* (see Section 1.5). Symbol tables can be either global or local in scope (see *referencing environment (scope)*).

**symbolic variables**

see macro variables.

**text**

a collection of characters and symbols that can form such things as variable names, data set names, SAS statement fragments, complete SAS statements, complete DATA and PROC steps, or even complete SAS programs (see Section 3.1).

**tokens**

*tokens* are created by the *word scanner*, and are the basic component parts of a SAS statement. There are several types of these tokens, but the two that have special meaning to the *macro language* are those that start with either the percent sign (%) or the ampersand (&). These two symbols are macro processor *triggers* (see Section 1.4).

**triggers**

triggers are symbols that have special meaning to the SAS processor. The two macro language triggers are the percent sign (%) and the ampersand (&). When the *word scanner* detects one of these macro triggers (followed by a letter or underscore), the *macro processor* is invoked and the statement or *token* is turned over to the *macro facility* for processing (see Section 1.4).

**word scanner**

when SAS statements are submitted for processing, they are broken up into their component parts (*tokens*) by the *word scanner* (see Section 1.4).

## Bibliography

Most of the following references were cited within the text of this book. All, even those not directly cited, contain information that may be of interest to you.

- “Questions and Answers.” *SAS Communications*, (1st quarter 1997): 48.
- “Technical Support.” *SAS Communications*, vol. 22, no. 4 (4th quarter 1996): 43.
- Abbott, David H. 2015. “Rapidly Assessing Data Completeness.” *Proceedings of the Twenty-Third Annual SouthEast SAS Users Group Conference*. Cary, NC: SAS Institute Inc. Available [http://www.lexjansen.com/sesug/2015/130\\_Final\\_PDF.pdf](http://www.lexjansen.com/sesug/2015/130_Final_PDF.pdf).
- Abdurazak, Tugluke. 2002. “Using SAS Macros to Create Automated Excel Reports Containing Tables, Charts and Graphs.” *Proceedings of the Twenty-Seventh Annual SAS Users Group International Conference*. Cary, NC: SAS Institute Inc. Available <http://www2.sas.com/proceedings/sugi27/p126-27.pdf>.
- Aboutaleb, Hany. 1997a. “Printing with SAS 6.10 and 6.11.” *Proceedings of the Pharmaceutical Industry SAS Users Group Conference* 279–282. Cary, NC: SAS Institute Inc.
- Aboutaleb, Hany. 1997b. “More about Missing Character Data.” *Proceedings of the Pharmaceutical Industry SAS Users Group Conference* 283–284. Cary, NC: SAS Institute Inc.
- Adams, John H. 2003. “The Power of Recursive SAS Macros: How Can a Simple Macro Do So Much?” *Proceedings of the Twenty-Eighth Annual SAS Users Group International Conference*. Cary, NC: SAS Institute Inc. Available <http://www2.sas.com/proceedings/sugi28/087-28.pdf>.
- Agbenyegah, Delali. 2015. “Tell Me What You Want: Conjoint Analysis Made Simple Using SAS®.” *Proceedings of the SAS Global Forum 2015 Conference*. Cary, NC: SAS Institute Inc. Available <http://support.sas.com/resources/papers/proceedings15/3042-2015.pdf>.
- Ake, Christopher F., and Arthur L. Carpenter. 2002. “Survival Analysis with PHREG: Using MI and MIANALYZE to Accommodate Missing Data.” *Proceedings of the Tenth Annual Western Users of SAS Software Conference* 102–107. Cary, NC: SAS Institute Inc. Available [http://www.wuss.org/proceedings10/analy/3067\\_3\\_ANL-Carpenter.pdf](http://www.wuss.org/proceedings10/analy/3067_3_ANL-Carpenter.pdf).
- Ake, Christopher F., and Arthur L. Carpenter. 2003. “Extending the Use of PROC PHREG in Survival Analysis.” *Proceedings of the Eleventh Annual Western Users of SAS Software Conference*. Cary, NC: SAS Institute Inc. Available [http://www.lexjansen.com/wuss/2003/DataAnalysis/i-extending\\_phreg.pdf](http://www.lexjansen.com/wuss/2003/DataAnalysis/i-extending_phreg.pdf).
- Alden, Kay. 2000. “SAS’ Best Kept Secret: Macro Windows for Applications Development.” *Proceedings of the Twenty-Fifth Annual SAS Users Group International Conference* 465–468. Cary, NC: SAS Institute Inc. Available <http://www2.sas.com/proceedings/sugi25/25/cc/25p076.pdf>.
- Allen, Richard R. 2002. “Recovering Lost Lines: Using the Whole Page with Proc Report.” *Proceedings of the Annual Pharmaceutical SAS Users Group Conference* 97–102. Cary, NC: SAS Institute Inc. Available <http://www.lexjansen.com/pharmasug/2002/proceed/Coders/cc08.pdf>.
- Allen, Rick. 2003. “An Automated MS PowerPoint Presentation Using SAS.” *Proceedings of the Twenty-Eighth Annual SAS Users Group International Conference*. Cary, NC: SAS Institute Inc. Available <http://www2.sas.com/proceedings/sugi28/092-28.pdf>.
- Andresen, Robert. 1997. “Macro and Sample Source Code to Wrap Character Variable Text Conditionally on Two Lines within DATA \_Null\_ -generated Report Column(s).” *Proceedings of the Twenty-Second Annual SAS Users Group International Conference* 453–455. Cary, NC: SAS Institute Inc. Available <http://www2.sas.com/proceedings/sugi22/CODERS/PAPER83.PDF>.
- Benjamin, William E., Jr. 1999. “A Pseudo-Recursive SAS Macro.” *Observations: The Technical Journal for SAS Software Users*. Cary, NC: SAS Institute Inc. Available <ftp://ftp.sas.com/techsup/download/observations/obswww18/obswww18.pdf>.

- Bennett, Aileen D. 2001. "Side by Side: Comparing Two Data Sets Using SAS Macros." *Proceedings of the Twenty-Sixth Annual SAS Users Group International Conference*. Cary, NC: SAS Institute Inc. Available <http://www2.sas.com/proceedings/sugi26/p097-26.pdf>.
- Bercov, Mark. 1993. "SAS Macros – What's Really Happening?": *Proceedings of the Eighteenth Annual SAS Users Group International Conference* 440–444. Cary, NC: SAS Institute Inc. Republished in *SAS Macro Facility Tips and Techniques, Version 6* (Cary, NC: SAS Institute Inc., 1994), 17–21. Available <http://www.sascommunity.org/sugi/SUGI93/Sugi-93-70%20Bercov.pdf>.
- Bessler, LeRoy, and Francesca Pierri. 2002. "%TREND: A Macro to Produce Maximally Informative Trend Charts with SAS/GRAFH, SAS, and ODS for the Web or Hardcopy." *Proceedings of the Twenty-Seventh Annual SAS Users Group International Conference*. Cary, NC: SAS Institute Inc. Available <http://www2.sas.com/proceedings/sugi27/p093-27.pdf>.
- Bessler, LeRoy. 2003. "Web Communication Effectiveness: Design and Methods to Get the Best Out of ODS, SAS, and SAS/GRAFH." *Proceedings of the Twenty-Eighth Annual SAS Users Group International Conference*. Cary, NC: SAS Institute Inc. Available <http://www2.sas.com/proceedings/sugi28/130-28.pdf>.
- Beverly, Bryan K. 1999. "'There's Nothing %To It But %To %Do It' – How To Do A Lot With Little Coding." *Proceedings of the Twelfth Annual NorthEast SAS Users Group Conference* 477–478. Cary, NC: SAS Institute Inc. Available <http://www.lexjansen.com/nesug/nesug99/po/po093.pdf>.
- Beverly, Bryan K. 2001. "'Note It and Float It': How to Use Automatic Macro Variables and E-mail Services for Error Trapping and End User Notification." *Proceedings of the Twelfth Annual MidWest SAS Users Group Conference* 253–257. Cary, NC: SAS Institute Inc. Available <http://www.lexjansen.com/mwsug/2001/Posters/POS-037-noteitandf.pdf>.
- Billings, Thomas E. 2015. "Strategies for Error Handling and Program Control: Concepts." *Proceedings of the SAS Global Forum 2015 Conference*. Cary, NC: SAS Institute Inc. Available <http://support.sas.com/resources/papers/proceedings15/1565-2015.pdf>.
- Blair, Kimberly S. 1998. "Using Macros to Produce Multiple Time Series Graphs." *Proceedings of the Twenty-Third Annual SAS Users Group International Conference* 935–939. Cary, NC: SAS Institute Inc. Available <http://www2.sas.com/proceedings/sugi23/Posters/p168.pdf>.
- Blood, Nancy K. 1992. "Using SAS Macros to Generate Code (So Your Program Can Figure Out What to Do for Itself)." *Proceedings of the Seventeenth Annual SAS Users Group International Conference* 24–28. Cary, NC: SAS Institute Inc. Republished in *SAS Macro Facility Tips and Techniques, Version 6* (Cary, NC: SAS Institute Inc., 1994), 138–142. Available <http://www.sascommunity.org/sugi/SUGI92/Sugi-92-06%20Blood.pdf>.
- Borgerding, Joleen, and Akiko Chai. 2000. "Utilizing SAS Functions to Access Variable Information." *Proceedings of the Eighth Annual Western Users of SAS Software Conference* 65–66. Cary, NC: SAS Institute Inc. Available <http://www.lexjansen.com/wuss/2000/WUSS00014.pdf>.
- Bramley, Michael P. D. 2001. "Combining Pattern-Matching and File I/O Functions: A SAS Macro to Generate a Unique Variable Name List." *Proceedings of the Twelfth Annual MidWest SAS Users Group Conference* 32–35. Cary, NC: SAS Institute Inc. Also *Proceedings of the Twenty-Seventh Annual SAS Users Group International Conference*. Cary, NC: SAS Institute Inc. Available <http://www2.sas.com/proceedings/sugi27/p037-27.pdf>.
- Bucherker, M. Michelle. 1998a. "%FLATFILE, and Make Your Life Easier." *Proceedings of the Eleventh Annual NorthEast SAS Users Group Conference* 645–652. Cary, NC: SAS Institute Inc. Also *Proceedings of the Sixth Annual Western Users of SAS Software Conference* (1998) 24–26. Cary, NC: SAS Institute Inc. Available <ftp://ftp.sas.com/techsup/download/misc/flatfile.pdf>.
- Bucherker, M. Michelle. 1998b. "Dictionary Tables." *Proceedings of the Sixth Annual Western Users of SAS Software Conference* 103–104. Cary, NC: SAS Institute Inc. Available <http://www.lexjansen.com/wuss/1998/WUSS98021.pdf>.
- Bryant, Connie. 1997. "Automated Generation of a SAS Macro Cross-Reference Table." *Proceedings of the Twenty-Second Annual SAS Users Group International Conference* 928–933. Cary, NC: SAS Institute Inc. Available <http://www2.sas.com/proceedings/sugi22/POSTERS/PAPER200.PDF>.

- Bryher, Monique. 1997a. "How Symbolic Variables Can Reduce Code in a Graphics Environment." *Proceedings of the Twenty-Second Annual SAS Users Group International Conference* 37–42. Cary, NC: SAS Institute Inc. Available <http://www2.sas.com/proceedings/sugi22/APPDEVEL/PAPER8.PDF>.
- Bryher, Monique. 1997b. "Building a Simple SAS Macro to Generate SQL Instructions in Frequently Used DB2 Tables." *Proceedings of the Twenty-Second Annual SAS Users Group International Conference* 43–47. Cary, NC: SAS Institute Inc. Available <http://www2.sas.com/proceedings/sugi22/APPDEVEL/PAPER9.PDF>.
- Burlew, Michele M. 2014. *SAS Macro Programming Made Easy, Third Edition*. Cary, NC: SAS Institute Inc. Available [https://www.sas.com/store/books/categories/getting-started/sas-macro-programming-made-easy-third-edition/prodBK\\_66298\\_en.html](https://www.sas.com/store/books/categories/getting-started/sas-macro-programming-made-easy-third-edition/prodBK_66298_en.html)
- Burnett-Isaacs, Kate. 2016a. "No More Bad Dates!: A Guide to SAS® Dates in Macro Language." *Proceedings of the SAS Global Forum 2016 Conference*. Cary, NC: SAS Institute Inc. Available <http://support.sas.com/resources/papers/proceedings16/7260-2016.pdf>.
- Burnett-Isaacs, Kate. 2016b. "How to Create Data-Driven Lists." *Proceedings of the SAS Global Forum 2016 Conference*. Cary, NC: SAS Institute Inc. Available <http://support.sas.com/resources/papers/proceedings16/9540-2016.pdf>.
- Burroughs, Scott. 2001. "A Few Ways to Use Macro Variables, Comparison Operators, and 'other=' , Format Statements to Shorten Your Programs." *Proceedings of the Fourteenth Annual NorthEast SAS Users Group Conference* 307–308. Cary, NC: SAS Institute Inc. Available <http://www.lexjansen.com/nesug01/cc/cc4002.pdf>.
- Carey, Helen, and Ginger Carey. 1996. *SAS Today! A Year of Terrific Tips*. Cary, NC: SAS Institute Inc.
- Carpenter, Arthur L. 1994. "Playing with Macros: Take the Work out of Learning to Do Macros." *Proceedings of the Nineteenth Annual SAS Users Group International Conference* 368–372. Cary, NC: SAS Institute Inc. Available <http://www.sascommunity.org/sugi/SUGI94/Sugi-94-69%20Carpenter.pdf>.
- Carpenter, Arthur L. 1996. "Programming for Job Security: Tips and Techniques to Maximize Your Indispensability." *Proceedings of the Twenty-first Annual SAS Users Group International Conference* 1637–1640. Cary, NC: SAS Institute Inc. Available <http://www.sascommunity.org/sugi/SUGI96/Sugi-96-262%20Carpenter.pdf>.
- Carpenter, Arthur L. 1997. "Resolving and Using &&var&I Macro Variables." *Proceedings of the Twenty-Second Annual SAS Users Group International Conference* 437–440. Cary, NC: SAS Institute Inc. Available <http://www2.sas.com/proceedings/sugi22/CODERS/PAPER77.PDF>.
- Carpenter, Arthur L. 1998. "Advanced Macro Topics: Utilities and Examples." *Proceedings of the Twenty-Third Annual SAS Users Group International Conference* 287–292. Cary, NC: SAS Institute Inc. Also published in *Proceedings of the Fifth Annual Western Users of SAS Software Conference* (1997) 494–499. Cary, NC: SAS Institute Inc. Available <http://www2.sas.com/proceedings/sugi23/Advtutor/p49.pdf>.
- Carpenter, Arthur L. 1999a. "Macro Quoting Functions, Other Special Character Masking Tools, and How to Use Them." *Proceedings of the Twenty-Fourth Annual SAS Users Group International Conference* 237–241. Cary, NC: SAS Institute Inc. Available <http://www2.sas.com/proceedings/sugi24/Advtutor/p38-24.pdf>. Also published in *Proceedings of the Twelfth Annual NorthEast SAS Users Group Conference* (1999) 3–7. Cary, NC: SAS Institute Inc. Available <http://www.lexjansen.com/nesug/nesug99/ad/ad088.pdf>. Also published in *Proceedings of the Sixth Annual Western Users of SAS Software Conference* (1998) 453–457. Cary, NC: SAS Institute Inc. Available [http://www.sascommunity.org/wiki/Macro\\_Quoting\\_Functions,\\_Other\\_Special\\_Character\\_Masking\\_Tools,\\_and\\_How\\_to\\_Use\\_Them](http://www.sascommunity.org/wiki/Macro_Quoting_Functions,_Other_Special_Character_Masking_Tools,_and_How_to_Use_Them).
- Carpenter, Arthur L. 1999b. "Using ANNOTATE MACROS as Shortcuts." *Proceedings of the Twelfth Annual NorthEast SAS Users Group Conference* 367–372. Cary, NC: SAS Institute Inc. Also published in *Proceedings of the Seventh Annual Western Users of SAS Software Conference* (1999) 453–458. Cary, NC: SAS Institute Inc. Available <http://www2.sas.com/proceedings/sugi24/Infovis/p168-24.pdf>.

- Carpenter, Arthur L. 2000a. "Using Macro Functions." *Proceedings of the Twenty-Fifth Annual SAS Users Group International Conference* 24–30. Cary, NC: SAS Institute Inc. Also published in *Proceedings of the Eighth Annual Western Users of SAS Software Conference* (2000) 521–527. Cary, NC: SAS Institute Inc. Available  
<http://www2.sas.com/proceedings/sugi25/25/aa/25p004.pdf>
- Carpenter, Arthur L. 2000b. "Placing Dates in Your Titles: Do It Dynamically." *Proceedings of the Twenty-Fifth Annual SAS Users Group International Conference* 521–523. Cary, NC: SAS Institute Inc. Also published in *Proceedings of the Seventh Annual Western Users of SAS Software Conference* (1999) 76–78. Cary, NC: SAS Institute Inc. Available  
<http://www2.sas.com/proceedings/sugi25/25/cc/25p093.pdf>
- Carpenter, Arthur L. 2001. "Building and Using Macro Libraries." *Proceedings of the Twelfth Annual MidWest SAS Users Group Conference* 151–157. Cary, NC: SAS Institute Inc. Also published in *Proceedings of the Ninth Annual Western Users of SAS Software Conference* (2001) 605–610. Cary, NC: SAS Institute Inc. Also published in *Proceedings of the Annual Pharmaceutical SAS Users Group Conference* (2002) 191–198. Cary, NC: SAS Institute Inc. Available  
<http://www2.sas.com/proceedings/sugi27/p017-27.pdf>
- Carpenter, Arthur L. 2002. "Macro Functions: How to Make Them—How to Use Them." *Proceedings of the Twenty-Seventh Annual SAS Users Group International Conference*. Cary, NC: SAS Institute Inc. Also published in *Proceedings of the Annual Pharmaceutical SAS Users Group Conference* (2002) 87–91. Cary, NC: SAS Institute Inc. Available  
<http://www2.sas.com/proceedings/sugi27/p100-27.pdf>
- Carpenter, Arthur L. 2003. "Creating Display Manager Abbreviations and Keyboard Macros for the Enhanced Editor." *Proceedings of the Twenty-Eighth Annual SAS Users Group International Conference*. Cary, NC: SAS Institute Inc. Also published in *Proceedings of the Annual Pharmaceutical SAS Users Group Conference* (2003) 127–130. Cary, NC: SAS Institute Inc. Available  
<http://www2.sas.com/proceedings/sugi28/108-28.pdf>
- Carpenter, Arthur L. 2004. "Five Ways to Create Macro Variables: A Short Introduction to the Macro Language." *Proceedings of the Twelfth Annual Western Users of SAS Software Conference (WUSS)*. Cary, NC: SAS Institute Inc. Also published in *Proceedings of the Annual Pharmaceutical SAS Users Group Conference (PharmaSUG)* 2004. Cary, NC: SAS Institute Inc. Also published in *Proceedings of the Thirteenth Annual SouthEast SAS Users Group Conference (SESUG)* 2005. Cary, NC: SAS Institute Inc. Also published in *Proceedings of the MidWest SAS User Group Conference (MWSUG)* 2005. Cary, NC: SAS Institute Inc. Also presented at WUSS (2009) and PNWSUG (2009). Available  
[http://www.sascommunity.org/wiki/Five\\_Ways\\_to\\_Create\\_Macro\\_Variables:\\_A\\_Short\\_Introduction\\_to\\_the\\_Macro\\_Language](http://www.sascommunity.org/wiki/Five_Ways_to_Create_Macro_Variables:_A_Short_Introduction_to_the_Macro_Language)
- Carpenter, Arthur L. 2005. "Make 'em %LOCAL: Avoiding Macro Variable Collisions." *Proceedings of the Annual Pharmaceutical SAS Users Group Conference (PharmaSUG)* 2005. Cary, NC: SAS Institute Inc. Also published in *Proceedings of the Thirteenth Annual Western Users of SAS Software Conference (WUSS)* 2005. Cary, NC: SAS Institute Inc. Available  
[http://www.calaxy.com/papers/62\\_TT04.pdf](http://www.calaxy.com/papers/62_TT04.pdf)
- Carpenter, Arthur L. 2007. "Advanced PROC REPORT: Getting Your Tables Connected Using Links." *Proceedings of the Annual Pharmaceutical SAS Users Group Conference (PharmaSUG)* 2007. Cary, NC: SAS Institute Inc. Also presented in 2007 at the Fifteenth Annual Western Users of SAS Software Conference (WUSS), in 2008 at MWSUG, in 2009 at WUSS, SESUG, SCSUG, SAS Global Forum, and PNWSUG. Available  
<http://support.sas.com/resources/papers/proceedings09/026-2009.pdf>
- Carpenter, Arthur L. 2008. "The Path, The Whole Path, And Nothing But the Path, So Help Me Windows." *Proceedings of the SAS Global Forum 2008 Conference*. Cary, NC: SAS Institute Inc. Available  
<http://www2.sas.com/proceedings/forum2008/023-2008.pdf>
- Carpenter, Arthur L. 2009. "Manual to Automatic: Changing Your Program's Transmission." *Proceedings of the Seventeenth Annual Western Users of SAS Software Conference* 2009. Also published in *Proceedings of the Annual Pharmaceutical SAS Users Group Conference (PharmaSUG)* 2010. Cary, NC: SAS Institute Inc. Also published in *Proceedings of the SAS Global Forum 2010*

- Conference*. Cary, NC: SAS Institute Inc. Available  
[http://www.sas.com/content/dam/SAS/en\\_ca/User%20Group%20Presentations/Vancouver-User-Group/ArthurCarpenter-ManualtoAuto-2010.pdf](http://www.sas.com/content/dam/SAS/en_ca/User%20Group%20Presentations/Vancouver-User-Group/ArthurCarpenter-ManualtoAuto-2010.pdf).
- Carpenter, Arthur L. 2012. *Carpenter's Guide to Innovative SAS® Techniques*. Cary, NC: SAS Institute Inc. Available  
[http://www.sascommunity.org/wiki/Category:Carpenter%20%99s\\_Guide\\_to\\_Innovative\\_SA\\_S\\_Techniques](http://www.sascommunity.org/wiki/Category:Carpenter%20%99s_Guide_to_Innovative_SA_S_Techniques).
- Carpenter, Arthur L. 2013. "Using PROC FCMP to the Fullest: Getting Started and Doing More." *Proceedings of the SAS Global Forum 2013 Conference*. Cary, NC: SAS Institute Inc. Available  
<http://support.sas.com/resources/papers/proceedings13/139-2013.pdf>.
- Carpenter, Arthur L. 2014a. "Quotes within Quotes: When Single ('') and Double ("") Quotes are not Enough." *Proceedings of the Annual Pharmaceutical SAS Users Group Conference (PharmaSUG) 2014*. Cary, NC: SAS Institute Inc. Also published in *Proceedings of the SAS Global Forum 2015 Conference*. Cary, NC: SAS Institute Inc. Available  
<http://support.sas.com/resources/papers/proceedings15/2221-2015.pdf>.
- Carpenter, Arthur L. 2014b. "Before You Get Started: A Macro Language Preview in Three Parts." *Proceedings of the SAS Global Forum 2014 Conference*. Cary, NC: SAS Institute Inc. Available  
<http://support.sas.com/resources/papers/proceedings14/1444-2014.pdf>.
- Carpenter, Arthur L. 2014c. "Are You Missing Out? Working with Missing Values to Make the Most of What is not There." *Proceedings of the Annual Pharmaceutical SAS Users Group Conference*. Cary, NC: SAS Institute Inc. Available  
<http://www.pharmasug.org/proceedings/2014/TT/PharmaSUG-2014-TT02.pdf>.
- Carpenter, Arthur L., and Janice D. Callahan. 1988. "Subsetting Data into Groups for Complete Processing within Each Group." *Proceedings of the Thirteenth Annual SAS Users Group International Conference*. Cary, NC: SAS Institute Inc. Available  
<http://www.sascommunity.org/sugi/SUGI88/Sugi-13-166%20Carpenter%20Callahan.pdf>.
- Carpenter, Arthur L., and Richard O. Smith. 2000. "Clinical Data Management: Building a Dynamic Application." *Proceedings of the Annual Pharmaceutical SAS Users Group Conference* 151–156. Cary, NC: SAS Institute Inc. Also published in *Proceedings of the Eighth Annual Western Users of SAS Software Conference* (2000) 3–8. Cary, NC: SAS Institute Inc. Available  
[http://www.sascommunity.org/wiki/Clinical\\_Data\\_Management:\\_Building\\_a\\_Dynamic\\_Application](http://www.sascommunity.org/wiki/Clinical_Data_Management:_Building_a_Dynamic_Application).
- Carpenter, Arthur L., and Richard O. Smith. 2001. "Library and File Management: Building a Dynamic Application." *Proceedings of the Annual Pharmaceutical SAS Users Group Conference* 125–132. Cary, NC: SAS Institute Inc. Also published in *Proceedings of the Ninth Annual Western Users of SAS Software Conference* (2001) 266–272. Cary, NC: SAS Institute Inc. Also published in *Proceedings of the Twenty-Seventh Annual SAS Users Group International Conference* (2002). Cary, NC: SAS Institute Inc. Available <http://www2.sas.com/proceedings/sugi27/p021-27.pdf>.
- Carpenter, Arthur L., and Richard O. Smith. 2002. "ODS and Web Enabled Device Drivers: Displaying and Controlling Large Numbers of Graphs." *Proceedings of the Tenth Annual Western Users of SAS Software Conference* 182–189. Cary, NC: SAS Institute Inc. Also published in *Proceedings of the Annual Pharmaceutical SAS Users Group Conference* (2003). Cary, NC: SAS Institute Inc. Available  
[http://www.sascommunity.org/wiki/ODS\\_and\\_Web\\_Enabled\\_Device\\_Drivers:\\_Displaying\\_and\\_Controlling\\_Large\\_Numbers\\_of\\_Graphs](http://www.sascommunity.org/wiki/ODS_and_Web_Enabled_Device_Drivers:_Displaying_and_Controlling_Large_Numbers_of_Graphs).
- Carpenter, Arthur L., and Tony Payne. 1998. "Programming for Job Security Revisited: Even More Tips and Techniques to Maximize Your Indispensability." *Proceedings of the Twenty-Third Annual SAS Users Group International Conference* 1547–1556. Cary, NC: SAS Institute Inc. Available  
<http://www2.sas.com/proceedings/sugi23/Training/p275.pdf>.
- Carpenter, Arthur L., and Tony Payne. 2001. "A Bit More on Job Security: Long Names and Other V8 Tips." *Proceedings of the Twenty-Sixth Annual SAS Users Group International Conference*. Cary, NC: SAS Institute Inc. Available <http://www2.sas.com/proceedings/sugi26/p235-26.pdf>.

- Casas, Angelina Cecilia. 2002. "A Data Dictionary, A More Complete Alternative to Proc Contents." *Proceedings of the Annual Pharmaceutical SAS Users Group Conference* 79–82. Cary, NC: SAS Institute Inc. Available <http://www.lexjansen.com/pharmasug/2002/proceed/Coders/cc04.pdf>.
- Chakravarthy, Venky. 2003. "You Be Your Own QC Cop: Check Your .SAS, .LOG and .LST Dates from within SAS Using Operating System Commands." *Proceedings of the Annual Pharmaceutical SAS Users Group Conference*. Cary, NC: SAS Institute Inc. Available <http://www.lexjansen.com/pharmasug/2003/CodersCorner/cc099.pdf>.
- Chan, Vincent, and Lorena Ortiz. 2016. "Building Macros for Quick Survey Scoring." *Proceedings of the SAS Global Forum 2016 Conference*. Cary, NC: SAS Institute Inc. Available <http://support.sas.com/resources/papers/proceedings16/11521-2016.pdf>.
- Chapman, David D. 2001. "Selecting Unrestricted and Simple Random with Replacement Samples Using Base SAS and PROC SURVEYSELECT." *Proceedings of the Fourteenth Annual NorthEast SAS Users Group Conference* 749–757. Cary, NC: SAS Institute Inc. Also published in *Proceedings of the SAS Global Forum 2012 Conference*. Cary, NC: SAS Institute Inc. Available <http://support.sas.com/resources/papers/proceedings12/347-2012.pdf>.
- Chen, Chang-Min. 2003. "Reading Data from a Large Number of External Files: How to Do It Efficiently?" *Proceedings of the Annual Pharmaceutical SAS Users Group Conference*. Cary, NC: SAS Institute Inc. Available <http://www.lexjansen.com/pharmasug/2003/CodersCorner/cc010.pdf>.
- Chen, Ling Y., and Steven A. Gilbert. 2002. "Run All Your SAS Programs in One Program: Automatically." *Proceedings of the Twenty-Seventh Annual SAS Users Group International Conference*. Cary, NC: SAS Institute Inc. Available <http://www2.sas.com/proceedings/sugi27/p105-27.pdf>.
- Chen, X. Hong. 2001. "%ToC: A Macro for Generating Table of Contents from SAS Output." *Proceedings of the Twenty-Sixth Annual SAS Users Group International Conference*. Cary, NC: SAS Institute Inc. Available <http://www2.sas.com/proceedings/sugi26/p094-26.pdf>.
- Cheng, Alice. 2015. "How to Speed Up Your Validation Process Without Really Trying?" *Proceedings of the Twenty-Third Western Users of SAS Software Conference*. Cary, NC: SAS Institute Inc. Available [http://www.lexjansen.com/wuss/2015/86\\_Final\\_Paper\\_PDF.pdf](http://www.lexjansen.com/wuss/2015/86_Final_Paper_PDF.pdf).
- Cheng, Alice M. 1999. "Robust Programming Techniques in the SAS System." *Proceedings of the Twelfth Annual NorthEast SAS Users Group Conference* 494–503. Cary, NC: SAS Institute Inc. Available <http://www.lexjansen.com/nesug99/po/po159.pdf>.
- Cheng, Alice M., Michael R. Wise, and Justina M. Flavin. 2016. "How to Speed Up Your Validation Process Without Really Trying?" *Proceedings of the SAS Global Forum 2016 Conference*. Cary, NC: SAS Institute Inc. Available <http://support.sas.com/resources/papers/proceedings16/10840-2016.pdf>.
- Cheng, Wei. 2000. "RPT\_UTIL: A SAS Macro for Enhancing Reports in Clinical Trials." *Proceedings of the Annual Pharmaceutical SAS Users Group Conference* 59–64. Cary, NC: SAS Institute Inc. Available <http://www.lexjansen.com/pharmasug/2000/Coders/cc03.pdf>.
- Chow, Ming H. 1999. "SAS MACRO: As a Dynamic Code Generator." *Proceedings of the Twelfth Annual NorthEast SAS Users Group Conference* 504–507. Cary, NC: SAS Institute Inc. Available <http://www.lexjansen.com/nesug99/po/po176.pdf>.
- Chung, Chang Y., and John King. 2009. "IS THIS MACRO PARAMETER BLANK?" *Proceedings of the SAS Global Forum 2009 Conference*. Cary, NC: SAS Institute Inc. Available <http://support.sas.com/resources/papers/proceedings09/022-2009.pdf>.
- Cohen, Barry R. 1998. "Supporting the 'Program-Analyze-Write-Review' Process with a Development Environment for Base SAS and the Macro Language." *Proceedings of the Annual Pharmaceutical SAS Users Group Conference* 63–68. Cary, NC: SAS Institute Inc. Available <http://www2.sas.com/proceedings/sugi23/Appdevl/p28.pdf>.
- Cohen, John J. 1998. "A Tutorial on the SAS Macro Language." *Proceedings of the Eleventh Annual NorthEast SAS Users Group Conference* 221–226. Cary, NC: SAS Institute Inc. Also presented at the SAS Global Forum 2012 Conference. Available <http://support.sas.com/resources/papers/proceedings12/249-2012.pdf>.

- Conley, Brian. 2000. "Saving Time with Variable Labels." *Proceedings of the Thirteenth Annual NorthEast SAS Users Group Conference* 259–260. Cary, NC: SAS Institute Inc. Available <http://www.lexjansen.com/nesug/nesug00/cc/cc4001.pdf>.
- Crawford, Peter. 2016. "More Hidden Base SAS® Features to Impress Your Colleagues." *Proceedings of the SAS Global Forum 2016 Conference*. Cary, NC: SAS Institute Inc. Available <http://support.sas.com/resources/papers/proceedings16/2120-2016.pdf>.
- Croonen, Nancy, and Henri Theuwissen. 2002. "Table Lookup: Techniques Beyond the Obvious." *Proceedings of the Twenty-Seventh Annual SAS Users Group International Conference*. Cary, NC: SAS Institute Inc. Available <http://www2.sas.com/proceedings/sugi27/p011-27.pdf>.
- Culp, Jennifer C. 1991. "Using SAS Macros with SAS/AF and SAS/FSP Software in Developing User-Friendly Applications." *Proceedings of the Sixteenth Annual SAS Users Group International Conference* 196–203. Cary, NC: SAS Institute Inc. Republished in *SAS Macro Facility Tips and Techniques, Version 6* (Cary, NC: SAS Institute Inc., 1994), 93–100. Available <http://www.sascommunity.org/sugi/SUGI91/Sugi-91-37%20Culp.pdf>.
- Davis, Michael. 2001. "You Could Look It Up: An Introduction to SASHELP Dictionary Views." *Proceedings of the Twenty-Sixth Annual SAS Users Group International Conference*. Cary, NC: SAS Institute Inc. Also published in *Proceedings of the Thirteenth Annual NorthEast SAS Users Group Conference* (2000) 177–186. Cary, NC: SAS Institute Inc. Available <http://www.ats.ucla.edu/stat/sas/library/nesug00/bt3004.pdf>.
- Davis, Neil. 1997. "Rapid Applications Development Using the SAS System." *Proceedings of the Twenty-Second Annual SAS Users Group International Conference* 18–24. Cary, NC: SAS Institute Inc. Available <http://www2.sas.com/proceedings/sugi22/APPDEVEL/PAPERS5.PDF>.
- DiIorio, Frank C. 1999. "Removing Macro Variables from the SAS Environment." *Proceedings of the Seventh Annual SouthEast SAS Users Group Conference* 214–216. Cary, NC: SAS Institute Inc. Available <http://analytics.ncsu.edu/sesug/1999/109.pdf>.
- diTommaso, Dante. 2003. "Taking Control of Macro Variables." *Proceedings of the Twenty-Eighth Annual SAS Users Group International Conference*. Cary, NC: SAS Institute Inc. Available <http://www2.sas.com/proceedings/sugi28/095-28.pdf>.
- Dorfman, Paul M. 2001. "QuickSorting an Array." *Proceedings of the Twenty-Sixth Annual SAS Users Group International Conference*. Cary, NC: SAS Institute Inc. Available <http://www2.sas.com/proceedings/sugi26/p096-26.pdf>.
- Duling, David R. 2015. "Make Better Decisions with Optimization." *Proceedings of the SAS Global Forum 2015 Conference*. Cary, NC: SAS Institute Inc. Available <http://support.sas.com/resources/papers/proceedings15/SAS1785-2015.pdf>.
- Dynder, Andrea, Barry R. Cohen, and Gary Cunningham. 2000. "A SAS Macro for Creating a Standard Directory Structure." *Proceedings of the Annual Pharmaceutical SAS Users Group Conference* 3–8. Cary, NC: SAS Institute Inc. Available <http://www.lexjansen.com/pharmasug/2000/AppDev/ad01.pdf>.
- Eberhardt, Peter. 2016. "Solving the 1,001-Piece Puzzle in 10 (or Fewer) Easy Steps: Using SASv9.cfg, autoexec.sas, SAS Registry, and Options to Set Up Base SAS®." *Proceedings of the SAS Global Forum 2016 Conference*. Cary, NC: SAS Institute Inc. Available <http://support.sas.com/resources/papers/proceedings16/11700-2016.pdf>.
- Eddlestone, Mary-Elizabeth. 1997. "Getting the Most out of "INTO" in PROC SQL: An Example for Creating Macro Variables." *Proceedings of the Tenth Annual NorthEast SAS Users Group Conference* 307–308. Cary, NC: SAS Institute Inc. Available <http://www.lexjansen.com/nesug/nesug97/coders/eddlest.pdf>.
- Edgington, Jim, and Jay Zhou. 2002. "Post-processing MPRINT Outputs to Generate Macro-Free Code." *Proceedings of the Annual Pharmaceutical SAS Users Group Conference* 477–482. Cary, NC: SAS Institute Inc. Available <http://www.lexjansen.com/pharmasug/2002/proceed/TechTech/tt14.pdf>.
- Ewing, Daphne. 2002. "Macros: Data Listings with Power." *Proceedings of the Twenty-Seventh Annual SAS Users Group International Conference*. Cary, NC: SAS Institute Inc. Available <http://www2.sas.com/proceedings/sugi27/p185-27.pdf>.

- Fahmy, Adel. 2003. "Duplicates: Any Database, Any Client, Any Time." *Proceedings of the Annual Pharmaceutical SAS Users Group Conference* 189-194. Cary, NC: SAS Institute Inc. Available <http://www.lexjansen.com/pharmasug/2003/DataManagement/dm008.pdf>.
- Fehd, Ronald. 1997a. "%ARRAY: Construction and Usage of Arrays of Macro Variables." *Proceedings of the Twenty-Second Annual SAS Users Group International Conference* 447–450. Cary, NC: SAS Institute Inc. Also published in *Proceedings of the Twenty-Ninth Annual SAS Users Group International Conference*. Cary, NC: SAS Institute Inc. Available <http://www2.sas.com/proceedings/sugi29/070-29.pdf>.
- Fehd, Ronald. 1997b. "%SHOWCOMB: A Macro to Produce a Data Set with Frequency of Combinations of Responses from Multiple-Response Data." *Proceedings of the Twenty-Second Annual SAS Users Group International Conference* 939–943. Cary, NC: SAS Institute Inc. Available <http://www2.sas.com/proceedings/sugi22/POSTERS/PAPER204.PDF>.
- Fehd, Ronald. 1997c. "%CHECKALL: A Macro to Produce a Frequency of Response Data Set from Multiple-Response Data." *Proceedings of the Twenty-Second Annual SAS Users Group International Conference* 1084–1088. Cary, NC: SAS Institute Inc. Available <http://www2.sas.com/proceedings/sugi22/POSTERS/PAPER236.PDF>.
- Fehd, Ronald. 2000. "The Writing for Reading SAS Style Sheet: Tricks, Traps & Tips from SAS-L's Macro Maven." *Proceedings of the Twenty-Fifth Annual SAS Users Group International Conference* 217–220. Cary, NC: SAS Institute Inc. Also published in *Proceedings of the SouthEast SAS Users Group Conference* (1999) 16-18. Cary, NC: SAS Institute Inc. Also published in *Proceedings of the Fourteenth Annual NorthEast SAS Users Group Conference* (2001) 133–136. Cary, NC: SAS Institute Inc. Available <http://www2.sas.com/proceedings/sugi25/25/ad/25p038.pdf>.
- Fehd, Ronald. 2001. "A Beginner's Tour of a Project Using SAS Macros Led by SAS-L's Macro Maven." *Proceedings of the Twenty-Sixth Annual SAS Users Group International Conference*. Also published in *Proceedings of the Fourteenth Annual NorthEast SAS Users Group Conference* (2001) 214–222. Cary, NC: SAS Institute Inc. Available <http://www2.sas.com/proceedings/sugi26/p066-26.pdf>.
- Fehd, Ronald J. 2014. "Macro Design Ideas: Theory, Template, Practice." *Proceedings of the SAS Global Forum 2014 Conference*. Cary, NC: SAS Institute Inc. Available <http://support.sas.com/resources/papers/proceedings14/1899-2014.pdf>.
- Fehd, Ronald J., and Art Carpenter. 2007. "List Processing Basics: Creating and Using Lists of Macro Variables." *Proceedings of the SAS Global Forum 2007 Conference*. Cary, NC: SAS Institute Inc. Available [http://www.sascommunity.org/wiki/List\\_Processing\\_Basics\\_Creating\\_and\\_Using\\_Lists\\_of\\_Macro\\_Variables](http://www.sascommunity.org/wiki/List_Processing_Basics_Creating_and_Using_Lists_of_Macro_Variables).
- Felty, Kelly, and Diane Nicholson. 1999. "The Power of PAGEOF (A Valuable Page Numbering Macro)." *Proceedings of the Twenty-Fourth Annual SAS Users Group International Conference* 543–546. Cary, NC: SAS Institute Inc. Available <http://www2.sas.com/proceedings/sugi24/Coders/p081-24.pdf>.
- Ferriola, Frank. 2016. "Avoid Change Control by Using Control Tables." *Proceedings of the SAS Global Forum 2016 Conference*. Cary, NC: SAS Institute Inc. Available <http://support.sas.com/resources/papers/proceedings16/10662-2016.pdf>.
- Ferriola, Frank, and Avery Long. 2015. "Standardizing the Standardization Process." *Proceedings of the SAS Global Forum 2015 Conference*. Cary, NC: SAS Institute Inc. Available <http://support.sas.com/resources/papers/proceedings15/3209-2015.pdf>.
- First, Steven. 2001a. "Advanced Macro Topics." *Proceedings of the Twenty-sixth Annual SAS Users Group International Conference*. Cary, NC: SAS Institute Inc. Also published in *Proceedings of the Twenty-Seventh Annual SAS Users Group International Conference* (2002). Cary, NC: SAS Institute Inc. Available <http://www2.sas.com/proceedings/sugi27/p020-27.pdf>.
- First, Steven. 2001b. "An Introduction to SAS Macros." *Proceedings of the Twenty-Sixth Annual SAS Users Group International Conference*. Cary, NC: SAS Institute Inc. Available <http://www2.sas.com/proceedings/sugi26/p153-26.pdf>.

- Flavin, Justina M., and Arthur L. Carpenter. 2000. "Taking Control and Keeping It: Creating and Using Conditionally Executable SAS Code." *Proceedings of the Twenty-Fifth Annual SAS Users Group International Conference* 517–520. Cary, NC: SAS Institute Inc. Also published in *Proceedings of the Twenty-Sixth Annual SAS Users Group International Conference* (2001). Cary, NC: SAS Institute Inc. Also published in *Proceedings of the Eighth Annual Western Users of SAS Software Conference* (2000) 81–84. Cary, NC: SAS Institute Inc. Available <http://www2.sas.com/proceedings/sugi26/p074-26.pdf>.
- Flavin, Justina M., and Arthur L. Carpenter. 2003. "Is the Legend in Your SAS/GGRAPH Output Telling the Right Story?" *Proceedings of the Eleventh Annual Western Users of SAS Software Conference* (2003). Cary, NC: SAS Institute Inc. Also published in *Proceedings of the Twenty-Ninth Annual SAS Users Group International Conference* (2004). Cary, NC: SAS Institute Inc. Also published in *Proceedings of the Annual Pharmaceutical SAS Users Group Conference* (2004). Cary, NC: SAS Institute Inc. Available <http://www2.sas.com/proceedings/sugi29/086-29.pdf>.
- Frankel, David S., and Mark A. Kochanski. 1991. "A Debugging Facility for SAS Macro Systems." *Proceedings of the Sixteenth Annual SAS Users Group International Conference* 209–214. Cary, NC: SAS Institute Inc. Republished in *SAS Macro Facility Tips and Techniques, Version 6* (Cary, NC: SAS Institute Inc., 1994), 53–58. Available <http://www.lexjansen.com/seasug/1993/SCSUG93009.pdf>.
- Friendly, Michael. 1991. *SAS System for Statistical Graphics*. Cary, NC: SAS Institute Inc. Available <http://www.datavis.ca/books/sssg/>.
- Galligan, Olena. 2011. "Has Anybody Opened My File? Find Out Before Trying to Update it." *Proceedings of the Nineteenth Annual Western Users of SAS Software Conference*. Cary, NC: SAS Institute Inc. Available [http://www.wuss.org/proceedings11/Papers\\_Galligan\\_O\\_74889.pdf](http://www.wuss.org/proceedings11/Papers_Galligan_O_74889.pdf).
- Gau, Linda C. 1999. "Using SAS Software Window to Dynamically Manage Your Routine Programs." *Proceedings of the Twenty-Fourth Annual SAS Users Group International Conference*, Miami Beach, FL 575–577. Cary, NC: SAS Institute Inc. Also published in *Proceedings of the Sixth Annual Western Users of SAS Software Conference* (1998) 114–116. Cary, NC: SAS Institute Inc. Available <http://www2.sas.com/proceedings/sugi24/Coders/p091-24.pdf>.
- Geary, Hugh. 1997. "A Macro Tool for Quickly Producing a Handy Documented Listing of SAS Data Sets for Use as a Reference While Writing Programs to Analyze the Same." *Proceedings of the Twenty-Second Annual SAS Users Group International Conference* 949–954. Cary, NC: SAS Institute Inc. Available <http://www2.sas.com/proceedings/sugi22/POSTERS/PAPER206.PDF>.
- Gerlach, John R. 1997. "The Six Ampersand Solution." *Proceedings of the Tenth Annual NorthEast SAS Users Group Conference* 629–630. Cary, NC: SAS Institute Inc. Available <http://www.lexjansen.com/nesug/nesug97/posters/gerlach.pdf>.
- Gerlach, John.R., and Simant Misra. 2002. "Splitting a Large SAS Data Set." *Proceedings of the Twenty-Seventh Annual SAS Users Group International Conference*. Cary, NC: SAS Institute Inc. Available <http://www2.sas.com/proceedings/sugi27/p083-27.pdf>.
- Gershelyn, Yefim. 2002. "Macro for Restoring SAS Transport Files." *Proceedings of the Twenty-Seventh Annual SAS Users Group International Conference*. Cary, NC: SAS Institute Inc. Available <http://www2.sas.com/proceedings/sugi27/p081-27.pdf>.
- Gilmore, Jodie, and Linda Helwig. 1990. "Debugging Your SAS Macro Application under VMS: A Practical Approach." *Proceedings of the Fifteenth Annual SAS Users Group International Conference* 825–831. Cary, NC: SAS Institute Inc. Republished in *SAS Macro Facility Tips and Techniques, Version 6* (Cary, NC: SAS Institute Inc., 1994), 66–72. Cary, NC: SAS Institute Inc. Available <http://www.sascommunity.org/sugi/SUGI90/Sugi-90-139%20Gilmore%20Helwig.pdf>.
- Glass, Roberta, and Louise Hadden. 2016. "Document and Enhance Your SAS® Code, Data Sets, and Catalogs with SAS Functions, Macros, and SAS Metadata." *Proceedings of the SAS Global Forum 2016 Conference*. Cary, NC: SAS Institute Inc. Available <http://support.sas.com/resources/papers/proceedings16/8300-2016.pdf>.
- Goddard, Jonathan R. 2003. "Efficient Reporting with Large Numbers of Variables: A SAS Method." *Proceedings of the Twenty-Eighth Annual SAS Users Group International Conference*. Cary, NC: SAS Institute Inc. Available <http://www2.sas.com/proceedings/sugi28/150-28.pdf>.

- Gondara, Lovedeep. 2014. "Generating Dynamic Tables Using PROC SQL and PROC TABULATE." *Proceedings of the SAS Global Forum 2014 Conference*. Cary, NC: SAS Institute Inc. Available <http://support.sas.com/resources/papers/proceedings14/1594-2014.pdf>.
- Graebner, Robert W. 1998. "Generating SAS Source Code with SAS Macros." *Proceedings of the Twenty-Third Annual SAS Users Group International Conference* 440–441. Cary, NC: SAS Institute Inc. Available <http://www2.sas.com/proceedings/sugi23/Coders/p81.pdf>.
- Graham, Brandon, and Scott Osowski. 2013. "Avoiding SAS Data Set Locks in a Windows Environment." *Proceedings of the Annual Pharmaceutical SAS Users Group Conference, PharmaSUG*. Cary, NC: SAS Institute Inc. Available <http://www.lexjansen.com/pharmasug/2013/BB/PharmaSUG-2013-BB13.pdf>.
- Grant, Paul. 1994. "The 'SKIP' Statement." *Proceedings of the Second Annual Western Users of SAS Software Conference* 87–88. Cary, NC: SAS Institute Inc. Also published in *Proceedings of the Twenty-Third Annual SAS Users Group International Conference* 426–427. Cary, NC: SAS Institute Inc. Also published in *Proceedings of the Eleventh Annual NorthEast SAS Users Group Conference* (1998) 335–336. Cary, NC: SAS Institute Inc. Available <http://www.ats.ucla.edu/stat/sas/library/nesug98/p142.pdf>.
- Gunshenan, Michael. 2003. "A SAS Macro for Simple Statistical Summary." *Proceedings of the Annual Pharmaceutical SAS Users Group Conference*. Cary, NC: SAS Institute Inc. Available <http://www.lexjansen.com/pharmasug/2003/Posters/P089.pdf>.
- Hadden, Louise. 2003. "What's in a Map? A Macro-Driven Drill-Down Geo-Graphical Representation System." *Proceedings of the Twenty-Eighth Annual SAS Users Group International Conference*. Cary, NC: SAS Institute Inc. Available <http://www2.sas.com/proceedings/sugi28/135-28.pdf>.
- Hadden, Louise. 2015. "Better Metadata Through SAS® II: %SYSFUNC, PROC DATASETS, and Dictionary Tables." *Proceedings of the SAS® Global Forum 2015 Conference*. Cary, NC: SAS Institute Inc. Cary, NC: SAS Institute Inc. Available <http://support.sas.com/resources/papers/proceedings15/3458-2015.pdf>.
- Hahl, Thomas J., and Philip R. Shelton. 1995. "Input/Output." *Observations: The Technical Journal for SAS Software Users*, vol. 4, no. 2 (1st quarter): 76–78. Cary, NC: SAS Institute Inc.
- Hahl, Thomas J., and Philip R. Shelton. 1996. "Dropping Variables That Have Only Missing Values." *Observations: The Technical Journal for SAS Software Users*, vol. 5, no. 4 (3rd quarter): 18–22. Cary, NC: SAS Institute Inc.
- Hamilton, Jack. 1998. "Some Utility Applications of the Dictionary Tables in PROC SQL." *Proceedings of the Sixth Annual Western Users of SAS Software Conference* 85–90. Cary, NC: SAS Institute Inc. Available [http://www.albany.edu/~msz03/epi697/papers/sql\\_dict.pdf](http://www.albany.edu/~msz03/epi697/papers/sql_dict.pdf).
- Hamilton, Jack. 2000. "Workarounds for SASWare Ballot Items." *Proceedings of the Twenty-Fifth Annual SAS Users Group International Conference* 1158–1162. Cary, NC: SAS Institute Inc. Available <http://www2.sas.com/proceedings/sugi25/25po/25p223.pdf>.
- Hamilton, Jack. 2001. "How Many Observations Are in My Data Set?" *Proceedings of the Twenty-Sixth Annual SAS Users Group International Conference*. Cary, NC: SAS Institute Inc. Available <http://www2.sas.com/proceedings/sugi26/p095-26.pdf>.
- Hayden, Vanessa. 2002. "Bulletproofing Your SAS Results." *Proceedings of the Twenty-Seventh Annual SAS Users Group International Conference*. Cary, NC: SAS Institute Inc. Available <http://www2.sas.com/proceedings/sugi27/p098-27.pdf>.
- Heaton, Edward. 2001. "Top-Down Programming with SAS Macros." *Proceedings of the Fourteenth Annual NorthEast SAS Users Group Conference* 67–75. Cary, NC: SAS Institute Inc. Also published in *Proceedings of the Fourteenth Annual SouthEast SAS Users Group Conference*. Cary, NC: SAS Institute Inc. Available [http://analytics.ncsu.edu/sesug/2006/AP04\\_06.PDF](http://analytics.ncsu.edu/sesug/2006/AP04_06.PDF).
- Heaton-Wright, Lawrence. 2003. "The SAS DATA Step/Macro Interface." *Proceedings of the Annual Pharmaceutical SAS Users Group Conference*. Cary, NC: SAS Institute Inc. Also presented at the 2005 PhUse Conference. Available <http://www.lexjansen.com/phuse/2005/ts/ts09.pdf>.

- Henderson, Don. 2014. "PROC STREAM and SAS® Server Pages: Generating Custom HTML Reports." *Proceedings of the SAS Global Forum 2014 Conference*. Cary, NC: SAS Institute Inc. Available <http://support.sas.com/resources/papers/proceedings14/1738-2014.pdf>.
- Henderson, Don, and Arthur L. Carpenter. 2012. "Macro Programming Best Practices: Styles, Guidelines and Conventions Including the Rationale Behind Them." Presented at the SAS Global Forum 2012 Conference. Notes from the panel discussion available [http://www.sascommunity.org/wiki/Macro\\_Programming\\_Best\\_Practices:\\_Styles,\\_Guidelines\\_and\\_Conventions\\_Including\\_the\\_Rationale\\_Behind\\_Them](http://www.sascommunity.org/wiki/Macro_Programming_Best_Practices:_Styles,_Guidelines_and_Conventions_Including_the_Rationale_Behind_Them).
- Henry, Joseph. 2015. "REST at Ease with SAS®: How to Use SAS to Get Your REST." *Proceedings of the SAS Global Forum 2015 Conference*. Cary, NC: SAS Institute Inc. Available <http://support.sas.com/resources/papers/proceedings15/SAS1927-2015.pdf>.
- Hessel, Colin. 1998. "Introducing the SAS System for Windows into a UNIX Statistical Computing Environment: Challenges from a User's Perspective." *Proceedings of the Sixth Annual Western Users of SAS Software Conference* 370–372. Cary, NC: SAS Institute Inc. Available <http://www.lexjansen.com/wuss/1998/WUSS98085.pdf>.
- Hirabayashi, Sharon Matsumoto. 2001. "%ABC\_FREQ: A SAS Macro Utility to Generate Enhanced Frequency Reports." *Proceedings of the Fourteenth Annual NorthEast SAS Users Group Conference* 619–628. Cary, NC: SAS Institute Inc. Available <http://www.lexjansen.com/nesug/nesug01/ps/ps8005.pdf>.
- Holland, Philip R. 2016. "Writing Reusable Macros." *Proceedings of the SAS Global Forum 2016 Conference*. Cary, NC: SAS Institute Inc. Available <http://support.sas.com/resources/papers/proceedings16/6503-2016.pdf>.
- Howell, Andrew. 2015. "To %Bquote or not to %Bquote? That is the question.. (which drives SAS® Macro programmers around the bend.)" *Proceedings of the SAS Global Forum 2015 Conference*. Cary, NC: SAS Institute Inc. Available <http://support.sas.com/resources/papers/proceedings15/3268-2015.pdf>.
- Huang, Liping. 2003. "Report? Make It Easy—An Example of Creating Dynamic Reports into Excel." *Proceedings of the Twenty-Eighth Annual SAS Users Group International Conference*. Cary, NC: SAS Institute Inc. Available <http://www2.sas.com/proceedings/sugi28/101-28.pdf>.
- Hubbell, Katie A. 1990. "Conquering the Dreaded Macro Error." *Proceedings of the Fifteenth Annual SAS Users Group International Conference*, Nashville, TN, 1–7. Republished in *SAS Macro Facility Tips and Techniques, Version 6* (Cary, NC: SAS Institute Inc. 1994), 59–65. Available <http://www.lexjansen.com/nesug/nesug89/NESUG89019.pdf>.
- Hughes, Troy Martin. 2014a. "From a One-Horse to a One-Stoplight Town: A Base SAS Solution to Preventing Data Access Collisions through the Detection and Deployment of Shared and Exclusive File Locks." *Proceedings of the Twenty-Second Annual Western Users of SAS Software Conference*. Cary, NC: SAS Institute Inc. Available [http://www.lexjansen.com/wuss/2014/69\\_Final\\_Paper\\_PDF.pdf](http://www.lexjansen.com/wuss/2014/69_Final_Paper_PDF.pdf).
- Hughes, Troy Martin. 2014b. "Why Aren't Exception Handling Routines Routine? Toward Reliably Robust Code through Increased Quality Standards in Base SAS." *Proceedings of the Twenty-Fifth Annual MidWest SAS Users Group Conference*. Cary, NC: SAS Institute Inc. Available <http://www.mwsug.org/proceedings/2014/BB/MWSUG-2014-BB17.pdf>.
- Hughes, Troy Martin. 2016a. "Sorting a Bajillion Records: Conquering Scalability in a Big Data World." *Proceedings of the SAS Global Forum 2016 Conference*. Cary, NC: SAS Institute Inc. Available <http://support.sas.com/resources/papers/proceedings16/11888-2016.pdf>.
- Hughes, Troy Martin. 2016b. "SAS® Spontaneous Combustion: Securing Software Portability through Self-Extracting Code." *Proceedings of the SAS Global Forum 2016 Conference*. Cary, NC: SAS Institute Inc. Available <http://support.sas.com/resources/papers/proceedings16/11768-2016.pdf>.
- Izrael, David, David C. Hoaglin, and Michael P. Battaglia. 2000. "A SAS Macro for Balancing a Weighted Sample." *Proceedings of the Twenty-Fifth Annual SAS Users Group International Conference* 1350–1355. Cary, NC: SAS Institute Inc. Available <http://www2.sas.com/proceedings/sugi25/25/st/25p258.pdf>.

- Jaffe, Jay A. 1999. "SAS Macros: Beyond the Basics." *Proceedings of the Twenty-Fourth Annual SAS Users Group International Conference* 881–890. Cary, NC: SAS Institute Inc. Also published in *Proceedings of the Sixth Annual Western Users of SAS Software Conference* (1998), 523–532. Cary, NC: SAS Institute Inc. Available <http://www2.sas.com/proceedings/sugi24/Handson/p149-24.pdf>.
- Jensen, Karl, and Matt Greathouse. 2000. "The Autocall Macro Facility in the SAS for Windows Environment." *Proceedings of the Twenty-Fifth Annual SAS Users Group International Conference* 463–464. Cary, NC: SAS Institute Inc. Available <http://www2.sas.com/proceedings/sugi25/25cc/25p075.pdf>.
- Jia, Justin, and Amanda Lin. 2015. "Yes, SAS® can do! --- Manage External Files With SAS Programming." *Proceedings of the SAS Global Forum 2015 Conference*. Cary, NC: SAS Institute Inc. Available <http://support.sas.com/resources/papers/proceedings15/3262-2015.pdf>.
- Jiang, Jonson C. 2003. "Macros Used to Create Descriptive Statistic Summary Table of Neurotoxicity Scores." *Proceedings of the Annual Pharmaceutical SAS Users Group Conference* 135–138. Cary, NC: SAS Institute Inc. Available <http://www.lexjansen.com/pharmasug/2003/CodersCorner/cc029.pdf>.
- Jin, Jiang, Ye Jin, and Diane Wang. 2003. "If Only 'Page 1 of 1000.'" *Proceedings of the Twenty-Eighth Annual SAS Users Group International Conference*. Cary, NC: SAS Institute Inc. Available <http://www2.sas.com/proceedings/sugi28/081-28.pdf>.
- Johnson, Jim. 2001. "Programming Squared (Writing Programs That Write Programs)." *Proceedings of the Annual Pharmaceutical SAS Users Group Conference* 423–428. Cary, NC: SAS Institute Inc. Also published in *Proceedings of the Fourteenth Annual NorthEast SAS Users Group Conference* (2001) 76–81. Cary, NC: SAS Institute Inc. Available <http://www.lexjansen.com/nesug/nesug01/at/at1011.pdf>.
- Johnson, Martha, and Michael Gilman. 1993. "Submitting Macro Language Code in Window Commands." *Observations: The Technical Journal for SAS Software Users*, vol. 2, no. 4 (3rd quarter): 50–54. Cary, NC: SAS Institute Inc.
- Kelley, Francis J. 2003. "So Many Files, So Little Time (or Inclination) to Type Their Names: Spreadsheets by the Hundreds." *Proceedings of the Twenty-Eighth Annual SAS Users Group International Conference*. Cary, NC: SAS Institute Inc. Available <http://www2.sas.com/proceedings/sugi28/074-28.pdf>.
- Kelly, Tim. 2000. "SASHELP: A Backstage Pass." *Proceedings of the Thirteenth Annual NorthEast SAS Users Group Conference* 269–270. Cary, NC: SAS Institute Inc. Available <http://www.lexjansen.com/nesug/nesug00/cc/cc4008.pdf>.
- Kenney, Tim. 1999. "Using SAS Macros to Automate External File Identification and Manipulation—The Power of %SYSFUNC." *Proceedings of the Seventh Annual Western Users of SAS Software Conference* 90–93. Cary, NC: SAS Institute Inc. Available <http://www.lexjansen.com/wuss/1999/WUSS99021.pdf>.
- Kowitz, Kevin P. 2000. "Variable and Format Consistency - A Macro Approach for Reading in Flat Files." *Proceedings of the Twenty-Fifth Annual SAS Users Group International Conference* 1186–1188. Cary, NC: SAS Institute Inc. Available <http://www2.sas.com/proceedings/sugi25/25po/25p229.pdf>.
- Krenzke, Tom, Katie Hubbell, Mamadou Diallo, Amita Gopinath, and Sixia Chen. 2014. "Data Coarsening and Data Swapping Algorithms." *Proceedings of the SAS Global Forum 2014 Conference*. Cary, NC: SAS Institute Inc. Available <http://support.sas.com/resources/papers/proceedings14/1603-2014.pdf>.
- Kretzman, Peter. 1992. "Ifs, Ands, and Buts: A Case Study in Advanced Macro Implementation." *Proceedings of the Seventeenth Annual SAS Users Group International Conference* 176–182. Cary, NC: SAS Institute Inc. Republished in *SAS Macro Facility Tips and Techniques, Version 6* (Cary, NC: SAS Institute Inc., 1994), 143–149. <http://www.sascommunity.org/sugi/SUGI92/Sugi-92-33%20Kretzman.pdf>
- Kunselman, Thomas E. 2001. "Sinking the Big Hole: Using SAS/IntrNet Application Dispatcher Version 8 Features to Drill Down to the Depths of Your Data." *Proceedings of the Ninth Annual Western*

- Users of SAS Software Conference* 463–468. Cary, NC: SAS Institute Inc. Available <http://www.lexjansen.com/wuss/2001/WUSS01092.pdf>.
- Lafler, Kirk Paul. 2010. “Exploring DICTIONARY Tables and SASHELP Views.” *Proceedings of the SAS® Global Forum 2010 Conference*. Cary, NC: SAS Institute Inc. Available <http://support.sas.com/resources/papers/proceedings10/155-2010.pdf>.
- Lafler, Kirk Paul. 2015. “Hands-On SAS Macro Programming Essentials for New Users.” *Proceedings of the SAS Global Forum 2015 Conference*. Cary, NC: SAS Institute Inc. Available <http://support.sas.com/resources/papers/proceedings15/2506-2015.pdf>.
- Lafler, Kirk Paul. 2016. “SAS Debugging 101.” *Proceedings of the SAS Global Forum 2016 Conference*. Cary, NC: SAS Institute Inc. Available <http://support.sas.com/resources/papers/proceedings16/2748-2016.pdf>.
- Landers, K. Larry, and Monique Bryher. 1997. “Taking the Mystery out of SAS MACRO When Using CALL SYMPUT.” *Proceedings of the Fifth Annual Western Users of SAS Software Conference* 29–34. Cary, NC: SAS Institute Inc. Also published in *Proceedings of the Twenty-Third Annual SAS Users Group International Conference* 445–450. Cary, NC: SAS Institute Inc. Available <http://www2.sas.com/proceedings/sugi23/Coders/p83.pdf>.
- Langston, Richard D. 2013. “A Macro to Verify a Macro Exists.” *Proceedings of the SAS Global Forum 2013 Conference*. Cary, NC: SAS Institute Inc. Available <http://support.sas.com/resources/papers/proceedings13/339-2013.pdf>.
- Langston, Richard D. 2015a. “New Macro Features Added in SAS® 9.3 and SAS® 9.4.” *Proceedings of the SAS® Global Forum 2015 Conference*. Cary, NC: SAS Institute Inc. Available <http://support.sas.com/resources/papers/proceedings15/SAS1575-2015.pdf>.
- Langston, Richard D. 2015b. “Don’t Be a Litterbug: Best Practices for Using Temporary Files in SAS.” *Proceedings of the SAS® Global Forum 2015 Conference*. Cary, NC: SAS Institute Inc. Available <http://support.sas.com/resources/papers/proceedings15/SAS1573-2015.pdf>.
- Larsen, Erik S. 2001. “A Macro to Center Text in a DATA \_NULL\_ Step.” *Proceedings of the Twenty-Sixth Annual SAS Users Group International Conference*. Cary, NC: SAS Institute Inc. Also published in *Proceedings of the Twelfth Annual NorthEast SAS Users Group Conference* (1999) 322–323.
- Cary, NC: SAS Institute Inc. Also published in *Proceedings of the Eighth Annual SouthEast SAS Users Group Conference* (2000). Cary, NC: SAS Institute Inc. Available <http://www2.sas.com/proceedings/sugi26/p087-26.pdf>.
- LeBouton, Kimberly J., and Thomas W. Rice. 2000. “Smokin’ with UNIX Pipes.” *Proceedings of the Twenty-Fifth Annual SAS Users Group International Conference* 555–558. Cary, NC: SAS Institute Inc. Available <http://www2.sas.com/proceedings/sugi25/25cc/25p103.pdf>.
- Leighton, Ralph W. 1997. “SAS Macros - A Gentle Introduction for the Fearful.” *Proceedings of the Twenty-Second Annual SAS Users Group International Conference* 25–30. Cary, NC: SAS Institute Inc. Available <http://www2.sas.com/proceedings/sugi22/APPDEVEL/PAPER6.PDF>.
- Leprince, Daniel J., and Elizabeth Li. 2003. “A Plot and a Table per Page Times Hundreds in a Single PDF File.” *Proceedings of the Twenty-Eighth Annual SAS Users Group International Conference*. Cary, NC: SAS Institute Inc. Also available in *Proceedings of the Eleventh Annual Western Users of SAS Software Conference* (2003). Cary, NC: SAS Institute Inc. Available <http://www2.sas.com/proceedings/sugi28/141-28.pdf>.
- Levin, Lois. 2001. “SAS Programming Conventions.” *Proceedings of the Fourteenth Annual NorthEast SAS Users Group Conference* 838–844. Cary, NC: SAS Institute Inc. Also published in *Proceedings of the Twenty-Eighth Annual SAS Users Group International Conference*. Cary, NC: SAS Institute Inc. Available <http://www2.sas.com/proceedings/sugi28/241-28.pdf>.
- Levine, Howard. 1989. “Building Macro-Based Systems.” *Proceedings of the Fourteenth Annual SAS Users Group International Conference* 96–102. Cary, NC: SAS Institute Inc. Republished in *SAS Macro Facility Tips and Techniques, Version 6* (Cary, NC: SAS Institute Inc., 1994), 37–43. Available <http://www.sascommunity.org/sugi/SUGI89/Sugi-89-15%20Levine.pdf>.
- Li, Arthur X. 2010. “When Best to Use the %LET Statement, the SYMPUT Routine, or the INTO Clause to Create Macro Variables.” *Proceedings of the SAS Global Forum 2016 Conference*. Cary, NC: SAS Institute Inc. Available <http://support.sas.com/resources/papers/proceedings10/028-2010.pdf>.

- Liang, Shuhua, and Fagen Xie. 2016. "Increasing Efficiency by Parallel Processing." *Proceedings of the SAS Global Forum 2016 Conference*. Cary, NC: SAS Institute Inc. Available <http://support.sas.com/resources/papers/proceedings16/10780-2016.pdf>.
- Litzsinger, Michael A., Lisa K. Brooks, and Michael A. Riddle. 2002. "A Modular Approach to Portable Programming." *Proceedings of the Twenty-Seventh Annual SAS Users Group International Conference*. Cary, NC: SAS Institute Inc. An earlier version of this paper written by Michael A. Litzsinger and Lisa Kaye Brooks was published in *Proceedings of the Fourteenth Annual NorthEast SAS Users Group Conference* (2001) 152–159. Cary, NC: SAS Institute Inc. Available <http://www2.sas.com/proceedings/sugi27/p034-27.pdf>.
- Long, Ying. 2003. "Space Odyssey: Concatenate Zip Files into One Master File." *Proceedings of the Twenty-Eighth Annual SAS Users Group International Conference*. Cary, NC: SAS Institute Inc. Available <http://www2.sas.com/proceedings/sugi28/072-28.pdf>.
- Lopez, Roberto. 1998. "%MAKFMT: A Dynamic Approach to Format Creation." *Proceedings of the Eleventh Annual NorthEast SAS Users Group Conference* 660–668. Cary, NC: SAS Institute Inc. Available <http://www.lexjansen.com/nesug/nesug98/solu/p144.pdf>.
- Lopez, Victor A., and Bill Knowlton. 2016. "Data Review Listings on Auto-Pilot: Using SAS® and Windows Server to Automate Reports and Flag Incremental Data Records." *Proceedings of the SAS Global Forum 2016 Conference*. Cary, NC: SAS Institute Inc. Available <http://support.sas.com/resources/papers/proceedings16/10841-2016.pdf>.
- Lougee, Claudine. 2016. "To Macro or not to Macro: That Is the Question." *Proceedings of the SAS Global Forum 2016 Conference*. Cary, NC: SAS Institute Inc. Available <http://support.sas.com/resources/papers/proceedings16/11822-2016.pdf>.
- Lund, Pete. 2016. "Something Old, Something New: Flexible Reporting with DATA Step-based Tools." *Proceedings of the SAS Global Forum 2016 Conference*. Cary, NC: SAS Institute Inc. Available <http://support.sas.com/resources/papers/proceedings16/10040-2016.pdf>.
- Lund, Pete. 1998. "Need More 'Functionality'? Using the SAS Macro Language to Create User-Written Functions." *Proceedings of the Seventeenth Annual Pacific Northwest SAS Users Group Conference* 84–91. Cary, NC: SAS Institute Inc. Available <http://www.lexjansen.com/pnwsug/1998/PNWSUG98013.pdf>.
- Lund, Pete. 2000a. "A Macro Utility for Generating Formatted Log Comments." *Proceedings of the Twenty-fifth Annual SAS Users Group International Conference* 480–483. Cary, NC: SAS Institute Inc. Available <http://www2.sas.com/proceedings/sugi25/25cc/25p080.pdf>.
- Lund, Pete. 2000b. "My Favorite Functions." *Proceedings of the Twenty-Fifth Annual SAS Users Group International Conference* 484–486. Cary, NC: SAS Institute Inc. Available <http://www2.sas.com/proceedings/sugi25/25cc/25p081.pdf>.
- Lund, Pete. 2001a. "Make Your Life a Little Easier: A Collection of SAS Macro Utilities." *Proceedings of the Twenty-Sixth Annual SAS Users Group International Conference*. Cary, NC: SAS Institute Inc. Also published in *Proceedings of the Ninth Annual Western Users of SAS Software Conference* (2001) 92–101. Cary, NC: SAS Institute Inc. Available <http://www.lexjansen.com/wuss/2001/WUSS01021.pdf>.
- Lund, Pete. 2001b. "A Macro Utility for Generating Formatted Log Comments." *Proceedings of the Ninth Annual Western Users of SAS Software Conference* 104–107. Cary, NC: SAS Institute Inc. Available <http://www.lexjansen.com/wuss/2001/WUSS01023.pdf>.
- Lund, Pete. 2001c. "My Favorite Functions: Using the SAS Macro Language to Create User-Written Functions." *Proceedings of the Ninth Annual Western Users of SAS Software Conference* (2001) 629–635. Cary, NC: SAS Institute Inc. Available <http://www.lexjansen.com/wuss/2001/WUSS01130.pdf>.
- Lund, Pete. 2002. "A Quick and Easy Data Dictionary Macro." *Proceedings of the Twenty-Seventh Annual SAS Users Group International Conference*. Cary, NC: SAS Institute Inc. Also published in *Proceedings of the Tenth Annual Western Users of SAS Software Conference* (2002) 7–12. Cary, NC: SAS Institute Inc. Available <http://www2.sas.com/proceedings/sugi27/p099-27.pdf>.
- Lund, Pete. 2003a. "SAS Helps Those Who Help Themselves: Creating Tools to Aid in Your Application Development." *Proceedings of the Twenty-Eighth Annual SAS Users Group International*

- Conference*. Cary, NC: SAS Institute Inc. Available <http://www2.sas.com/proceedings/sugi28/029-28.pdf>.
- Lund, Pete. 2003b. "Make Your Life a Little Easier: A Collection of SAS Macro Utilities." *Proceedings of the Twenty-Eighth Annual SAS Users Group International Conference*. Cary, NC: SAS Institute Inc. Available <http://www2.sas.com/proceedings/sugi28/085-28.pdf>.
- Lund, Pete. 2003c. "Keep Those Formats Rolling: A Macro to Manage the FMTSEARCH= Option." *Proceedings of the Twenty-Eighth Annual SAS Users Group International Conference*. Cary, NC: SAS Institute Inc. Available <http://www2.sas.com/proceedings/sugi28/116-28.pdf>.
- Luo, Haining, and Haiping Luo. 2003. "Building a Metadata Repository for Data Sets." *Proceedings of the Twenty-Eighth Annual SAS Users Group International Conference*. Cary, NC: SAS Institute Inc. Available <http://www2.sas.com/proceedings/sugi28/045-28.pdf>.
- Mace, Michael A. 1997. "Using %WINDOW to Obtain User Criteria for Reports." *Proceedings of the Tenth Annual NorthEast SAS Users Group Conference* 761–765. Cary, NC: SAS Institute Inc. Also published in *Proceedings of the 1999 MidWest SAS Users Group Conference*. Cary, NC: SAS Institute Inc. Available <http://www.lexjansen.com/mwsug/1999/paper17.pdf>.
- Mace, Michael A. 1998. "Obtaining User Criteria without Editing the Code." *Proceedings of the Eleventh Annual NorthEast SAS Users Group Conference* 669–674. Cary, NC: SAS Institute Inc. Also published in *Proceedings of the 1999 MidWest SAS Users Group Conference*. Cary, NC: SAS Institute Inc. Available <http://www.lexjansen.com/mwsug/1999/paper19.pdf>.
- Mace, Michael A. 1999. "Using &SYSBUFFR to Gather User Input." *Proceedings of the Twelfth Annual NorthEast SAS Users Group Conference* 324–325. Cary, NC: SAS Institute Inc. Available <http://www.lexjansen.com/nesug99/cc/cc146.pdf>.
- Mace, Michael A. 2000. "Using %WINDOW to Gather User Criteria." *Proceedings of the Twenty-Fifth Annual SAS Users Group International Conference* 142–146. Cary, NC: SAS Institute Inc. Also published in *Proceedings of the Thirteenth Annual NorthEast SAS Users Group Conference* 610–614. Cary, NC: SAS Institute Inc. Available <http://www2.sas.com/proceedings/sugi25/25/ad/25p022.pdf>.
- Mace, Michael A. 2002. "%WINDOW: You Can Talk to the Users, and They Can Talk Back." *Proceedings of the Twenty-Seventh Annual SAS Users Group International Conference*. Cary, NC: SAS Institute Inc. Available <http://www2.sas.com/proceedings/sugi27/p192-27.pdf>.
- Mace, Michael A. 2003. "%WINDOW: Get the Parameters the User Wants and You Need." *Proceedings of the Twenty-Eighth Annual SAS Users Group International Conference*. Cary, NC: SAS Institute Inc. Available <http://www2.sas.com/proceedings/sugi28/021-28.pdf>.
- Mahnken, Jonathan D. 2002. "Many-to-Many Merging Using the SAS Macro Facility." *Proceedings of the Twenty-Seventh Annual SAS Users Group International Conference*. Cary, NC: SAS Institute Inc. Available <http://www2.sas.com/proceedings/sugi27/p089-27.pdf>.
- Maldonado, Miguel, Jared Dean, Wendy Czika, and Susan Haller. 2014. "Leveraging Ensemble Models in SAS® Enterprise Miner™." *Proceedings of the SAS Global Forum 2014 Conference*. Cary, NC: SAS Institute Inc. Available <http://support.sas.com/resources/papers/proceedings14/SAS133-2014.pdf>.
- Mao, Cailiang. 2001. "Automate and Customize Printing of SAS Output in UNIX." *Proceedings of the Annual Pharmaceutical SAS Users Group Conference* 87–88. Cary, NC: SAS Institute Inc. Available [http://www.lexjansen.com/pharmasug/2001/Proceed/Coders/CC03\\_mao.pdf](http://www.lexjansen.com/pharmasug/2001/Proceed/Coders/CC03_mao.pdf).
- Mao, Sam. 2003. "Cross-Referencing Data Contents and Case Report Form." *Proceedings of the Annual Pharmaceutical SAS Users Group Conference*. Cary, NC: SAS Institute Inc. Available <http://www.lexjansen.com/pharmasug/2003/CodersCorner/cc028.pdf>.
- Mason, Phil. 2016. "Creating Amazing Visualisations With SAS® Stored Processes and Javascript Libraries." *Proceedings of the SAS Global Forum 2016 Conference*. Cary, NC: SAS Institute Inc. Available <http://support.sas.com/resources/papers/proceedings16/6960-2016.pdf>.
- Mast, Greg. 1997. "Managing Disk Space with SAS." *Proceedings of the Twenty-Second Annual SAS Users Group International Conference* 1536–1541. Cary, NC: SAS Institute Inc. Available <http://www2.sas.com/proceedings/sugi22/TRAINING/PAPER325.PDF>.

- Matise, Joe. 2015. "Unravelling the Knot of Ampersands." *Proceedings of the SAS Global Forum 2015 Conference*. Cary, NC: SAS Institute Inc. Available <http://support.sas.com/resources/papers/proceedings15/3285-2015.pdf>.
- McCartney, Jeff, and Raymond Hu. 2001. "SAS Shorts: Valuable Tips for Everyday Programming." *Proceedings of the Fourteenth Annual NorthEast SAS Users Group Conference* 92–97. Cary, NC: SAS Institute Inc. Available <http://www.lexjansen.com/nesug/nesug01/at/at1013.pdf>.
- McMullen, Quentin. 2012. "%Assert() your way to sleep-filled nights: A one line data validation macro." *Proceedings of the Twenty-Fifth Annual NorthEast SAS Users Group Conference*. Cary, NC: SAS Institute Inc. Available <http://www.lexjansen.com/nesug/nesug12/cc/cc31.pdf>.
- Michel, Denis. 2005. "CALL EXECUTE: A Powerful Data Management Tool." *Proceedings of the Thirtieth Annual SAS Users Group International Conference*. Cary, NC: SAS Institute Inc. Available <http://www2.sas.com/proceedings/sugi30/027-30.pdf>.
- Michelsen, Jesper. 2014a. "ODBC Connection to a Database using Keywords and SAS® Macros." *Proceedings of the SAS Global Forum 2014 Conference*. Cary, NC: SAS Institute Inc. Available <http://support.sas.com/resources/papers/proceedings14/1297-2014.pdf>.
- Michelsen, Jesper. 2014b. "What's on my Mainframe? A macro that gives you a solid overview of your data on the mainframe." *Proceedings of the SAS Global Forum 2014 Conference*. Cary, NC: SAS Institute Inc. Available <http://support.sas.com/resources/papers/proceedings14/1296-2014.pdf>.
- Miralles, Romain. 2016. "Developing an On-Demand Web Report Platform Using Stored Processes and SAS® Web Application Server." *Proceedings of the SAS Global Forum 2016 Conference*. Cary, NC: SAS Institute Inc. Available <http://support.sas.com/resources/papers/proceedings16/10740-2016.pdf>.
- Molter, Michael, Scott Millard, and Steve Paciocco. 2003. "Dynamically Building SQL Queries Using Metadata Tables and Macro Processing." *Proceedings of the Twenty-Eighth Annual SAS Users Group International Conference*. Cary, NC: SAS Institute Inc. Available <http://www2.sas.com/proceedings/sugi28/043-28.pdf>.
- Moors, David. 2016. "Importing Metadata Programmatically Using the SAS® Batch Import Tool." *Proceedings of the SAS Global Forum 2016 Conference*. Cary, NC: SAS Institute Inc. Available <http://support.sas.com/resources/papers/proceedings16/6463-2016.pdf>.
- Moriak, Chris. 2005. "Making Remote Processing Less Remote." *Proceedings of the Eighteenth Annual NorthEast SAS Users Group Conference*. Cary, NC: SAS Institute Inc. Available <http://www.lexjansen.com/nesug/nesug05/io/io2.pdf>.
- Morrill, John, Kristi Wiser, and Jay Zhou. 2002. "A Data-Driven Macro Automating the Data Presentation Process by Generating Tailored, Customizable SAS Code: Relax, Let %TABGEN Do Your Work!" *Proceedings of the Annual Pharmaceutical SAS Users Group Conference* 43–48. Cary, NC: SAS Institute Inc. Available <http://www.lexjansen.com/pharmasug/2002/proceed/AppDev/ad10.pdf>.
- Mounib, Edgar L., and Thiru Satchi. 2000. "Matched Sampling Using SAS Software." *Proceedings of the Annual Pharmaceutical SAS Users Group Conference* 275–280. Cary, NC: SAS Institute Inc. Available <http://www.lexjansen.com/pharmasug/2000/Posters/p07.pdf>.
- Muller, Roger D. 2014. "Managing the Organization of SAS® Format and Macro Code Libraries in Complex Environments Including PC SAS, SAS® Enterprise Guide®, and UNIX SAS." *Proceedings of the SAS Global Forum 2014 Conference*. Cary, NC: SAS Institute Inc. Available <http://support.sas.com/resources/papers/proceedings14/1862-2014.pdf>.
- Muller, Roger D. 2016. "Considerations in Organizing the Structure of SAS® Macro Libraries." *Proceedings of the SAS Global Forum 2016 Conference*. Cary, NC: SAS Institute Inc. Available <http://support.sas.com/resources/papers/proceedings16/11887-2016.pdf>.
- Murphy, William C. 2003a. "Using a SAS Macro to Document the Database." *Proceedings of the Twenty-Eighth Annual SAS Users Group International Conference*. Cary, NC: SAS Institute Inc. Available <http://www2.sas.com/proceedings/sugi28/091-28.pdf>.

- Murphy, William C. 2003b. "Filling Report Templates with the SAS System and DDE." *Proceedings of the Twenty-Eighth Annual SAS Users Group International Conference*. Cary, NC: SAS Institute Inc. Available <http://www2.sas.com/proceedings/sugi28/217-28.pdf>.
- Murphy, William C. 2007. "Changing Data Set Variables into Macro Variables." *Proceedings of the SAS Global Forum 2007 Conference*. Cary, NC: SAS Institute Inc. Available <http://www2.sas.com/proceedings/forum2007/050-2007.pdf>.
- Musterer, Robert, and Owen Jiang. 1995. "Use of the CALL EXECUTE Routine to Pass a List of Variables from a DATA Step to a PROC Step." *Proceedings of the Eighth Annual NorthEast SAS Users Group Conference* 552–554. Cary, NC: SAS Institute Inc. Available <http://www.lexjansen.com/nesug/nesug95/NESUG95103.pdf>.
- Noda, Art, Helena Chmura Kraemer, and Vyjeyanthi S. Periyakoil. 2000. "Using SAS to Calculate Kappa and Confidence Intervals for Binary Data with Multiple Raters, and for the Consensus of Multiple Diagnoses." *Proceedings of the Eighth Annual Western Users of SAS Software Conference* (2000) 154–159. Cary, NC: SAS Institute Inc. Available <http://www.lexjansen.com/wuss/2000/WUSS00034.pdf>.
- Norton, Andrew A. 1991. "Screen Control Language versus Macros in Batch Environments." *Proceedings of the Sixteenth Annual SAS Users Group International Conference* 1325–1330. Cary, NC: SAS Institute Inc. Republished in *SAS Macro Facility Tips and Techniques, Version 6* (Cary, NC: SAS Institute Inc., 1994), 106–111. Cary, NC: SAS Institute Inc. Available <http://www.sascommunity.org/sugi/SUGI91/Sugi-91-237%20Norton.pdf>.
- O'Connor, Susan M. 1991. "A Roadmap to Macro Facility Error Messages and Debugging." *Proceedings of the Sixteenth Annual SAS Users Group International Conference* 215–222. Cary, NC: SAS Institute Inc. Republished in *SAS Macro Facility Tips and Techniques, Version 6* (Cary, NC: SAS Institute Inc., 1994), 73–80. Cary, NC: SAS Institute Inc. Available <http://www.sascommunity.org/sugi/SUGI91/Sugi-91-40%20OConnor.pdf>.
- O'Connor, Susan M. 1992. "Macros Invocation Hierarchy: Session Compiled, Autocall, and Compiled Stored Macros." *Proceedings of the Seventeenth Annual SAS Users Group International Conference* 19–23. Cary, NC: SAS Institute Inc. Republished in *SAS Macro Facility Tips and Techniques, Version 6* (Cary, NC: SAS Institute Inc., 1994), 101–105. Cary, NC: SAS Institute Inc. Available <http://www.sascommunity.org/sugi/SUGI92/Sugi-92-05%20OConnor.pdf>.
- O'Connor, Susan M. 1998. "Macro Internals for the User - Developer's Overview." *Proceedings of the Seventh Annual SouthEast SAS Users Group Conference* (1999). Cary, NC: SAS Institute Inc. Available <http://analytics.ncsu.edu/sesug/1999/027.pdf>.
- O'Connor, Susan. 1999. "Secrets of Macro Quoting Functions—How and Why." *Proceedings of the Twelfth Annual NorthEast SAS Users Group Conference* 228–239. Cary, NC: SAS Institute Inc. Available <http://www.ats.ucla.edu/stat/sas/library/nesug99/bt185.pdf>.
- Olaleye, David. 1998. "Using SAS MACRO Facility and DATA Step Functions for Automated and Selective Deletion of Records with Missing Information." *Proceedings of the Eleventh Annual NorthEast SAS Users Group Conference* 675–684. Cary, NC: SAS Institute Inc. Available <http://www.lexjansen.com/nesug/nesug98/solu/p098.pdf>.
- Otteson, Rebecca, and Leanne Goldstein. 2015. "Getting Your Hands on Reproducible Graphs." *Proceedings of the SAS Global Forum 2015 Conference*. Cary, NC: SAS Institute Inc. Available <http://support.sas.com/resources/papers/proceedings15/3432-2015.pdf>.
- Pahmer, Emmy. 2014. "Making the Log a Forethought Rather Than an Afterthought." *Proceedings of the SAS Global Forum 2014 Conference*. Cary, NC: SAS Institute Inc. Available <http://support.sas.com/resources/papers/proceedings14/1556-2014.pdf>.
- Pahmer, Emmy. 2015. "Did Your Join Work Properly?" *Proceedings of the SAS Global Forum 2015 Conference*. Cary, NC: SAS Institute Inc. Available <http://support.sas.com/resources/papers/proceedings15/1762-2015.pdf>.
- Palmer, Lynn. 1997. "Methods of Finding a Small Group of Records in Two Million without Using Merge." *Proceedings of the Fifth Annual Western Users of SAS Software Conference* 381–386. Cary, NC: SAS Institute Inc. Available <http://www.lexjansen.com/wuss/1997/WUSS97092.pdf>.

- Palmer, Lynn. 1998. "Use Call Symput for More Informative Titles and Footnotes." *Proceedings of the Sixth Annual Western Users of SAS Software Conference* 110–113. Cary, NC: SAS Institute Inc. Available <http://www.lexjansen.com/wuss/1998/WUSS98024.pdf>.
- Palmer, Lynn. 2001. "Use Call Symput and %SYSFUNC for More Informative Titles and Footnotes." *Proceedings of the Twenty-Sixth Annual SAS Users Group International Conference*. Cary, NC: SAS Institute Inc. Available <http://www2.sas.com/proceedings/sugi26/p088-26.pdf>.
- Parker, Chevell. 2014. "Secrets from a SAS Technical Support Guy: Combining the Power of the SAS® Output Delivery System with Microsoft Excel Worksheets." *Proceedings of the SAS Global Forum 2014 Conference*. Cary, NC: SAS Institute Inc. Available <http://support.sas.com/resources/papers/proceedings14/SAS177-2014.pdf>.
- Parker, Chevell. 2015. "Staying Relevant in a Competitive World: Using the SAS® Output Delivery System to Enhance, Customize, and Render Reports." *Proceedings of the SAS® Global Forum 2015 Conference*. Cary, NC: SAS Institute Inc. Available <http://support.sas.com/resources/papers/proceedings15/SAS1880-2015.pdf>.
- Parker, Peter. 2000. "SAS Software Macros - You're Only Limited by Your Imagination." *Proceedings of the Eighth Annual SouthEast SAS Users Group Conference*. Cary, NC: SAS Institute Inc. Also published in *Proceedings of the Thirteenth Annual NorthEast SAS Users Group Conference* (2000) 53–62. Cary, NC: SAS Institute Inc. Available <http://analytics.ncsu.edu/sesug/2000/p-308.pdf>.
- Parker, Peter. 2003. "Using SAS Software to Analyze Web Logs." *Proceedings of the Twenty-Eighth Annual SAS Users Group International Conference*. Cary, NC: SAS Institute Inc. Available <http://www2.sas.com/proceedings/sugi28/026-28.pdf>.
- Pass, Ray. 1998. "What We Really Need Is a %BY Statement." *Proceedings of the Twenty-Third Annual SAS Users Group International Conference* 428–429. Cary, NC: SAS Institute Inc. Also published in *Proceedings of the Eleventh Annual NorthEast SAS Users Group Conference* (1998) 337–338. Cary, NC: SAS Institute Inc. Available <http://www2.sas.com/proceedings/sugi23/Coders/p77.pdf>.
- Pass, Ray. 2000. "What We Really Need Is a %BY Statement—V2." *Proceedings of the Twenty-Fifth Annual SAS Users Group International Conference* 492–493. Cary, NC: SAS Institute Inc. Also published in *Proceedings of the Thirteenth Annual NorthEast SAS Users Group Conference* (2000). Also *Proceedings of the Fourteenth Annual NorthEast SAS Users Group Conference* (2001) 327–328. Cary, NC: SAS Institute Inc. Available <http://www2.sas.com/proceedings/sugi25/25cc/25p083.pdf>.
- Peszek, Iza, and John Toxell. 2000. "An Interactive SAS Macro Catalog." *Proceedings of the Annual Pharmaceutical SAS Users Group Conference* 387–394. Cary, NC: SAS Institute Inc.
- Peterson, Donald W. 1999. "Customized Pagination Using %PAGEOFNO Macro." *Proceedings of the Twenty-Fourth Annual SAS Users Group International Conference* 547–548. Cary, NC: SAS Institute Inc. Available <http://www2.sas.com/proceedings/sugi24/Coders/p082-24.pdf>.
- Phillips, Jeff, Veronica Walgamotte, and Derek Drummond. 1993. "Warning: Apparent Macro Invocation Not Resolved... Techniques for Debugging Macro Code." *Proceedings of the Eighteenth Annual SAS Users Group International Conference* 424–429. Cary, NC: SAS Institute Inc. Republished in *SAS Macro Facility Tips and Techniques, Version 6* (Cary, NC: SAS Institute Inc., 1994), 47–52. Cary, NC: SAS Institute Inc. Available <http://www.sascommunity.org/sugi/SUGI94/Sugi-94-68%20Phillips%20Walgamotte%20Drummond.pdf>.
- Piet, John M. 2000. "Table Lookup on the Fly Using SYMPUT and SYMGET." *Proceedings of the Twenty-Fifth Annual SAS Users Group International Conference* 473–476. Cary, NC: SAS Institute Inc. Available <http://www2.sas.com/proceedings/sugi25/25cc/25p078.pdf>.
- Plath, Robert. 2002. "Optimum Sampling by Ratio Estimation." *Proceedings of the Tenth Annual Western Users of SAS Software Conference* 135–141. Cary, NC: SAS Institute Inc. Available <http://www.lexjansen.com/wuss/2002/WUSS02029.pdf>.
- Pochon, Philip M., and Thomas H. Burger. 1998. "Validation of SAS Macro Systems." *Proceedings of the Annual Pharmaceutical SAS Users Group Conference* 90–94. Cary, NC: SAS Institute Inc. Available [http://www.lexjansen.com/pharmasug/1998/DATA\\_VAL/POCHON.PDF](http://www.lexjansen.com/pharmasug/1998/DATA_VAL/POCHON.PDF).

- Price, Jennifer. 1998. "Using AUTOEXEC.SAS to Customize SAS Sessions." *Proceedings of the Eleventh Annual NorthEast SAS Users Group Conference* 555–556. Cary, NC: SAS Institute Inc. Available <http://www.lexjansen.com/nesug/nesug98/post/p040.pdf>.
- Pugh, Dorothy E. 2000. "A Robust Generalized Axis-Scaling SAS Macro." *Proceedings of the Eighth Annual SouthEast SAS Users Group Conference*. Cary, NC: SAS Institute Inc. Available <http://analytics.ncsu.edu/sesug/2000/p-412.pdf>.
- Rajecki, Aldona A., and Eric E. Kahle. 2000. "SAS Extract Macro to Create a Comma Delimited File for Microsoft Excel 97 Import." *Proceedings of the Thirteenth Annual NorthEast SAS Users Group Conference* 279–280. Cary, NC: SAS Institute Inc. Available <http://www.lexjansen.com/nesug/nesug00/cc/cc4013.pdf>.
- Rasheed, Harun, and Amarnath Vijayarangan. 2014. "Chasing the log file while running the SAS® program." *Proceedings of the SAS Global Forum 2014 Conference*. Cary, NC: SAS Institute Inc. Available <http://support.sas.com/resources/papers/proceedings14/1762-2014.pdf>.
- Ravi, Prasad. 2003. "Renaming All Variables in a SAS Data Set Using the Information from PROC SQL's Dictionary Tables." *Proceedings of the Twenty-Eighth Annual SAS Users Group International Conference*. Cary, NC: SAS Institute Inc. Available <http://www2.sas.com/proceedings/sugi28/118-28.pdf>.
- Reading, Pamela L. 2000. "Using SAS to Write SAS - Automate Your Programming Tasks." *Proceedings of the Eighth Annual SouthEast SAS Users Group Conference*. Cary, NC: SAS Institute Inc. Available <http://www2.sas.com/proceedings/sugi26/p092-26.pdf>.
- Ren, Quan. 1999. "Designing User Interface by Using the Macro Facility." *Proceedings of the Twenty-Fourth Annual SAS Users Group International Conference* 578–579. Cary, NC: SAS Institute Inc. Available <http://www2.sas.com/proceedings/sugi24/Coders/p092-24.pdf>.
- Rhoades, Stephen. 2001. "Recursion? SAS? Let's Fake It!" *Proceedings of the Fourteenth Annual NorthEast SAS Users Group Conference* 652–657. Cary, NC: SAS Institute Inc. Available <http://www.lexjansen.com/nesug/nesug01/ps/ps8013.pdf>.
- Rhoads, Amy, and Kent Letourneau. 2002. "You CAN Save Your Log and View It, Too: An Improved Process for Automatically Saving the Contents of the Log and Output Windows." *Proceedings of the Annual Pharmaceutical SAS Users Group Conference* 107–111. Cary, NC: SAS Institute Inc. Available <http://www.lexjansen.com/pharmasug/2002/proceed/Coders/cc10.pdf>.
- Riba, S. David. 1997. "Self-Modifying SAS Programs: A DATA Step Interface." *Observations: The Technical Journal for SAS Software Users*. Cary, NC: SAS Institute Inc. Also published in *Proceedings of the Ninth Annual NorthEast SAS Users Group Conference*. Cary, NC: SAS Institute Inc. Available <http://www.lexjansen.com/nesug/nesug96/NESUG96029.pdf>.
- Riba, S. David. 2000. "How to Use the Data Step Debugger." *Proceedings of the Eighth Annual Western Users of SAS Software Conference* 586–594. Cary, NC: SAS Institute Inc. Also published in *Proceedings of the Twenty-Fifth Annual SAS Users Group International Conference*. Cary, NC: SAS Institute Inc. Available <http://www2.sas.com/proceedings/sugi25/25/btu/25p052.pdf>.
- Roberts, Clark. 1997. "Building and Using Macro Variable Lists." *Proceedings of the Twenty-Second Annual SAS Users Group International Conference* 441–443. Cary, NC: SAS Institute Inc. Available <http://www2.sas.com/proceedings/sugi22/CODERS/PAPER78.PDF>.
- Rook, Christopher J., and Shi-Tao Yeh. 2001. "SAS Macros as File Management Utility Programs." *Proceedings of the Twenty-Sixth Annual SAS Users Group International Conference*. Cary, NC: SAS Institute Inc. Also published in *Proceedings of the Fourteenth Annual NorthEast SAS Users Group Conference* (2001) 663–667. Cary, NC: SAS Institute Inc. Available <http://www2.sas.com/proceedings/sugi26/p219-26.pdf>.
- Roper, Christopher A. 2001. "Accessing and Utilizing the Win32 API from SAS." *Proceedings of the Fourteenth Annual NorthEast SAS Users Group Conference* 170–175. Cary, NC: SAS Institute Inc. Also published in *Proceedings of the Twenty-Sixth Annual SAS Users Group International Conference*. Cary, NC: SAS Institute Inc. Available <http://www2.sas.com/proceedings/sugi26/p281-26.pdf>.
- Rosenbloom, Mary F. O., and Art Carpenter. 2011. "Macro Quoting to the Rescue: Passing Special Characters." *Proceedings of the Nineteenth Annual Western Users of SAS Software Conference*

2011. Cary, NC: SAS Institute Inc. Also published in *Proceedings of the Pharmaceutical SAS Users Group 2012 Conference*. Cary, NC: SAS Institute Inc. Also published in *Proceedings of the SAS Global Forum 2013 Conference*. Cary, NC: SAS Institute Inc. Available <http://support.sas.com/resources/papers/proceedings13/005-2013.pdf>.
- Rosenbloom, Mary F. O., and Art Carpenter. 2014. "Are You a Control Freak? Control Your Programs – Don't Let Them Control You!" *Proceedings of the Twenty-Second Annual Western Users of SAS Software Conference*. Cary, NC: SAS Institute Inc. Also published in *Proceedings of the SAS Global Forum 2015 Conference*. Cary, NC: SAS Institute Inc. Available <http://support.sas.com/resources/papers/proceedings15/2220-2015.pdf>.
- Sadof, Michael G. 1997. "Macros from Beginning to Mend: A Simple and Practical Approach to the SAS Macro Facility." *Proceedings of the Tenth Annual NorthEast SAS Users Group Conference* 257–263. Cary, NC: SAS Institute Inc. Available <http://www.ats.ucla.edu/stat/sas/library/nesug99/bt066.pdf>.
- SAS Institute Inc. 2014. "Using Macro Variables in SCL Programs." *SAS 9.4 Component Language: Reference*, 3d ed. Cary, NC: SAS Institute Inc. Discusses macro variables, and the substitution of text in SUBMIT blocks. Available <http://support.sas.com/documentation/cdl/en/sclref/67564/HTML/default/n1mjbfbsazcquanan1oe605j0rcfq4.htm>.
- SAS Institute Inc. 2015. *SAS 9.4 Macro Language: Reference*, 4th ed. Cary, NC: SAS Institute Inc. Available <http://support.sas.com/documentation/cdl/en/mcerref/67912/PDF/default/mcerref.pdf>.
- Satchi, Thiru. 2000. "Using the Host-Variable 'INTO:' in PROC SQL." *Proceedings of the Annual Pharmaceutical SAS Users Group Conference* 273–274. Cary, NC: SAS Institute Inc. Available <http://www.lexjansen.com/pharmasug/2000/Posters/p06.pdf>.
- Satchi, Thiru. 2001. "Using the Magical Keyword 'INTO:' in PROC SQL." *Proceedings of the Fourteenth Annual NorthEast SAS Users Group Conference* 253–259. Cary, NC: SAS Institute Inc. Also *Proceedings of the Twenty-Seventh Annual SAS Users Group International Conference*. Cary, NC: SAS Institute Inc. Available <http://www2.sas.com/proceedings/sugi27/p071-27.pdf>.
- Sattler, Jim. 2003. "A Table-Driven Solution for Clinical Data Submission." *Proceedings of the Twenty-Eighth Annual SAS Users Group International Conference*. Cary, NC: SAS Institute Inc. Available <http://www2.sas.com/proceedings/sugi28/040-28.pdf>.
- Schwarz, Kate. 2015. "Feeling Anxious about Transitioning from Desktop to Server? Key Considerations to Diminish Your Administrators' and Users' Jitters." *Proceedings of the SAS Global Forum 2015 Conference*. Cary, NC: SAS Institute Inc. Available <http://support.sas.com/resources/papers/proceedings15/SAS1750-2015.pdf>.
- Shen, Yanyun. 2003. "A Simplified and Efficient Way to Map Variable Attributes of a Clinical Data Warehouse." *Proceedings of the Twenty-Eighth Annual SAS Users Group International Conference*. Cary, NC: SAS Institute Inc. Available <http://www2.sas.com/proceedings/sugi28/117-28.pdf>.
- Shi, Changhong and Sylvianne B. Roberge. 2003. "Application of Some Advanced PROC SQL Features in Clinical Trial Programming." *Proceedings of the Annual Pharmaceutical SAS Users Group Conference* 195–200. Cary, NC: SAS Institute Inc. Available <http://www.lexjansen.com/pharmasug/2003/datamanagement/dm012.pdf>.
- Shilling, Brian C., and Timothy A. Kelly. 2001. "Open a Window and Lock the Door: A User Friendly Application That Maintains Validation Integrity." *Proceedings of the Annual Pharmaceutical SAS Users Group Conference* 407–414. Cary, NC: SAS Institute Inc. Available [http://www.lexjansen.com/pharmasug/2001/Proceed/TechTech/TT10\\_shilling.pdf](http://www.lexjansen.com/pharmasug/2001/Proceed/TechTech/TT10_shilling.pdf).
- Shtern, Elena. 2014. "Useful Tips When Deploying SAS® Code in a Production Environment." *Proceedings of the SAS Global Forum 2014 Conference*. Cary, NC: SAS Institute Inc. Available <http://support.sas.com/resources/papers/proceedings14/SAS258-2014.pdf>.
- Sissing, Lori. 2014. "%LET Me Help You Improve Your Reporting." *Proceedings of the Twenty-Fifth Annual MidWest SAS Users Group Conference*. Cary, NC: SAS Institute Inc. Available <http://mwsug.org/proceedings/2014/BI/MWSUG-2014-BI09.pdf>.

- Sisson, Emily K.Q. 2016. "File Management Using Pipes and X Commands in SAS®." *Proceedings of the SAS Global Forum 2016 Conference*. Cary, NC: SAS Institute Inc. Available <http://support.sas.com/resources/papers/proceedings16/8780-2016.pdf>.
- Smiley, Christine A. 1999. "A Fast Format Macro - How to Quickly Create a Format by Specifying the Endpoints." *Proceedings of the Twenty-Fourth Annual SAS Users Group International Conference* 585–588. Cary, NC: SAS Institute Inc. Available <http://www2.sas.com/proceedings/sugi24/Coders/p096-24.pdf>.
- Smith, Curtis A. 1999. "Dynamically Creating a Where Statement." *Proceedings of the Twenty-Fourth Annual SAS Users Group International Conference* 600–602. Cary, NC: SAS Institute Inc. Also published in *Proceedings of the Sixth Annual Western Users of SAS Software Conference* (1998) 125–127. Cary, NC: SAS Institute Inc. Available <http://www2.sas.com/proceedings/sugi24/Coders/p101-24.pdf>.
- Smith, Curtis A. 2003. "Creating Drill-Down Graphs Using SAS/GRAFH and the Output Delivery System." *Proceedings of the Twenty-Eighth Annual SAS Users Group International Conference*. Cary, NC: SAS Institute Inc. Also published in *Proceedings of the Tenth Annual Western Users of SAS Software Conference* (2002) 223–228. Cary, NC: SAS Institute Inc. Available <http://www2.sas.com/proceedings/sugi28/149-28.pdf>.
- Smith, Richard, and Art Carpenter. 2000. "The Lost Art of Annotate." *Proceedings of the Twenty-Fifth Annual SAS Users Group International Conference* 1050–1054. Cary, NC: SAS Institute Inc. Also published in *Proceedings of the Seventh Annual Western Users of SAS Software Conference* (1999) 420–424. Cary, NC: SAS Institute Inc. Available <http://www2.sas.com/proceedings/sugi25/25po/25p199.pdf>.
- Smith, Robert W. 1997. "Visual Hypothesis Testing with Confidence Intervals." *Proceedings of the Twenty-Second Annual SAS Users Group International Conference* 1252–1257. Cary, NC: SAS Institute Inc. Available <http://www2.sas.com/proceedings/sugi22/STATS/PAPER270.PDF>.
- Spicer, Jeanne. 2003. "'I'll Have What She's Having': Serving Up Meta-data to Academic Research Teams." *Proceedings of the Twenty-Eighth Annual SAS Users Group International Conference*. Cary, NC: SAS Institute Inc. Available <http://www2.sas.com/proceedings/sugi28/230-28.pdf>.
- Spotts, Bruce. 2002. "A Macro to Help with Accurate Output Documentation." *Proceedings of the Twenty-Seventh Annual SAS Users Group International Conference*. Cary, NC: SAS Institute Inc. Available <http://www2.sas.com/proceedings/sugi27/p097-27.pdf>.
- Squire, Jonathan. 1999. "Stop Counting Those Columns: A Macro Shell for Assigning Column Locations in Data \_NULL\_ Reporting." *Proceedings of the Seventh Annual Western Users of SAS Software Conference* 115–116. Cary, NC: SAS Institute Inc. Available <http://www.lexjansen.com/wuss/1999/WUSS99028.pdf>.
- Sridharma, Selvaratnam. 2003. "Splitting a Large SAS Data Set." *Proceedings of the Twenty-Eighth Annual SAS Users Group International Conference*. Cary, NC: SAS Institute Inc. Available <http://www2.sas.com/proceedings/sugi28/075-28.pdf>.
- Stanley, Don. 1994. *Beyond the Obvious with SAS Screen Control Language*. Cary, NC: SAS Institute Inc.
- Staum, Roger. 2002. "To Err Is Human; to Debug, Divine." *Proceedings of the Annual Pharmaceutical SAS Users Group Conference* 345–355. Cary, NC: SAS Institute Inc. Also published in *Proceedings of the Twenty-Seventh Annual SAS Users Group International Conference*. Cary, NC: SAS Institute Inc. Available <http://www2.sas.com/proceedings/sugi27/p064-27.pdf>.
- Stokke, Delayne. 2000. "Controlling Print Setup Options from within a Windows SAS Program." *Proceedings of the Twenty-Fifth Annual SAS Users Group International Conference* 452–457. Cary, NC: SAS Institute Inc. Available <http://www2.sas.com/proceedings/sugi25/25cc/25p073.pdf>.
- Stroupe, Jane. 2003. "Nine Steps to Get Started Using SAS Macros." *Proceedings of the Annual Pharmaceutical SAS Users Group Conference*. Cary, NC: SAS Institute Inc. Also published in *Proceedings of the Twenty-Eighth Annual SAS Users Group International Conference* (2003). Cary, NC: SAS Institute Inc. Available <http://www2.sas.com/proceedings/sugi28/056-28.pdf>.

- Stuelpner, Janet E. 1997. "Skipping, the Easy Way." *Proceedings of the Twenty-Second Annual SAS Users Group International Conference* 451–452. Cary, NC: SAS Institute Inc. Available <http://www2.sas.com/proceedings/sugi22/CODERS/PAPER82.PDF>.
- Suhr, Diana D. 2001. "%MACRO SKIP: My Favorite SAS Trick." *Proceedings of the Ninth Annual Western Users of SAS Software Conference* 542–543. Cary, NC: SAS Institute Inc. Available <http://www.lexjansen.com/wuss/2001/WUSS01109.pdf>.
- Sun, Eric, and Arthur L. Carpenter. 2011. "Protecting Macros and Macro Variables: It Is All About Control." *Proceedings of the Annual Pharmaceutical SAS Users Group Conference*. Cary, NC: SAS Institute Inc. Available <http://www.pharmasug.org/proceedings/2011/AD/PharmaSUG-2011-AD17.pdf>
- Sun, Jeff F. 1998. "Update a Two-Dimensional Matrix Using the Macro Facility." *Proceedings of the Twenty-Third Annual SAS Users Group International Conference* 437–439. Cary, NC: SAS Institute Inc. Available <http://www2.sas.com/proceedings/sugi23/Coders/p80.pdf>.
- Talbott, Helen-Jean, and Earl R. Westerlund. 1998. "How to Sort Production Reports Prior to Printing." *Proceedings of the Twenty-Third Annual SAS Users Group International Conference* 1039–1043. Cary, NC: SAS Institute Inc. Available <http://www2.sas.com/proceedings/sugi23/Posters/p188.pdf>.
- Tangedal, Mike. 2001. "The Ultimate SAS Macro (Make SAS Do All the Work!)." *Proceedings of the Twenty-Sixth Annual SAS Users Group International Conference*. Cary, NC: SAS Institute Inc. Available <http://www2.sas.com/proceedings/sugi26/p090-26.pdf>.
- Tassoni, Charles John, Baibai Chen, and Clara Chu. 1997. "One-to-One Matching of Case/Controls Using SAS Software." *Proceedings of the Twenty-Second Annual SAS Users Group International Conference* 1189–1190. Cary, NC: SAS Institute Inc. Available <http://www2.sas.com/proceedings/sugi22/POSTERS/PAPER257.PDF>.
- Teberg, John. 2002. "Jumping through the Months, with the Help of %SYSFUNC." *Proceedings of the Tenth Annual Western Users of SAS Software Conference* 65–67. Cary, NC: SAS Institute Inc. Available <http://www.lexjansen.com/wuss/2002/WUSS02014.pdf>.
- Thompson, Paul A. 2015. "Structuring your SAS® Applications for Long - Term Survival: Reproducible Methods in Base SAS® Programming." *Proceedings of the SAS® Global Forum 2015 Conference*. Cary, NC: SAS Institute Inc. Available <http://support.sas.com/resources/papers/proceedings15/3187-2015.pdf>.
- Thornton, S. Patrick. 1999. "SAS Macro Application Development: Automating the DATA Step and INPUT Processes." *Proceedings of the Seventh Annual Western Users of SAS Software Conference* 55–59. Cary, NC: SAS Institute Inc. Available <http://www.lexjansen.com/wuss/1999/WUSS99011.pdf>.
- Thornton, S. Patrick. 2014. "Tools of the SAS® Trade: A Centralized Macro-based Reporting System." *Proceedings of the SAS Global Forum 2014 Conference*. Cary, NC: SAS Institute Inc. Available <http://support.sas.com/resources/papers/proceedings14/1640-2014.pdf>.
- Tindall, Bruce M., and Susan M. O'Connor. 1991. "Macro Tricks to Astound the Folks on Thursday Morning: Ten Immediately Useful Macro Techniques." *Proceedings of the Sixteenth Annual SAS Users Group International Conference* 189–195. Cary, NC: SAS Institute Inc. Republished in *SAS Macro Facility Tips and Techniques, Version 6* (Cary, NC: SAS Institute Inc., 1994), 117–123. Available <http://www.sascommunity.org/sugi/SUGI91/Sugi-91-36%20Tindall%20OConnor.pdf>.
- Tomb, Michael E., and James R Carter. 2001. "Macros with Global Vision: Using the SAS Dictionary Tables to Create Tools." *Proceedings of the Annual Pharmaceutical SAS Users Group Conference* 271–278. Cary, NC: SAS Institute Inc. Also published in *Proceedings of the Fifteenth Annual NorthEast SAS Users Group Conference* (2002). Cary, NC: SAS Institute Inc. Available <http://www.lexjansen.com/nesug/nesug02/ps/ps011.pdf>.
- Troxell, John K. 2001. "Complex Macros v. Simple Programs: Resolving the Conflict." *Proceedings of the Annual Pharmaceutical SAS Users Group Conference* 381–382. Cary, NC: SAS Institute Inc. Available [http://www.lexjansen.com/pharmasug/2001/Proceed/TechTech/TT05\\_troxell.pdf](http://www.lexjansen.com/pharmasug/2001/Proceed/TechTech/TT05_troxell.pdf).
- Troxell, John K. 2002. "Bulletproofing and Knowledge Encapsulation in Statistical Macros." *Proceedings of the Annual Pharmaceutical SAS Users Group Conference* 467–470. Cary, NC: SAS Institute

- Inc. Also published in the 2003 *Proceedings of the SouthEast SAS Users Group Conference*. Cary, NC: SAS Institute Inc. Available <http://analytics.ncsu.edu/sesug/2003/AD07-Troxell.pdf>.
- Troxell, John K., and Chang-Min Chen. 2003. "Invisible Environmental Awareness Techniques." *Proceedings of the Annual Pharmaceutical SAS Users Group Conference*. Cary, NC: SAS Institute Inc. Available <http://www.lexjansen.com/pharmasug/2003/TechnicalTechniques/tt075.pdf>.
- Tze, Sylvia. 2000. "A Case-Transforming Macro for More Readable Test." *Proceedings of the Twenty-Fifth Annual SAS Users Group International Conference* 477–479. Cary, NC: SAS Institute Inc. Available <http://www2.sas.com/proceedings/sugi25/25cc/25p079.pdf>.
- Vandenbroucke, David A. 2016. "A Universal File Flattener." *Proceedings of the SAS Global Forum 2016 Conference*. Cary, NC: SAS Institute Inc. Available <http://support.sas.com/resources/papers/proceedings16/1780-2016.pdf>.
- Varney, Brian. 2016. "Greenspace: A Macro to Improve a SAS® Data Set Footprint." *Proceedings of the SAS Global Forum 2016 Conference*. Cary, NC: SAS Institute Inc. Available <http://support.sas.com/resources/papers/proceedings16/8180-2016.pdf>.
- Viergever, William. 2003. "Tips from the Hood: Challenging Problems and Tips from SAS-L." *Proceedings of the Twenty-Eighth Annual SAS Users Group International Conference*. Cary, NC: SAS Institute Inc. Available <http://www2.sas.com/proceedings/sugi28/020-28.pdf>.
- Vijayarangan, Amarnath. 2016. "Virtual Accessing of a SAS® Data Set Using OPEN, FETCH, and CLOSE Functions with %SYSFUNC and %DO Loops." *Proceedings of the SAS Global Forum 2016 Conference*. Cary, NC: SAS Institute Inc. Available <http://support.sas.com/resources/papers/proceedings16/8140-2016.pdf>.
- Virgile, Bob. 1997. "Magic with CALL EXECUTE." *Proceedings of the Twenty-Second Annual SAS Users Group International Conference* 465–466. Cary, NC: SAS Institute Inc. Available <http://www2.sas.com/proceedings/sugi22/CODERS/PAPER86.PDF>.
- Virgile, Robert. 2013. *SAS Macro Language Magic: Discovering Advanced Techniques*. Cary, NC: SAS Institute Inc. Available [http://www.sas.com/store/prodBK\\_66301\\_en.html](http://www.sas.com/store/prodBK_66301_en.html).
- Wang, Deli. 2015. "CREATIVE USES OF VECTOR PLOTS USING SAS®." *Proceedings of the SAS Global Forum 2015 Conference*. Cary, NC: SAS Institute Inc. Available <http://support.sas.com/resources/papers/proceedings15/2242-2015.pdf>.
- Wang, Xiaohui. 2003. "Techniques to Create a Standard Folder Structure in a Windows NT/XP Share Area for Submission Projects and Their Protocols." *Proceedings of the Annual Pharmaceutical SAS Users Group Conference*. Cary, NC: SAS Institute Inc. Available <http://www.lexjansen.com/pharmasug/2003/TechnicalTechniques/tt078.pdf>.
- Ward, David L. 1999. "Managing SAS Programs." *Proceedings of the Twenty-Fourth Annual SAS Users Group International Conference* 528–530. Cary, NC: SAS Institute Inc. Available <http://www2.sas.com/proceedings/sugi24/Coders/p076-24.pdf>.
- Ward, David. 2001. "Using Recursion in the SAS System." *Proceedings of the Annual Pharmaceutical SAS Users Group Conference* 97–98. Cary, NC: SAS Institute Inc. Available [http://www.lexjansen.com/pharmasug/2001/Proceed/Coders/CC12\\_ward.pdf](http://www.lexjansen.com/pharmasug/2001/Proceed/Coders/CC12_ward.pdf).
- Watts, Perry. 2002a. "Using ODS and the Macro Facility to Construct Color Charts and Scales for SAS Software Applications." *Proceedings of the Twenty-Seventh Annual SAS Users Group International Conference*. Cary, NC: SAS Institute Inc. Available <http://www2.sas.com/proceedings/sugi27/p125-27.pdf>.
- Watts, Perry. 2002b. *Multiple-Plot Displays: Simplified with Macros*. Cary, NC: SAS Institute Inc.

- Watts, Perry. 2003a. "Advanced Programming Techniques for Working with Color in SAS Software." *Proceedings of the Sixteenth Annual NorthEast SAS Users Group Conference*. Cary, NC: SAS Institute Inc. Also published in *Proceedings of the Twenty-Ninth Annual SAS Users Group International Conference*. Cary, NC: SAS Institute Inc. Available <http://www2.sas.com/proceedings/sugi29/091-29.pdf>.
- Watts, Perry. 2003b. "Working with RGB and HLS Color Coding Systems in SAS Software." *Proceedings of the Twenty-Eighth Annual SAS Users Group International Conference*. Cary, NC: SAS Institute Inc. Available <http://www2.sas.com/proceedings/sugi28/234-28.pdf>.
- Werner, Nina L. 2014. "Understanding Double Ampersand [&&] SAS® Macro Variables". *Proceedings of the Twenty-Fifth Annual MidWest SAS Users Group Conference*. Cary, NC: SAS Institute Inc. Available <http://mwsug.org/proceedings/2014/BI/MWSUG-2014-BI03.pdf>.
- Westerlund, Earl R. 1991. "SAS Macro Language Features for Application Development." *Proceedings of the Sixteenth Annual SAS Users Group International Conference* 245–248. Cary, NC: SAS Institute Inc. Republished in *SAS Macro Facility Tips and Techniques, Version 6* (Cary, NC: SAS Institute Inc., 1994), 89–92. Available <http://www.sascommunity.org/sugi/SUGI91/Sugi-91-45%20Westerlund.pdf>.
- Whitaker, Ken. 1989. "Using Macro Variable Lists." *Proceedings of the Fourteenth Annual SAS Users Group International Conference* 1531–1536. Cary, NC: SAS Institute Inc. Republished in *SAS Macro Facility Tips and Techniques, Version 6* (Cary, NC: SAS Institute Inc., 1994), 154–159. Available <http://www.sascommunity.org/sugi/SUGI89/Sugi-89-287%20Whitaker.pdf>.
- White, Michael L. 2001. "A SAS Macro to Isolate All Date Values from a Data Library into a SAS Dataset." *Proceedings of the Twenty-Sixth Annual SAS Users Group International Conference*. Cary, NC: SAS Institute Inc. Available <http://www2.sas.com/proceedings/sugi26/p098-26.pdf>.
- Whitlock, H. Ian. 1993. "A Macro to Make External Flat Files." *Proceedings of the Eighteenth Annual SAS Users Group International Conference* 258–263. Cary, NC: SAS Institute Inc. Republished in *SAS Macro Facility Tips and Techniques, Version 6* (Cary, NC: SAS Institute Inc., 1994) 225–230. Available <http://www.sascommunity.org/sugi/SUGI93/Sugi-93-43%20Whitlock.pdf>.
- Whitlock, H. Ian. 1997. "CALL EXECUTE: How and Why." *Proceedings of the Twenty-Second Annual SAS Users Group International Conference* 410–414. Cary, NC: SAS Institute Inc. Available <http://www2.sas.com/proceedings/sugi22/CODERS/PAPER70.PDF>.
- Whitlock, Ian. 1998. "The RESOLVE Function: What Is It Good For?" *Proceedings of the Eleventh Annual NorthEast SAS Users Group Conference* 352–353. Cary, NC: SAS Institute Inc. Available <http://www.lexjansen.com/nesug/nesug98/code/p088.pdf>.
- Whitlock, Ian. 1999a. "Getting Started with Macro." *Proceedings of the Twenty-Fourth Annual SAS Users Group International Conference* 434–443. Cary, NC: SAS Institute Inc. Also published in *Proceedings of the Twelfth Annual NorthEast SAS Users Group Conference* (1999) 274–283. Cary, NC: SAS Institute Inc. Also published in *Proceedings of the SouthEast SAS Users Group Conference* (1999). Cary, NC: SAS Institute Inc. Available <http://www.ats.ucla.edu/stat/sas/library/nesug99/bt046.pdf>.
- Whitlock, Ian. 1999b. "Make Your Own Macro Processor." *Proceedings of the Twelfth Annual NorthEast SAS Users Group Conference* 356–357. Cary, NC: SAS Institute Inc. Also published in *Proceedings of the Seventh Annual SouthEast SAS Users Group Conference* (1999). Cary, NC: SAS Institute Inc. Available <http://analytics.ncsu.edu/sesug/1999/104.pdf>.
- Whitlock, Ian. 1999c. "An OBS Limit with a WHERE Condition." *Proceedings of the Seventh Annual SouthEast SAS Users Group Conference*. Cary, NC: SAS Institute Inc. Available <http://analytics.ncsu.edu/sesug/1999/103.pdf>.
- Whitlock, Ian. 2000a. "Macro Design Considerations for a Word Wrapping Macro." *Proceedings of the Eighth Annual SouthEast SAS Users Group Conference*. Cary, NC: SAS Institute Inc. Available <http://analytics.ncsu.edu/sesug/2000/p-411.pdf>.
- Whitlock, Ian. 2000b. "Dictionary Files." *Proceedings of the Thirteenth Annual NorthEast SAS Users Group Conference* 303–304. Cary, NC: SAS Institute Inc. Available <http://www.lexjansen.com/nesug/nesug00/cc/cc4024.pdf>.

- Whitlock, Ian. 2001a. "SAS Macro: The First Steps to Power." *Proceedings of the Fourteenth Annual NorthEast SAS Users Group Conference* 283–292. Cary, NC: SAS Institute Inc. Available <http://www.lexjansen.com/nesug/nesug01/bt/bt3012.pdf>.
- Whitlock, Ian. 2001b. "What User Defined Formats Are Available?" *Proceedings of the Fourteenth Annual NorthEast SAS Users Group Conference* 349–350. Cary, NC: SAS Institute Inc. Available <http://www.lexjansen.com/nesug/nesug01/cc/cc4024.pdf>.
- Whitlock, Ian. 2003a. "A Serious Look at Macro Quoting." *Proceedings of the Twenty-Eighth Annual SAS Users Group International Conference*. Cary, NC: SAS Institute Inc. Available <http://www2.sas.com/proceedings/sugi28/011-28.pdf>.
- Widawski, Mel. 1997a. "A General Purpose Macro to Obtain a List of Files: Plus Macro Programming Techniques." *Proceedings of the Fifth Annual Western Users of SAS Software Conference* 94–99. Cary, NC: SAS Institute Inc. Available <http://www2.sas.com/proceedings/sugi23/Coders/p82.pdf>.
- Widawski, Mel. 1997b. "A General System for Custom Conversion of dBase Data into SAS." *Proceedings of the Fifth Annual Western Users of SAS Software Conference* 255–260. Cary, NC: SAS Institute Inc. Available <http://www.lexjansen.com/wuss/1997/WUSS97055.pdf>.
- Widawski, Mel. 1999. "Beginners Guide to Flexibility: Macro Variables." *Proceedings of the Seventh Annual Western Users of SAS Software Conference* 504–508. Cary, NC: SAS Institute Inc. Also published in *Proceedings of the Eighth Annual Western Users of SAS Software Conference* (2000) 475–479. Cary, NC: SAS Institute Inc. Available <http://www.lexjansen.com/wuss/2000/WUSS00084.pdf>.
- Widawski, Mel. 2002. "Flexible Code the Easy Way: SAS Macro Variables." *Proceedings of the Tenth Annual Western Users of SAS Software Conference* 570–574. Cary, NC: SAS Institute Inc. Updated and presented in *Proceedings of the Fourteenth Annual Western Users of SAS Software Conference* (2006). Cary, NC: SAS Institute Inc. Available [http://www.lexjansen.com/wuss/2006/SAS\\_essentials/ESS-Widawski.pdf](http://www.lexjansen.com/wuss/2006/SAS_essentials/ESS-Widawski.pdf).
- Williams, Christianna S. 2001. "Wholly MACRO! A Handful of Techniques to Simplify Your SAS Code and Make It Recyclable." *Proceedings of the Fourteenth Annual NorthEast SAS Users Group Conference* 293–302. Cary, NC: SAS Institute Inc. Available <http://www.lexjansen.com/nesug/nesug01/bt/bt3013.pdf>.
- Wilson, Nancy K. 2015. "Getting Your SAS® Program to do Your Typing for You!" *Proceedings of the SAS Global Forum 2015 Conference*. Cary, NC: SAS Institute Inc. Available <http://support.sas.com/resources/papers/proceedings15/3474-2015.pdf>.
- Wobus, Diana Zhang, and John Charles Gober. 1997. "A Step-by-Step Illustration of Building a Data Analysis Tool with Macros." *Proceedings of the Twenty-Second Annual SAS Users Group International Conference* 226–232. Cary, NC: SAS Institute Inc. Available <http://www2.sas.com/proceedings/sugi22/ADVTUTOR/PAPER42.PDF>.
- Wong, Steve. 2002. "Don't Copy This Program: Use SYSPARM Instead." *Proceedings of the Annual Pharmaceutical SAS Users Group Conference* 83–85. Cary, NC: SAS Institute Inc. Available <http://www.lexjansen.com/pharmasug/2002/proceed/Coders/cc05.pdf>.
- Wright, Wendi L. 2001. "Creating Macro Calls Using PROC FREQ." *Proceedings of the Fourteenth Annual NorthEast SAS Users Group Conference* 706–710. Cary, NC: SAS Institute Inc. Updated and represented in *Proceedings of the Seventeenth Annual NorthEast SAS Users Group Conference*. Cary, NC: SAS Institute Inc. Available <http://www.lexjansen.com/nesug/nesug04/pm/pm09.pdf>.
- Wu, William, Steven Li, and Yun Guan. 2016. "Importing Data Directly from PDF into SAS® Data Sets." *Proceedings of the SAS Global Forum 2016 Conference*. Cary, NC: SAS Institute Inc. Available <http://support.sas.com/resources/papers/proceedings16/9320-2016.pdf>.
- Xia, Huanhong, and Richard Birkenmaier. 2001. "P-Value Calculation Made Easy with the SAS Call Execute Routine." *Proceedings of the Annual Pharmaceutical SAS Users Group Conference* 89–92. Cary, NC: SAS Institute Inc. Available [http://www.lexjansen.com/pharmasug/2001/proceed/coders/cc01\\_xia.pdf](http://www.lexjansen.com/pharmasug/2001/proceed/coders/cc01_xia.pdf).
- Yao, Arthur K. 1997. "SAS Code Generator Based on Table-Driven Methodology in a Batch Environment." *Proceedings of the Twenty-Second Annual SAS Users Group International*

- Conference* 31–36. Cary, NC: SAS Institute Inc. Available <http://www2.sas.com/proceedings/sugi22/APPDEVEL/PAPER7.PDF>.
- Yarbrough, Kimberly D. 2000. “Macro Makes PROC MEANS Flexible.” *Proceedings of the Thirteenth Annual NorthEast SAS Users Group Conference* 311–312. Cary, NC: SAS Institute Inc. Available <http://www.lexjansen.com/nesug00/cc/cc4028.pdf>.
- Yee, Michael. 2001. “Quick and Dirty Data Laundering: A Scalable Solution for Range Checking Data.” *Proceedings of the Twenty-Sixth Annual SAS Users Group International Conference*. Cary, NC: SAS Institute Inc. Available <http://www2.sas.com/proceedings/sugi26/p099-26.pdf>.
- Yen, Peng-fang, and Jeff Gudmundson. 2000. “One Macro Does It All - Advanced Enhancement of PROC REPORT.” *Proceedings of the Twenty-Fifth Annual SAS Users Group International Conference* 189–194. Cary, NC: SAS Institute Inc. Available <http://www2.sas.com/proceedings/sugi25/25/ad/25p032.pdf>.
- Yindra, Chris. 1997. “&&&, ;, and Other Hieroglyphics: Advanced Macro Topics.” *Proceedings of the Twenty-Second Annual SAS Users Group International Conference* 242–250. Cary, NC: SAS Institute Inc. Also published in *Proceedings of the Eleventh Annual NorthEast SAS Users Group Conference* (1998) 124–132. Cary, NC: SAS Institute Inc. Available <http://www2.sas.com/proceedings/sugi22/ADVTUTOR/PAPER44.PDF>.
- Yindra, Chris. 1998. “%SYSFUNC: The Brave New Macro World.” *Proceedings of the Twenty-Third Annual SAS Users Group International Conference* 259–265. Cary, NC: SAS Institute Inc. Also published in *Proceedings of the Eleventh Annual NorthEast SAS Users Group Conference* (1998) 695–702. Cary, NC: SAS Institute Inc. Available <http://www2.sas.com/proceedings/sugi23/Advtutor/p44.pdf>.
- Yu, Hsiwei (Michael), and Gary Huang. 2003. “Return Code from Macro; Passing Parameter by Reference.” *Proceedings of the Twenty-Eighth Annual SAS Users Group International Conference*. Cary, NC: SAS Institute Inc. Available <http://www2.sas.com/proceedings/sugi28/080-28.pdf>.
- Yu, Hsiwei (Michael). 1998. “%SYSFUNC (Bridge to External Data).” *Proceedings of the Eleventh Annual NorthEast SAS Users Group Conference* 703–711. Cary, NC: SAS Institute Inc. Available <http://www.lexjansen.com/nesug/nesug98/solu/p035.pdf>.
- Yuan, Li, and Bill Zhang. 1999. “Presenting Multi-Level Information in One Plot by Dynamically Generating the PLOT Statement.” *Proceedings of the Twenty-Fourth Annual SAS Users Group International Conference* 613–616. Cary, NC: SAS Institute Inc. Available <http://www2.sas.com/proceedings/sugi24/Coders/p105-24.pdf>.
- Zdeb, Mike S. 1999a. “An Introduction to Macro Variables and Macro Programs.” *Proceedings of the Twelfth Annual NorthEast SAS Users Group Conference* 302–309. Cary, NC: SAS Institute Inc. Available <http://www.ats.ucla.edu/stat/sas/library/nesug99/bt108.pdf>.
- Zdeb, Mike S. 1999b. “Creating Macro Variables via PROC SQL.” *Proceedings of the Twelfth Annual NorthEast SAS Users Group Conference* 360–361. Cary, NC: SAS Institute Inc. Available <http://www.ats.ucla.edu/stat/sas/library/nesug99/cc107.pdf>.
- Zhang, John Q. 1998. “More about ‘INTO:Host-Variable’ in PROC SQL: Examples.” *Proceedings of the Eleventh Annual NorthEast SAS Users Group Conference* 354–355. Cary, NC: SAS Institute Inc. Available <http://www.ats.ucla.edu/stat/sas/library/nesug98/p193.pdf>.
- Zhang, Julia, David Chen, and Tor-Lai Wong. 2003. “Metadata Application on Clinical Trial Data in Drug Development.” *Proceedings of the Twenty-Eighth Annual SAS Users Group International Conference*. Cary, NC: SAS Institute Inc. Available <http://www2.sas.com/proceedings/sugi28/238-28.pdf>.
- Zhou, Jay. 2002. “%Qexcel Tackles the Challenges of Converting Data to SAS from Excel.” *Proceedings of the Annual Pharmaceutical SAS Users Group Conference* 65–70. Cary, NC: SAS Institute Inc. Available <http://www.pharmasug.org/psug2002/bp2002/ad15.pdf>.
- Zirbel, Doug. 2002. “Finally—An Easy Way to Compare Two SAS Files!” *Proceedings of the Twenty-Seventh Annual SAS Users Group International Conference*. Cary, NC: SAS Institute Inc. Available <http://www2.sas.com/proceedings/sugi27/p088-27.pdf>.

# Index

## Symbols

& (ampersand) 8, 10, 27–29, 36, 120, 146, 249–250, 397–399, 406  
- (dash) 136  
. (dot/period) 301, 400, 410  
= (equal sign) 18, 51  
# (pound sign) 69  
| (vertical bars) 271

## A

Abbott, David H. 312  
%ABC macro 63–64, 422  
%ABORT statement 84–85, 217, 218  
Aboutaleb, Hany 152, 154, 381  
Adams, John H. 427  
%AERPT macro 260  
Agbenyegah, Delali 261  
Ake, Christopher F. 306  
Alden, Kay 210  
%ALL macro 62  
\_all\_option 23  
Allen, Richard R. 382  
%ALLYR macro 74, 81  
ampersand (&) 8, 10, 27–29, 36, 120, 146, 249–250, 397–399, 406  
Andresen, Robert 382  
%ANOVA macro 62  
APPEND statement 337  
%APPLYFMT macro 370, 371  
arguments, calling macros with 233–236  
arithmetic operations, implied 415–416  
ARRAY statement 241  
arrays, macro  
    *See* macro arrays  
assignment statements 106–107  
asterisk-style comments 73, 79, 421–423  
attitude, characters with  
    *See* quoting functions  
attributes, in macro functions 177–178  
ATTRN function 240–241, 301, 302, 353  
auto display, of ODS styles 344–345  
autocall facility  
    options 42–44  
    using 261–265, 407  
autocall macros  
    about 180, 268–270  
    %CMPRES 269, 271–272  
    %CMY 275  
    %CNS 275

color conversions with 275–277  
%COLORMAC 275–277  
%COMPSTOR 266, 269, 274–275  
%DATATYP 269, 273–274  
%HELPCLR 275–277  
%HLS 275  
%HLS2RGB 276  
%HSV 275  
%KVERIFY 269, 270  
%LEFT 99, 143–144, 151, 159–161, 172, 176, 183, 248, 250, 251, 263–264, 269, 270–271, 411, 414  
%LOWCASE 151, 160–161, 264, 269, 272–273  
%QCMPRES 269, 271–272  
%QLEFT 135, 151, 159–160, 183, 250, 251, 269, 270–271  
%QLOWCASE 151, 160–161, 269, 272–273  
%QTRIM 151, 161–162, 269, 273  
%RGB 276  
%RGB2HLS 276  
%TRIM 269, 273, 411  
%VERIFY 151, 176, 263, 269, 270  
AUTOEXEC file  
    about 429–432  
    controlling 430  
    using 333–334  
automatic macro variables  
    *See also* macro variables  
    about 29–33, 214, 297  
    deciphering errors 216–220  
    defined 8  
    parameter buffer 220–223  
    &SYSFILRC 224–225  
    &SYSLIBRC 224–225  
    &SYSMACRONAME 33, 224  
    &SYSNOBS 223–224  
    &SYSPARM 214–216  
\_automatic\_option 23  
automatic SQL-generated macro variables 105  
automating, with macros 382  
AUTONAME option 329–330  
AXIS statement 293

## B

Battaglia, Michael P. 157, 176  
Benjamin, William E., Jr. 427  
Bercov, Mark 15, 135  
Bessler, LeRoy 381, 382  
best practices 409–411  
%BESTEVER macro 233

- Beverly, Bryan K. 32, 76  
 Billings, Thomas E. 32, 216  
 Birkenmaier, Richard 122  
 Blair, Kimberly S. 118  
 blind quotes  
   *See* %BQUOTE function  
 Blood, Nancy K. 282, 307  
 Borgerding, Joleen 176  
 %BQUOTE function 134–137, 141, 144, 160, 184,  
   249, 251, 340, 406  
 Bramley, Michael P.D. 176, 302, 360  
 branching program flow  
   *See* program flow  
 %BREAKUP macro 303  
 bridging functions 132  
 Brooks, Lisa K. 42  
 Bryant, Connie 44  
 Bryher, Monique 92, 400  
 Bucheker, M. Michelle 297  
 building blocks, using macro variables as 27–28  
 %BUILDMATRIX macro 401  
 %BUILDVARLIST macro 358–360  
 Burger Thomas H. 409  
 Burlew, Michele M. 8, 12, 135, 176, 254, 283  
 Burnett-Isaacs, Kate 171, 309  
 Burroughs, Scott 360  
 BY statement 345  
 BY variables 47–48, 65, 157  
 &BYLIST macro 59
- C**
- CALL DEFINE routine 388  
 CALL EXECUTE routine 65, 121–128, 205, 206,  
   226–228, 291, 295, 313–317  
 CALL SYMDEL routine 228–229  
 CALL SYMPUTX routine 307  
 CALL VNAME routine 355–356  
 Callahan, Janice D. 118, 282, 304  
 capitalization 408  
 CARDS statement 377  
 Carey, Ginger 44, 256  
 Carey, Helen 44, 256  
 Carpenter, Arthur L. 4, 15, 38, 44, 118, 132, 135,  
   140, 146, 180, 226, 248, 249, 254, 256,  
   260, 282, 286, 291, 304, 306, 319, 331,  
   332, 334, 345, 382, 389, 405, 408, 411,  
   412, 415, 421, 430, 432  
*Carpenter's Guide to Innovative SAS® Techniques*  
   (Carpenter) 233  
 Carter, James R. 122, 297  
 Casas, Angelina Cecilia 98, 104, 295  
 CAT function 306  
 catalogs, copying unkown numbers of 336  
 CATALOGS table 296  
 %CATCOPY macro 336  
 CATT function 92, 104, 240, 306, 315  
 CATX function 116  
 Chai, Akiko 176  
 Chakravarthy, Venky 342  
 Chapman, David D. 393  
 chapter exercises  
   data set values 129–130  
   macro functions 193  
   macro variables 33–34  
   macros 44, 51–52, 86–87  
 character variable 91  
 charts, controlling  
   *See* Output Delivery System (ODS)  
 %CHECK macro 152  
 %CHECKIT macro 424–425  
 %CHECKPROD macro 202  
 Chen, Babai 77, 292  
 Chen, Chang-Min 176, 296, 342, 378  
 Chen, David 117  
 Chen, Ling Y. 341, 381  
 Chen, X. Hong 382  
 Cheng, Alice M. 125, 145, 220, 409  
 Cheng, Wei 382  
 %CHKDIR macro 383–384, 386  
 %CHKSRCOPY macro 258  
 %CHKSURVEY macro 420  
 %CHKWT macro 168  
 Chow, Ming H. 122, 317  
 Chu, Clara 77, 292  
 Chung, Chang Y. 137, 147, 245, 415  
 %CLINICRPT macro 388–389  
 %CLINRPT macro 384–385  
 CLOSE function 301  
 CMDMAC option 407  
 %CMPRES macro 269, 271–272  
 %CMY macro 275  
 %CMYK macro 275  
 %CNS macro 275  
 %CNTMALES macro 429  
 %CNTVAR macro 155–156, 166  
 code  
   *See also* dynamic programming  
   building dynamically 66–69  
   commenting blocks of 37  
   substitution of 5–6  
 Cohen, Barry R. 210, 212, 335  
 Cohen, John J. 35  
 collisions, macro variable 419–420  
 %COLONCMPR macro 190  
 color conversions, with autocall macros 275–277  
 %COLORMAC macro 275–277  
 column indicators, naming 400–402  
 COLUMNS table 296  
 COMB function 182–183  
 command-style macros 406  
 commas, placing between words 364–365  
 %COMMENT macro 38  
 community forums (website) 245  
 COMPARE procedure 219–220

- compiled stored macros 407
- COMPRESS** function 303–304
- %COMPSTOR macro 266, 269, 274–275
- conditional execution 64–70
- Conley, Brian 302
- consistency 408
- constant text 36
- CONTENTS** procedure 19–21, 20, 37, 102, 111–112, 117, 296, 300, 301–302, 330, 338–339, 367–368, 396
- control files
  - building macro variable lists using 321–322
  - controlling and using 304–306
  - creating data validation checks dynamically using 324–327
  - creating empty data sets using 322–324
  - CSV 369–370
  - setting up for projects 319–321
- %COPY statement 258, 259
- %COPYALL macro 216–217
- %COPYRTE macro 274
- %CORREL macro 62
- COUNT function 98, 104
- %COUNTCLASS macro 361, 363
- %COUNTW function 222, 313, 361–362, 363
- Crawford, Peter 44
- Croonen, Nancy 122
- %CSTR function 364, 365–366
- CSV control file 369–370
- Cunningham, Gary 210, 212, 335
- %CURRDATE macro 379
  
- D**
- dash (-) 136
- data
  - controlling corrections and manipulations 369–371
  - controlling program flow with 116–121
  - controlling programs with 290–291
  - creating independence 289
  - working with 389–396
  - writing applications without hardcoded dependencies 317–327
- data dictionaries
  - See* control files
- data set values
  - about 89
  - chapter exercises 129–130
  - creating macro variables using SYMPUTX routine 90–98
- data sets
  - appending unknown 336–342
  - building formats from 349–350
  - building lists from 238–239
  - creating empty 322–324
  - splitting vertically 352–353
  - stepping through lists of 309
- using metadata 300–302
- working with 351–371
- DATA** statement 292
- DATA** step
  - code *versus* macro language 412–417
  - debugger 21
  - function 297–300
  - functions and statements 226–230, 341–342, 409
  - I/O functions 300–301
  - macro functions for 190–193
  - tools 11
  - using functions and routines 169–176
  - using functions to retrieve variable names 355–356
- data structures 331
- data tables, controlling processes with 303–304
- data validation checks, creating dynamically 324–327
- DATA\_NULL\_Step** 170–171, 353–355
- DATASETS** procedure 31, 174, 216
- %DATATYP macro 269, 273–274
- dates
  - converting 171
  - incrementing 191–192
  - specifying in titles 171–172
  - working with 183
- %DATSERNUM macro 419–420
- Davis, Michael 295
- Davis, Neil 32, 256
- DBDIR data set 319
- DCLOSE function 175
- DCREATE function 383–384
- %DEBUG macro 20–21, 232
- debugging macros 411–412
- debugging options 41
- %DEBUGNEW macro 38, 39, 232
- %DEF macro 63–64
- %DELFILE macro 172–173
- delimiters, using with %SCAN function 154–156
- %DELVARS macro 205, 206
- %DEMO macro 220–221
- /DES macro statement using 39
- DESCRIBE statement 296
- di Tommaso, Dante 206
- DICTIONARY.TABLES 338
- Dilorio, Frank C. 226
- directories
  - controlling 382–384
  - structure of 330–332
  - working with 350–351
- Display Manager, calling macros from 233–236
- %DISPLAY statement 146, 208–211
- %DISTINCTLIST macro 367–368
- DLLs (Dynamic Link Libraries) 336, 342
- DNUM function 175
- %DO block 68, 70–73, 138, 290

%DO loops 22, 73–76, 103, 118, 288–289, 291–293, 297, 306, 308–309, 311–312, 323, 326–328, 341, 345, 353, 370, 386, 387–388, 401–403, 420, 428  
 %DO statements 80–81, 166  
 %DO %UNTIL loops 76–77, 156, 291  
 %DO %WHILE loops 77–78, 189, 291, 312–313, 364, 368, 416  
 %DOBOTH macro 57–61, 65, 68, 72, 260  
 documentation 408  
 %DOIT macro 62–63, 76, 80, 125–126, 146, 422–423  
 DOPEN function 175  
 DOSUBL function 128, 226–228  
 dot(.) 301, 400, 410  
 double quotes 66  
 doubly scripted macro arrays 399–405  
 DREAD function 175  
 DROP statement 136  
 Drummond, Derek 42, 412  
 &DSET macro 59  
 %DSNPROMPT macro 209–210  
 %DUMPIT macro 376–377  
 Dynamic Link Libraries (DLLs) 336, 342  
 dynamic macro coding techniques 279  
 dynamic programming  
     about 7, 282, 335  
     adapting SAS environment 346–351  
     building SAS statements 327–328  
     controlling output 342–346  
     data sets 351–371  
     design elements 282–293  
     directory structure 330–332  
     file management 335–342  
     horizontal lists 309–313  
     information sources 293–307  
     naming conventions 328–330  
     unifying *fileref* and *libref* definitions 334  
     using AUTOEXEC file 333–334  
     using CALL EXECUTE 313–315  
     &&VAR&I constructs as vertical macro arrays 307–309  
     variables 351–371  
     writing applications without hardcoded data dependencies 317–327  
         writing %INCLUDE programs 315–317  
 Dynder, Andrea 210, 212, 335

**E**

Eberhardt, Peter 430  
 Eddlestone, Mary-Elizabeth 98, 104  
 Edgington, Jim 42  
 %ELSE statements 67, 179  
     *See also* %IF-%THEN-%ELSE statement  
 END= option 92, 354  
 %END statement 327

environments  
     adapting 346–351  
     referencing 12–15  
 %EOW macro 193  
 equal sign (=) 18, 51  
 errors and troubleshooting 216–220  
     *See also* debugging macros  
 %EVAL function 18, 69, 70, 77, 78, 136, 156, 162–166, 184, 416, 424, 426  
 evaluation functions  
     about 132, 162  
     %EVAL 18, 69, 70, 77, 78, 136, 156, 162–166, 184, 416, 424, 426  
     %SYSEVALF 147, 148, 166–169, 171, 186, 188, 245, 390  
 event sequencing 8–12  
 Ewing, Daphne 381  
 %EXIST macro 137–138, 177, 178–179, 180, 182  
 expressions  
     defined 65  
     evaluating 245–246  
 EXTFILES table 296

**F**

FACT Function 182–183  
 factorials, calculating 182–183  
 Fahmy, Adel 210  
 FDELETE function 173, 176  
 Fehd, Ronald 286, 291, 307, 409, 430  
 Felty, Kelly 382  
 Ferriola, Frank 304  
 FETCH function 238, 240–241, 301  
 FETCHOBS function 240–241, 301  
 FEXIST function 173, 176  
 file management  
     about 335–336  
     appending unknown data sets 336–342  
     copying unknown numbers of catalogs 336  
 FILE statement 315, 345  
 FILEEXIST function 173, 176, 211–212, 259, 351, 382–384  
 FILENAME statement 42, 145, 173, 175, 197, 224–225, 255, 277, 305, 333, 340, 341, 350, 376–377, 382–384  
 fileref 197, 334  
 FILEREF function 350  
 files, deleting using %SYSFUNC function 172–173  
 %FINDOUTLIERS macro 395  
 First, Steven 29, 35, 98, 104, 135, 254  
 %FIXRAW macro 327  
 flat files 331  
     *See also* data sets  
 Flavin, Justina M. 38, 405  
 FLDDIR data set 319, 320–321  
 FMTSEARCH option 347–349  
 %FMTSRCH macro 349–350

FOOTNOTE statement 8, 198  
 footnotes, coordinating 342–344  
 FORMAT procedure 101, 243–244, 293, 347–350  
 formats, building and maintaining 347–350  
 Frankel, David S. 42, 412  
 FREQ procedure 118, 270  
 Friendly, Michael  
*SAS System for Statistical Graphics, First Edition* 373  
 FSEDIT procedure 307, 309  
 Function keys, adding macro calls to 233  
 functions  
*See* macro functions  
 %FUZZRNGE function 184

**G**

Gau, Linda C. 210  
 Geary, Hugh 335, 400  
 Gerlach, John R. 307, 390, 398  
 %GETAUTOPATH macro 181, 264  
 %GETKEYS macro 398–399  
 GETOPTION function 176, 216, 298–300, 348  
 %GETVARS macro 356–360  
 Gilbert, Steven A. 341, 381  
 Gilmore, Jodie 42, 412  
 Glass, Roberta 32, 211, 300  
 global macro variables 12, 19, 409–410  
 \_global\_option 23  
 %GLOBAL statement 8, 15, 81–84, 213, 410, 427, 432  
 global symbol tables, saving 431  
 %GLOBALRETRIEVE macro 431, 432  
 %GLOBALSAVE macro 431  
 Gober, John Charles 304  
 Goddard, Jonathan R. 295, 382  
 Goldstein, Leanne 19  
 Gondara, Lovedeep 148  
 GOPTIONS statement 347  
 %GOTO statement 206–208, 291, 409, 428  
 %GRABPATH macro 198, 199  
 Graebner, Robert W. 317  
 Grant, Paul 38  
 graphs, controlling  
*See* Output Delivery System (ODS)  
 Greathouse, Matt 264  
 Guan, Yun 360  
 Gunshenan, Michael 297

**H**

Hadden, Louise 32, 211, 293, 389  
 Hahl, Thomas J. 352  
 Hamilton, Jack 122, 176, 180, 295, 302  
 "hanging" semicolon 68  
 Hayden, Vanessa 381  
 header text 408  
 Heaton, Edward 42, 56, 409, 412

Heaton-Wright, Lawrence 122, 342  
 %HELPCLR macro 275–277  
 Helwig, Linda 42, 412  
 Henderson, Don 228, 353, 408  
 Henry, Joseph 85  
 Hessel, Colin 32  
 %HIGHER macro 181  
 Hirabayashi, Sharon Matsumoto 270  
 %HLS macro 275  
 %HLS2RGB macro 276  
 Hoaglin, David C. 157, 176  
 %HOLDOPT macro 298–300  
 Holland, Philip R. 282  
 horizontal lists 285–286, 309–313, 360–364  
 Howell, Andrew 135  
 %HSV macro 275  
 Huang, Liping 210, 296  
 Hubbell, Katie A. 15  
 Hughes, Troy Martin 85, 123, 125, 216, 219, 300, 371  
 hyperlinks, controlling 384–389

**I**

%IF statement 11–12, 66, 67, 190, 221, 244, 292, 323–324, 326, 413, 414, 425  
 %IF-%THEN statement 179  
 %IF-%THEN-%ELSE statement 64–70, 290, 292, 395  
 IMPLMAC option 407  
 IMPORT procedure 305, 319, 370  
 IN comparison operator 69–70  
 %INCLUDE statement 254–255, 266, 315–317, 354–355, 380–381, 406, 431–432  
 indentation 407  
 index, creating reports as an 385–387  
 %INDEX function 151, 152, 186–187, 271  
 INDEXES table 296  
 INDEXW function 186–187, 368  
 INFILE statement 340  
 information sources  
 about 293  
 automatic macro variables 297  
 building macro variables based on 287–288  
 control files 304–306  
 controlling processes with data tables 303–304  
 DATA step functions 297–300  
 retrieving operating system information 300  
 SASHELP views 293–296  
 SET statement options 306–307  
 SQL DICTIONARY tables 296–297  
 %SYSFUNC function 297–300  
 using data set metadata 300–302  
 &INFUNC macro variable 222  
 &ININSIDE macro 419  
 initialization 431–432  
 &INLIST macro variable 70  
 %INNER macro 14

INPUT function 110, 244  
%INPUT macro 146, 378  
INPUTN function 171  
%INSIDE macro 419  
INTNX function 191–192  
INTO: operator 429  
I/O functions 300–301  
%ISITQUOTED macro 141–142  
iterative %DO loops 73–76  
iterative step execution 291  
Izrael, David 157, 176

## **J**

Jaffe, Jay A. 12, 35, 115, 122  
Jensen, Karl 264  
Jia, Justin 212, 384  
Jiang, Jonson C. 122, 315  
Jin, Jiang 122  
Jin, Ye 122  
Johnson, Jim or Martha 216, 317, 342

## **K**

Kahle, Eric E. 98, 104  
KEEP= option 159, 353, 366–368  
Kelley, Francis J. 342  
Kelly, Timothy A. 211, 295  
Kenney, Tim 175  
keyword (named) parameters  
    about 46, 408, 409  
    choosing between positional parameters and 50–51  
    defined 46  
    naming without equal sign 51  
    using 48–50  
King, John 137, 147, 245, 415  
Knowlton Bill 381  
Kochanski, Mark A. 412  
Kraemer, Helena Chmura 400  
Krenzke, Tom 317  
Kretzman, Peter 302  
Kunselman, Thomas E. 307, 404  
%KVERIFY macro 269, 270

## **L**

%LABEL statement 206–208  
LABEL statement 323  
Lafler, Kirk Paul 4, 295, 412  
Landers, K. Larry 92  
Langston, Richard D. 173, 196, 200, 206, 213, 384  
Larsen, Erik S. 180, 382  
layered symbol tables 427  
LeBouton, Kimberly J. 297, 342  
%LEFT macro 99, 143–144, 151, 159–161, 172, 176, 183, 248, 250, 251, 263–264, 269–271, 411, 414

&LEFTLIST macro variable 143–144  
Leighton, Ralph W. 17, 69, 76  
%LENGTH function 151, 153–154, 157, 189, 215, 244, 273, 323–324  
Leprince, Daniel J. 382  
%LET statement 4–6, 11–12, 18–19, 20, 37, 45, 46, 55, 65, 90, 119–120, 133, 135–136, 143, 160, 162, 216, 238, 271, 354, 357, 415, 419, 421–422, 425–427, 428  
Letourneau, Kent 210, 295, 315  
Levin, Lois 199, 409  
Levine, Howard 409  
Li, Arthur X. 12, 17  
Li, Elizabeth 382  
Liang, Shuhua 248  
LIBNAME statement 123, 197, 214, 218–219, 225, 257, 333, 350, 351, 384  
%LIBNAMES macro 333, 334  
libraries 350–351  
    *See also* macro libraries

libref 197, 334  
LIBREF function 350, 351  
LINK= option 388, 389  
list processing 291  
LIST statement 377  
%LISTDSN macro 310–311  
%LISTLAST macro 186  
%LISTLINES macro 377–378  
lists  
    about 251  
    building 364  
    horizontal 285–286, 309–313, 360–364  
    quoting words in 365–366  
    removing repeated words from 367–368  
    vertical 283–286  
%LISTSAS macro 277  
Litzsinger, Michael A. 42  
local macro variables 13, 178, 409–410  
    \_local\_option 23  
%LOCAL statement 15, 81–84, 178, 213, 239, 321–322, 409, 420, 427, 432  
%LOCATE macro 55, 56  
logic, macro functions with 187–190  
logical expressions, building 184  
logical program flow  
    *See* program flow

Long, Ying 341  
%LOOK macro 37, 39, 46–51, 55–61, 72, 117, 120  
Lopez, Roberto 350  
Lopez, Victor A. 381  
Lougee, Claudine 4  
%LOWCASE macro 151, 160–161, 264, 269, 272–273  
Lund, Pete 157, 176, 180, 184, 186, 190, 199, 208, 212, 223, 277, 300, 350, 360, 381, 382  
Luo, Haining 318  
Luo, Haiping 318

**M**

- Mace, Michael A. 210, 223  
%MACEXEC macro 200–201  
macro arrays  
  doubly scripted 399–405  
  selecting elements from 398–399  
macro Booleans 410  
macro calls  
  adding to Function keys 233  
  building 231–236  
  commenting 40  
  controlling 62–63  
  passing parameters through 58–61  
  resolving 178–180  
  unresolved 411  
macro code  
  about 37  
  executing 123–125  
  executing using CALL EXECUTE routine 121–128  
  length of 410  
macro comments 73, 79–81  
macro execution, termination of 84–85  
macro expression 8, 36  
Macro Facility  
  about 3–4  
  defined 7  
  using system options with 40–44  
macro functions  
*See also specific macro functions*  
  about 6, 36–37, 132, 196  
  attributes 177–178  
  building 176–181  
  chapter exercises 193  
  DATA step 190–193, 226–230  
  DATA step functions/routines 169–176  
  defined 8  
  deleting 212–213  
  evaluation functions 162–169  
  loading macro variable lists directly using 240–241  
  with logic 187–190  
  macro variable scopes 203  
  mixing 414  
  pulling variable names using 356–358  
  quoting functions 132–150  
  text functions 150–162  
  user-written 182–193  
  using DATA step functions and routines 169–176  
macro language  
  about 3–4, 195–196  
  automatic macro variables 214–225  
  DATA step code *versus* 412–417  
  DATA step functions/statements 226–230  
  efficiency and 405–406  
  elements of 6  
for formatted table lookups 243–244  
forming simple hash tables using 241–243  
functions 196–203  
macro statements 204–214  
outstanding recursion in 425–427  
on remote servers 246–248  
stages of learning 5  
system options 225–226  
tokens 283  
using quote marks in 415  
macro libraries  
  about 253–254, 410–411  
  autocall macros 268–278  
  establishing 254  
  interactive macro development 266–267  
  modifying SASAUTOS system variable 267–268  
  search order 265–266  
  structure and strategy 266  
  using autocall facility 261–265  
  using %INCLUDE statement as 254–255  
  using stored compiled 256–261  
macro lists, creating 384–385  
MACRO option 41, 407  
macro parameters 45–46, 427  
macro program statements  
  about 36  
  additional 78–85  
  defined 7  
macro programming best practices 409–411  
macro quoting 406  
macro references  
  about 8  
  defined 7  
  resolving 8  
  unresolved 28–29  
%MACRO statement 35–39, 45, 46, 48–50, 63–64, 220, 255, 257, 258, 261–265, 267, 277, 406, 421  
macro statements  
*See also specific macro statements*  
  about 6, 204  
  building dynamically 327–329  
  building 291–293  
  conditional 412  
  DATA step 226–230  
  executing 65–66  
  iterative execution of 70–78  
  %label 206–208  
  READONLY options 213–214, 410  
  using 178  
macro system options 406–407  
macro triggers 7  
macro variable references  
*See macro references*  
macro variables  
*See also automatic macro variables*

- about 12, 17
- appending 27–28
- assigning names 119–121
- assigning values 117–118
- automatic 29–33, 297
- automatic SQL-generated 105
- building based on information sources 287–288
- building lists of 96–98, 308
- chapter exercises 33–34
- collisions 419–420
- created in SQL procedure 429
- created with %DO 428
- created with %LET 428
- created with SYMPUT routine 428–429
- created with SYMPUTX routine 90–98, 428–429
- creating 238–241, 416–417, 428–429
- defined 7
- defining 18–19, 98–105
- defining in SQL procedure steps 98–105
- deleting 204–205, 228–229
- determining scopes 427–429
- displaying using %PUT statement 21–24
- global 12, 19, 409–410
- loading lists directly using macro functions 240–241
- local 13, 178, 409–410
- moving text from 106–116
- naming 18
- placing lists of values into series of 102–104
- placing single values into single 98–99
- removing 33
- resolving 24–29, 283
- special characters and 248–251
- storing system clock values in 378–379
- triple ampersand 397–399
- using 19–21, 95–96, 410
- using as a prefix 26–27
- using as a suffix 25–26
- using as building blocks 27–28
- using DATA step variable names as 239–240
- using dynamically 288–289
- working with 236–241
- in wrong symbol table 417–419
- macros
  - See also specific macros*
  - about 35
  - asterisk-style comments in 421–423
  - autocall facility options 42–44
  - automating with 382
  - calling from Display Manager 233–236
  - calling with arguments 233–236
  - chapter exercises 44, 51–52, 86–87
  - controlling execution of 432
  - controlling programs with 55–87
  - creating 35–39
  - debugging 411–412
  - defined 8
  - defining 37
  - documenting 49–50
  - general options 41
  - invoking 39–40
  - invoking macros with 55–64
  - masking special characters inside 140
  - nesting definitions 63–64
  - options 41–42
  - passing parameter values into 46–48
  - passing parameter values when calling 48–49
  - passing parameters between 55–56
  - passing parameters when macros call 56–57
  - protecting 432
  - syntax for 423–424
  - that perform change 371
  - user-written 182–193
- MACROS table 296
- MAKE\_C 370
- MAKE\_CASE 370
- %MAKECSV macro 394–396
- %MAKEDIR macro 211–212, 351
- MAKE\_N 369
- %MAKERUNBAT macro 379–381
- %MAKEVARS macro 362–364
- Maldonado, Miguel 170
- Mao, Cailiang 342
- Mao, Sam 176, 189, 297
- masking characters 145–146, 184
- Mason, Phil 148, 203, 285
- Mast, Greg 335
- Matise, Joe 28, 398, 400
- %MATRIXPRINT macro 403–404
- MAUTOCOMPLOC option 262–263
- MAUTOLOCDISPLAY option 262, 263
- MAUTOLOCINDES option 262, 263–264
- MAUTOSOURCE option 43, 407
- MAX function 295
- McMullen, Quentin 224
- MCOMPILENOTE option 265
- %MDARRAY macro 274
- MEANS procedure 6, 317
- MEMBERS table 296
- memory control options 225–226
- %MEND statement 35–39, 63–64, 255, 258, 261–265, 267, 421, 423–424
- MERROR option 41
- metadata (control data sets)
  - See* control files
- &METHOD macro variable 135–136
- MFILE option 41, 42
- Michel, Denis 315
- Michelsen, Jesper 32, 241, 351
- Millard, Scott 318
- MINDELIMITER option 69
- Miralles, Romain 317
- Misra, Simant 307, 390

missing values, compared with null values 414–415  
 %MKFMT 350  
 %MKLIB macro 350–351  
 MLOGIC option 41, 407, 411, 423–424  
 MLOGICNEST option 265  
 %MODFEM macro 208  
 modifiers 156–157  
 Molter, Michael 318  
 Moors, David 307  
 Moriak, Chris 248  
 Morrill, John 317  
 Mounib, Edgar L. 122  
 MPRINT option 41, 79, 407, 411, 423–424  
 MPRINTNEST option 265  
 MRECALL option 43, 407  
 MSTORED option 256, 407  
 MSYMTABMAX option 226  
 Muller, Roger D. 254, 282  
*Multiple-Plot Displays: Simplified with Macros (Watts)* 373  
 Murphy, William C. 176, 241, 342  
 MVARSIZE option 225

**N**

%&name 231–232  
 named parameters  
*See keyword (named) parameters*  
 names, assigning to macro variables 119–121  
 naming conventions 328–330, 408  
 naming macro variables 18  
 %NBRQUOTE function 134–135  
 nested functions 251  
 nested macro definitions 406, 410  
 nested symbol tables 13–15, 427  
 nesting macro definitions 63–64  
 Nicholson, Diane 382  
 NOBS option, SET statement 306–307, 390–391, 392  
 Noda, Art 400  
 NOMCOMPILE option 226  
 NOMFILE option 42  
 NOMPRINT option 42  
 non-integer comparisons 424  
 non-macro code, executing 122–123  
 NOPRINT option, SQL procedure 343  
 NR functions 142–143  
 %NRBQUOTE function 142–143, 145  
 %NRQUOTE function 135, 145  
 %NRSTR function 128, 133, 134–135, 142–143, 145–146, 205, 232, 363, 406  
 null values  
   compared with missing values 414–415  
   making comparisons to 244–245  
 number systems, converting 187  
 numbers, rounding 175–176  
 numeric range comparisons 424–425

**O**

%OBSCNT macro 85, 224, 300–301, 302, 322, 392  
 observations  
*See data sets*  
 O'Connor, Susan M. 12, 42, 44, 135, 256, 412  
 ODS  
*See Output Delivery System (ODS)*  
 Olaleye, David 176  
 open code 8  
 OPEN function 300–301  
 operating systems  
   retrieving information 300  
   working with 375–381  
 operators 8  
 options  
   about 6  
   autocall facility 42–44  
   debugging 41  
   SET statement 306–307  
   used with macro libraries 265  
 OPTIONS statement 56  
 OPTIONS statement 8, 42, 256, 348  
*See also system options*

OPTIONS table 296  
 OPTLOAD procedure 347  
 OPTSAVE procedure 347  
 %ORLIST macro 222  
 Ortiz, Lorena 309  
 Ottesen, Rebecca 19  
 OUT= option 258, 301–302  
 %OUTER macro 14  
 output, controlling 342–346  
 Output Delivery System (ODS)  
   about 342  
   auto display of styles 344–345  
   consolidating OUTPUT destination data sets 345–346  
   working with 381–389  
 OUTPUT destination data sets, consolidating 345–346  
 OUTPUT statement 345–346

**P**

Paciocco, Steve 318  
 Pahmer, Emmy 121, 199  
 Palmer, Lynn 91, 176  
 parameter buffer 220–223  
 parameters  
*See also keyword (named) parameters*  
*See also macro parameters*  
*See also positional parameters*  
 passing between macros 55–56  
 passing through macro calls 58–61  
 passing when macros call macros 56–57  
 types of 50  
 Parker, Chevell 228, 241

- Parker, Peter 210  
 parsed language 9  
 Pass, Ray 118  
 passing values, quoting before/after 139–140  
 PATHNAME function 174, 176, 275, 277, 300, 350, 351  
 %PATTERN macro 174–175  
 PATTERN statements, generating using PUTN function 174–175  
 PDV (Program Data Vector) 109  
 percent sign (%) 5–6, 8, 10, 36, 137, 249–250, 406  
 period(.) 301, 400, 410  
 Periyakoil, Vyjayanthi S. 400  
 Perl Regular Expressions, matching variable names to patterns using 358–360  
 %PERM function 182–183, 187–188  
 permutations, calculating 187–188  
 persistence  
   *See scopes*  
 Peszek, Iza 44  
 Peterson, Donald W. 382  
 Phillips, Jeff 42, 412  
 Pierri, Francesca 381, 382  
 Piet, John M. 109  
 PIPE device type 340  
 %PLACEIT macro 94  
 Plath, Robert 210  
 Pochon, Philip M. 409  
 POINT option, SET statement 390–391, 392  
 positional parameters  
   about 409  
   choosing between keyword parameters and 50–51  
   defined 46  
   using 46–48  
 pound sign (#) 69  
 %PRECOMP macro 55–56  
 prefix, using macro variables as a 26–27  
 PREFIX= option 354  
 Price, Jennifer 44  
 %PRIMARY 420  
 PRINT procedure 4, 6, 7, 9, 13, 19–21, 37, 39, 72, 86, 117, 123, 126, 132–133, 139, 159, 264, 378, 385–387, 402–404, 414  
 %PRINTIT macro 284–285, 342–344  
 %PRINTT macro 86  
 program control, with macros 55–87  
 Program Data Vector (PDV) 109  
 program flow, controlling with data 116–121  
 programming  
   *See also* dynamic programming  
   efficiency issues with 405–407  
   organizing 408  
   structure 408–409  
   style and 407–409  
 programs  
   controlling with data 290–291  
   executing series of 379–381  
 project structure 331–332  
 %PRTCLASS macro 311  
 %PRTDSN macro 123  
 PUT function 68, 91, 171, 244  
 %PUT statement 21–24, 37, 100, 124, 127–128, 142–143, 146, 157, 163, 220, 260, 288, 297, 301, 302, 315–317, 322, 402, 403, 411, 417–418, 422, 425–427  
 PUTC function 175, 243–244  
 PUTN function 171, 174–175, 187
- Q**
- %QCHARVAR macro 366  
 %QCMPRES macro 269, 271–272  
 %QLEFT macro 135, 151, 159–160, 183, 250, 251, 269, 270–271  
 %QLOWCASE macro 151, 160–161, 269, 272–273  
 %QSCAN function 151, 154–157, 176, 291, 310–311, 312, 361, 364, 404  
 %QSTR function 365–366  
 %QSUBSTR function 135, 151, 153, 157–158, 190, 198, 271, 273  
 %QSYSFUNC function 135, 170–173, 378  
 %QTRIM macro 151, 161–162, 269, 273  
 quotation marks, invisible 140–141  
 %QUOTE function 134–135, 145, 149–150, 366, 381  
 quotes  
   about 248  
   problems with 248–249  
   using in macro language 415  
 quoting functions  
   about 132–135, 148–149  
   %BQUOTE 134–137, 141, 144, 160, 184, 249, 251, 340, 406  
   considerations for 137–142  
   history of 134  
   %NRQUOTE 135, 145  
   %QUOTE 134–135, 145, 149–150, 366, 381  
   removing masking characters 145–146  
   %STR 133–135, 137–138, 141–145, 149–150, 156, 184, 245, 423–424  
   %SUPERQ 134–135, 146–148, 245, 250  
   types of 142–145  
 %QUPCASE function 135, 151, 158–159
- R**
- Rajecki, Aldona A. 98, 104  
 %RAND\_W macro 392  
 RANUNI function 170, 391  
 Rasheed, Harun 32  
 Ravi, Prasad 297  
 Reading, Pamela L. 317  
 %READNEW macro 75–76  
 READONLY options 213–214, 410

- recursion, in macro language 425–427  
referencing environments 12–15  
%REGINDEX macro 385–387  
%REGIONRPT macro 118, 279  
%REGRPT macro 387–388  
Ren, Quan 210  
%REPEAT function 189–190  
repeatability, generalized and controlled 318–319  
REPORT procedure 233, 317, 382, 388, 389  
RESOLVE function 96, 109–116, 245  
resolving macro variables 24–29  
RETAIN statement 106–107, 324  
%RETURN statement 85, 208  
REVERSE function 186  
%REVSCAN function 187, 189  
%RGB macro 276  
%RGB2HLS macro 276  
%RGBHEX macro 187  
Rhoades, Stephen 427  
Rhoads, Amy 210, 295, 315  
Riba, S. 21  
Rice, Thomas W. 297, 342  
Riddle, Michael A. 42  
Roberge, Sylvianne B. 98  
Roberts, Clark 78, 157  
Rook, Christopher J. 122, 176, 336, 342  
Roper, Christopher A. 336, 342  
Rosenbloom, Mary F.O. 135, 140, 146, 248, 249, 286  
ROUND function 187  
rounding numbers 175–176  
row indicators, naming 400–402  
RUN statement 218  
%RUNCHECK macro 218  
RXMATCH function 360
- S**
- SAS System for Statistical Graphics, First Edition* (Friendly) 373  
SASAUTO= option 43, 262, 266  
SASAUTOS option 267–268, 407  
sasCommunity (website) 206, 373  
SASHelp views 293–296  
SASHelp.VTABLE 337–338  
SASMSTORE= option 256, 257, 260–261, 274, 407  
Satchi, Thiru 98, 104, 122  
Sattler, Jim 318  
%SCAN function 151, 154–157, 186, 188–189, 291, 310–311, 361, 362, 367, 404–405, 415–416, 423–424  
%SCHOEN2 macro 49–50  
scopes  
  about 12–15  
  determining for macro variables 236–238  
  macro variable 203  
  of macro variables 427–429  
SECURE option 260
- SELECT statement 98, 100, 105, 328, 336  
semicolons 408  
%SENRATE macro 67, 71  
sequencing events 8–12  
SError option 41  
session compiled macros 407  
SET statement 66–67, 70–71, 75–76, 80–81, 92, 138, 289, 291–292, 301, 306–307, 327–328, 337, 390–391, 392  
%SETUP macro 62  
Shen, Yanyun 318  
Shi, Changhong 98  
Shilling, Brian C. 211  
%SHOWMACNEST macro 199–200  
%SHOWRPT macro 233–236  
Shtern, Elena 217  
Sissing, Lori 19  
Sisson, Emily K.Q. 125, 340  
%SLEEP function 169, 175, 185–186  
Smith, Curtis 396  
Smith, Richard O. 44, 256, 282, 293, 319, 331, 332, 334, 382, 389  
SORT procedure 59, 61, 72  
%SORTIT macro 47–48, 56–61, 65, 72  
SOURCE option 256  
special characters, masking inside macros 140  
Spicer, Jeanne 295  
%SPLIT macro 307, 352–353  
%SPLITUP macro 304  
Spotts, Bruce 381  
SQL COUNT function 98  
SQL DICTIONARY tables 296–297  
SQL procedure 29, 97–105, 116, 161, 206, 219, 296–297, 308, 310, 312, 343, 389–396, 409, 429
- SQL step  
  creating lists of variables using 356  
  quoting words in a 366  
SQLEXITCODE 105  
&SQLOBS macro variables 102, 103, 105  
SQLLOOPS 105  
&SQLRC macro variable 105, 219  
SQLXMSG 105  
SQLXOBS 105  
SQLXRC 105  
Squire, Jonathan 381  
statements  
  See macro statements  
statement-style macros 406  
Staum, Roger 412  
%STCODES macro 140, 141  
Stokke, Delayne 176  
STOP statement 138, 177, 307, 324  
STORE option 260  
%STOREOPT macro 346–347  
%STR function 133–135, 137–138, 141–145, 149–150, 156, 184, 245, 423–424

- Staelnner, Janet E. 38  
 STYLE= option 345  
 subscript resolution 400  
 %SUBSTR function 151, 157–158, 176, 271, 272, 273  
 suffix, using macro variables as a 25–26  
 Suhr, Diana D. 38  
 SUM function 402, 415–416  
 SUMMARY procedure 104, 289, 329–330  
 Sun, Eric 226, 260, 411, 432  
 Sun, Jeff F. 402  
 %SUPERQ function 134–135, 146–148, 245, 250  
 SURVEYSELECT procedure 391, 393  
 %SURVIVAL macro 306, 314–315  
 symbol tables  
     about 12–13  
     nested 13–15  
 SYMBOLGEN option 41, 309, 407, 411, 423–424  
 symbolic variables  
     *See* macro variables  
 %SYMCHECK macro 237  
 %SYMCHKUP macro 203  
 %SYMDEL statement 204–206, 228–229  
 %SYMEXIST function 203, 229–230, 236–238  
 SYMGET function 96, 107–109, 111–116, 126, 241–243  
 SYMGETN function 107–109  
 %SYMGLOBL function 203, 221, 229–230, 236–238  
 %SYMLOCAL function 203, 221, 229–230, 236–238  
 SYMPUT routine 93–95, 428–429  
 SYMPUTX routine 11, 90–98, 146, 239, 306, 308, 321–322, 396, 402, 416–417, 419, 428–429  
 %SYSCALL function 169–170, 297  
 &SYSCC macro variable 31–32, 218  
 &SYSDATE macro variable 29–30, 171, 378–379  
 &SYSDAY macro variable 29–30  
 &SYSDSN macro variable 30–31  
 &SYSERR macro variable 138, 216–217  
 &SYSERRORTEXT macro variable 216–217  
 %SYSEVALF function 147, 148, 166–169, 171, 186, 188, 245, 390  
 %SYSEXEC statement 84, 198, 211–212, 300, 302, 375, 382–384  
 &SYSFILRC macro variable 224–225  
 %SYSFUNC function 37, 170–176, 178–184, 187, 192, 297–301, 365, 378, 382–384  
 %SYSGET function 196–199, 300, 302, 375, 379  
 &SYSINFO macro variable 219–220  
 &SYSLAST macro variable 30–31  
 &SYSLIBRC macro variable 224–225  
 %SYLPUT statement 246–247  
 %SYSMACDELETE statement 212–213, 260, 267  
 %SYSMACEEXEC function 200–201  
 %SYSMACEEXIST function 200–201  
 &SYSMACRONAME macro variable 33, 224  
 %SYSMCHECK macro 170  
 %SYSMEXCDEPTH function 199–200  
 %SYSMEXECNAME function 199–200  
 SYSMSG function 218–219  
 &SYSNOBS macro variable 223–224  
 &SYSPARM macro variable 214–216  
 %SYSPROD function 201–202  
 &SYSRC macro variable 32  
 %SYSRPUT statement 246–247  
 &SYSSCP macro variable 32  
 &SYSSCPL macro variable 32  
 &SYSSERR macro variable 31–32  
 &SYSSITE macro variable 32  
 SYS\_SQL\_IP\_STMT 105  
 system environmental variables, accessing 196–199  
 system initialization, controlling 429–432  
 system options  
     about 225  
     maintaining 346–347  
     memory control 225–226  
     using 40–44, 411  
         using with Macro Facility 40–44  
 system termination, controlling 429–432  
 &SYSTIME macro variable 29–30, 378–379  
 &SYSUSERID macro variable 32  
 &SYSWARNINGTEXT macro variable 216–217

## T

- TABLES table 296  
 Talbott, Helen-Jean 118  
 Tangedal, Mike 409  
 task structures 331  
 Tassoni, Charles John 77, 98, 292  
 %TDATAPREP macro 72  
 TEMPLATE procedure 19  
 termination, executing statements 431–432  
 terminology 7–8  
 %TEST macro 51, 127–128, 230  
 text  
     constant 36  
     defined 7  
     moving from macro variables 106–116  
     repeating 189–190  
 text functions  
     about 132, 150–151  
     %INDEX 151, 152, 186–187, 271  
     %LEFT 159–160  
     %LENGTH 151, 153–154, 157, 189, 215, 244, 273, 323–324  
     %LOWCASE 160–161  
     %QLEFT 159–160  
     %QLOWCASE 160–161  
     %QSCAN 151, 154–157, 176, 291, 310–311, 312, 361, 364, 404  
     %QSUBSTR 135, 151, 153, 157–158, 190, 198, 271, 273

- %QTRIM 161–162
- %QUPCASE 135, 151, 158–159
- %SCAN 151, 154–157, 186, 188–189, 291, 310–311, 361, 362, 367, 404–405, 415–416, 423–424
- %SUBSTR 151, 157–158, 176, 271, 272, 273
- %TRIM 161–162
- %UPCASE 158–159
- text strings, comparing 190
- %THEN statements
  - See* %IF-%THEN-%ELSE statement
- Theuwissen, Henri 122
- Thompson, Paul A. 328
- Thornton, S. Patrick 275, 351, 402
- TIME() function 378
- timing 125–128
- %TIMING macro 124–125
- Tindall, Bruce M. 44
- TITLE statement 8, 19, 139, 171–172, 183, 343, 388, 398, 429
- titles
  - coordinating 342–344
  - specifying dates in 171–172
- TITLES table 296
- TODAY() function 378
- tokens 9, 283
- Tomb, Michael E. 122, 297
- %TOPCNT macro 389–391
- TRANSPOSE procedure 353–356
- TRANWRD function 245, 347, 365–366, 394
- TRIM function 151, 161–162, 176, 183, 208, 414
- %TRIM macro 269, 273, 411
- Troxell, John K. 44, 296, 342, 409
- %TRYIT macro 190, 418, 421–422
- TSLIT function 249, 340
- %TSLIT macro 386
- &TTL macro variable 139–140
- Tyndall, Russ 135, 427
- Tze, Sylvia 273
- U**
  - UNIVARIATE procedure 345–346
  - %UNQUOTE function 134, 145–146, 151, 162, 205, 312, 340, 363, 395, 396
  - %UPCASE function 37, 151, 158–159, 176, 190, 297
  - %USEDSSN macro 322
  - \_user\_option 23
  - user-written macros 182–193
- V**
  - VALUE statement 347–350
  - values
    - assigning to macro variables 117–118
    - building lists of 99–102
    - returning 181
- Vandenbroucke, David A. 77
- VAR statement 133, 366–367
- VARDIR data set 319, 320
- %VAREXIST macro 366–367
- &&VAR&I macro variable 28, 283–285, 307–309
- &&&VAR&I macro variable 402–404
- variables
  - See also* macro variables
  - checking for existence of 366–367
  - creating list of names of 353–360
  - creating lists of using SQL step 356
  - matching names to patterns using Perl Regular Expressions 358–360
  - pulling names using macro functions 356–358
  - working with 351–371
- &&VAR&J macro variable 398
- %VARLIST macro 103
- VARNAME function 176, 301, 356–358
- VARNUM DATA step 366–367
- VARNUM function 301
- VARTYPE function 176
- VCATALG view 294
- VCOLUMN view 294
- %VERIFY macro 151, 176, 263, 269, 270
- vertical bars (|) 271
- vertical lists 283–286
- vertical macro arrays 307–309
- VEXTFL view 294
- Viergever, William 122
- VIEWS table 296
- Vijayarangan, Amarnath 32, 238
- VINDEX view 294
- Virgile, Bob 65
- VMACRO view 294
- VMEMBER view 294
- VNAME routine 240, 360, 402
- VOPTION view 294
- VSCATLG view 294
- VSLIB view 294
- VSTABLE view 294
- VSTYLE view 294
- VSVIEW view 294
- VTABLE view 294
- VTITLE view 294
- VVIEW view 294
- W**
  - %WAKEUP macro 185–186
  - Walgamotte, Veronica 42, 412
  - Wang, Deli 261
  - Wang, Diane 122
  - Wang, Xiaohui 84, 176, 208, 210, 342, 430
  - Ward, David L. 427
  - Watts, Perry 187, 206
  - Multiple-Plot Displays: Simplified with Macros*  
373
  - Werner, Nina L. 21

Westerlund, Earl R. 118  
WHERE clause 99, 101, 248, 312, 337, 338, 394–396  
Whitaker, Ken 132, 352  
White, Michael L. 102  
Whitlock, H. Ian 35, 45, 115, 116, 121–122, 132, 135, 145, 152, 157, 251, 297, 315, 409  
Widawski, Mel 17, 102, 341, 378, 398  
Williams, Christianna S. 35  
Wilson, Nancy K. 125  
%WINDOW statement 146, 208–211  
Wiser, Kristi 317  
Wobus, Diana Zhang 304  
Wong, Steve 216  
Wong, Tor-Lai 117  
word scanner 8  
%WORDCOUNT macro 157, 312–313, 361  
words  
    creating lists of 362–364  
    finding last word in lists of 361  
    placing commas between 364–365  
    quoting in an SQL step 366  
    quoting in lists 365–366  
    removing from lists 367–368  
    searching for 186–187  
    using word count to retrieve last 361–362  
%WRDCNT 424  
Wright, Wendi 118  
Wu, William 360

## **X**

X statement 338–339, 341  
Xia, Huanhong 122  
Xie, Fagen 248

## **Y**

Yao, Arthur K. 322  
Yarbrough, Kimberly D. 398  
Yeh, Shi-Tao 122, 176, 336, 342  
Yindra, Chris 28, 69, 176, 291, 302, 398  
Yu, Hsiwei (Michael) 176, 296, 378

## **Z**

Zdeb, Mike S. 98  
Zender, Cynthia 4  
Zhang, Julia 117  
Zhou, Jay 42, 317