CHAPTER

5

Scope of Macro Variables

Introduction

Every macro variable has a *scope*.* A macro variable's scope determines how it is assigned values and how the macro processor resolves references to it.

Two types of scope exist for macro variables: *global* and *local*. Global macro variables exist for the duration of the SAS session and can be referenced anywhere in the program–either inside or outside a macro. Local macro variables exist only during the execution of the macro in which the variables are created and have no meaning outside the defining macro.

Scopes can be nested, like boxes within boxes. For example, suppose you have a macro A that creates the macro variable LOC1 and a macro B that creates the macro variable LOC2. If the macro B is nested (executed) within the macro A, LOC1 is local to both A and B. However, LOC2 is local only to B.

Macro variables are stored in *symbol tables*, which list the macro variable name and its value. There is a global symbol table, which stores all global macro variables. Local macro variables are stored in a local symbol table that is created at the beginning of the execution of a macro.

* Earlier macro facility documentation often used the term "referencing environment" instead of scope.

Global Macro Variables

Figure 5.1 on page 38 illustrates the global symbol table during execution of the following program:

```
%let county=Clark;

%macro concat;
   data _null_;
      length longname $20;
      longname="&county"||" County";
      put longname;
   run;
%mend concat;

%concat
```

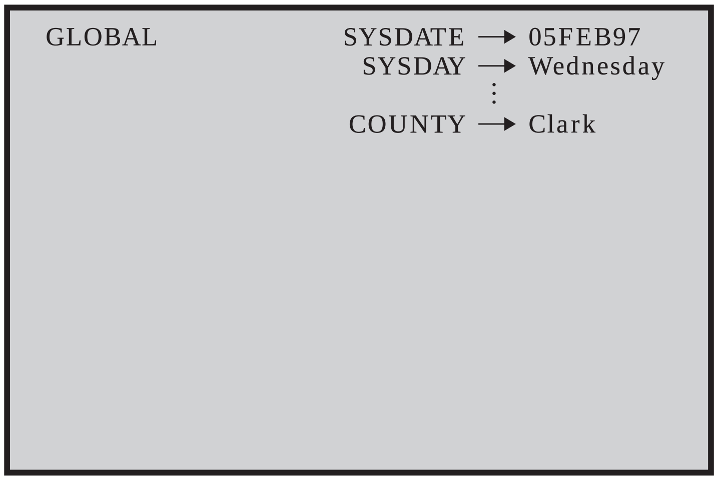
Calling the macro CONCAT produces the following statements:

```
data _null_;
      length longname $20;
      longname="Clark"||" County";
      put longname;
run;
```

The PUT statement writes the following to the SAS log:

```
Clark County
```

Figure 5.1 Global Macro Variables



Global macro variables include the following:

- □ all automatic macro variables except SYSPBUFF. See Chapter 13, "Macro Language Dictionary," for more information on SYSPBUFF and other automatic macro variables.
- □ macro variables created outside of any macro.

- macro variables created in %GLOBAL statements. See "Creating Global Macro Variables" later in this chapter for more information on the %GLOBAL statement.
- most macro variables created by the CALL SYMPUT routine. See “Special Cases of Scope with the CALL SYMPUT Routine” on page 54 for more information on the CALL SYMPUT routine.

You can create global macro variables any time during a SAS session or job. Except for some automatic macro variables, you can change the values of global macro variables any time during a SAS session or job.

In most cases, once you define a global macro variable, its value is available to you anywhere in the SAS session or job and can be changed anywhere. So, a macro variable referenced inside a macro definition is global if a global macro variable already exists by the same name (assuming the variable is not explicitly defined as local with the %LOCAL statement or in a parameter list). The new macro variable definition simply updates the existing global one. Exceptions that prevent you from referencing the value of a global macro variable are

- when a macro variable exists both in the global symbol table and in the local symbol table, you cannot reference the global value from within the macro that contains the local macro variable. In this case, the macro processor finds the local value first and uses it instead of the global value.
- if you create a macro variable in the DATA step with the SYMPUT routine, you cannot reference the value with an ampersand until the program reaches a step boundary. See Chapter 4, "Macro Processing," for more information on macro processing and step boundaries.

Local Macro Variables

Local macro variables are defined within an individual macro. Each macro you invoke creates its own local symbol table. Local macro variables exist only as long as a particular macro executes; when the macro stops executing, all local macro variables for that macro cease to exist.

Figure 5.2 on page 40 illustrates the local symbol table during the execution of the macro HOLINFO.

```
%macro holinfo(day,date);
   %let holiday=Christmas;
   %put *** Inside macro: ***;
   %put *** &holiday occurs on &day, &date, 1997. ***;
%mend holinfo;

%holinfo(Thursday,12/25)

%put *** Outside macro: ***;
%put *** &holiday occurs on &day, &date, 1997. ***;
```

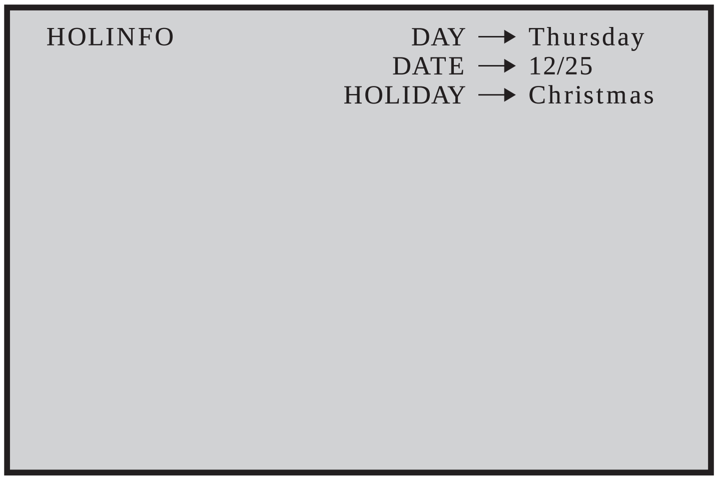
The %PUT statements write the following to the SAS log:

```
*** Inside macro: ***
*** Christmas occurs on Thursday, 12/25, 1997. ***

*** Outside macro: ***
WARNING: Apparent symbolic reference HOLIDAY not resolved.
WARNING: Apparent symbolic reference DAY not resolved.
WARNING: Apparent symbolic reference DATE not resolved.
```

```
*** &holiday occurs on &day, &date, 1997. ***
```

As you can see from the log, the local macro variables DAY, DATE, and HOLIDAY resolve inside the macro, but outside the macro they do not exist and therefore do not resolve.

Figure 5.2 Local Macro Variables



A macro's local symbol table is empty until the macro creates at least one macro variable. A local symbol table can be created by any of the following:

- the presence of one or more macro parameters
- a %LOCAL statement
- macro statements that define macro variables, such as %LET and the iterative %DO statement (assuming the variable does not already exist globally or a %GLOBAL statement is not used).

Note: Macro parameters are always local to the macro that defines them. You cannot make macro parameters global. (Although, you can assign the value of the parameter to a global variable; see "Creating Global Variables Based on the Value of Local Variables" later in this chapter.) △

When you invoke one macro inside another, you create nested scopes. Because you can have any number of levels of nested macros, your programs can contain any number of levels of nested scopes.

Writing the Contents of Symbol Tables to the SAS Log

While developing your macros, you may find it useful to write all or part of the contents of the global and local symbol tables to the SAS log. To do so, use the %PUT statement with one of the following options:

ALL	describes all currently defined macro variables, regardless of scope. This includes user-created global and local variables as well as automatic macro variables. Scopes are listed in the order of innermost to outermost.

AUTOMATIC	describes all automatic macro variables. The scope is listed as AUTOMATIC. All automatic macro variables are global except SYSPBUFF. See Chapter 12, "Macro Language Elements," and Chapter 13 for more information on specific automatic macro variables.
GLOBAL	describes all user-created global macro variables. The scope is listed as GLOBAL. Automatic macro variables are not listed.
LOCAL	describes user-created local macro variables defined within the currently executing macro. The scope is listed as the name of the macro in which the macro variable is defined.
USER	describes all user-created macro variables, regardless of scope. The scope is either GLOBAL, for global macro variables, or the name of the macro in which the macro variable is defined.

For example, consider the following program:

```
%let origin=North America;

%macro dogs(type=);
   data _null_;
      set all_dogs;
      where dogtype="&type" and dogorig="&origin";
      put breed " is for &type.";
   run;

   %put _user_;
%mend dogs;

%dogs(type=work)
```

The %PUT statement preceding the %MEND statement writes to the SAS log the scopes, names, and values of all user-generated macro variables:

```
DOGS    TYPE    work
GLOBAL    ORIGIN    North America
```

Because TYPE is a macro parameter, TYPE is local to the macro DOGS, with value **work**. Because ORIGIN is defined in open code, it is global.

How Macro Variables Are Assigned and Resolved

Before the macro processor creates a variable, assigns a value to a variable, or resolves a variable, the macro processor searches the symbol tables to determine whether the variable already exists. The search begins with the most local scope and, if necessary, moves outward to the global scope. The request to assign or resolve a variable comes from a macro variable reference in open code (outside a macro) or within a macro.

Figure 5.3 illustrates the search order the macro processor uses when it receives a macro variable reference that requests a variable be created or assigned. Figure 5.4 illustrates the process for resolving macro variable references. Both these figures represent the most basic type of search and do not apply in special cases, such as when a %LOCAL statement is used or the variable is created by CALL SYMPUT.

Figure 5.3 Search Order When Assigning or Creating Macro Variables

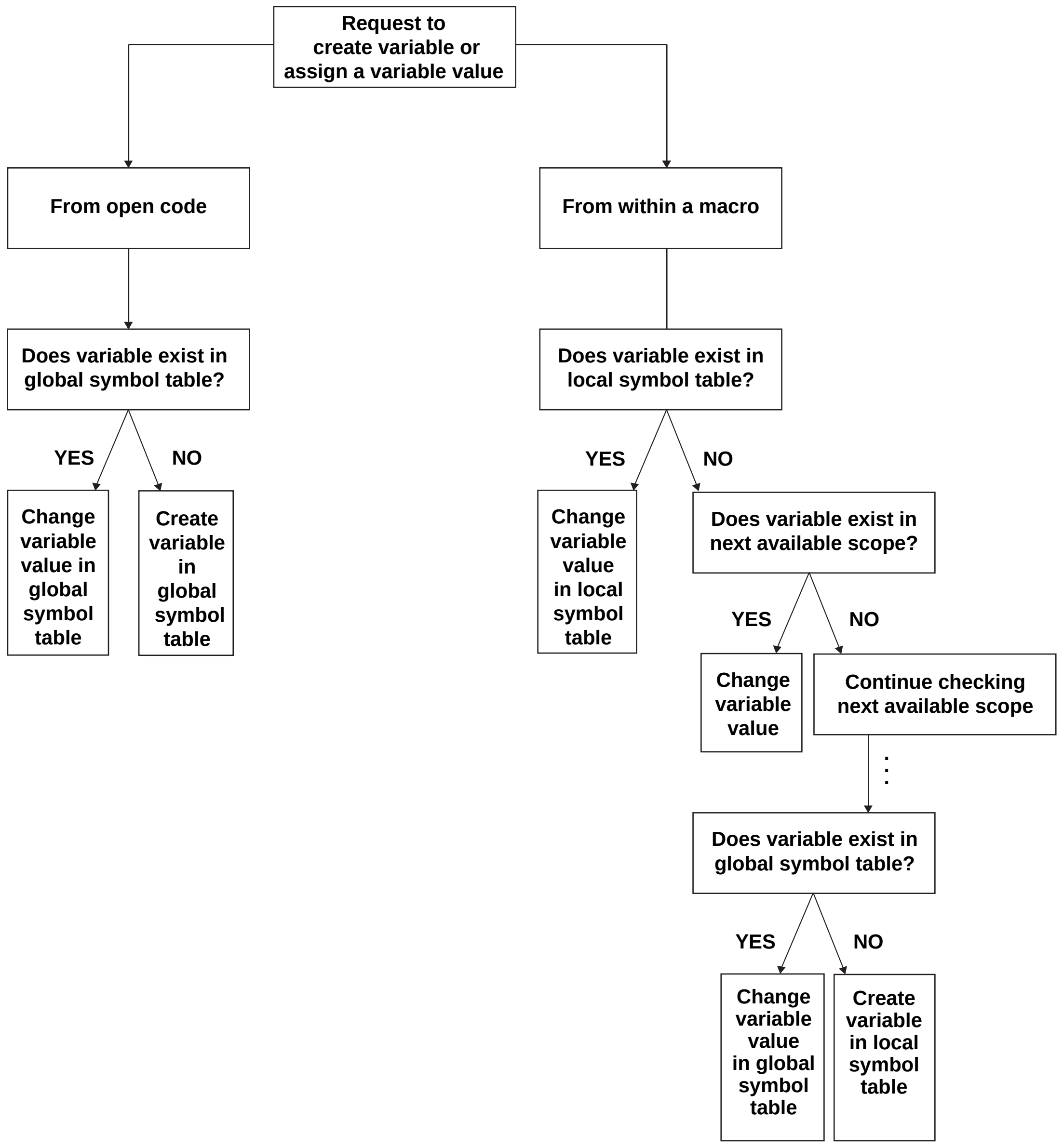
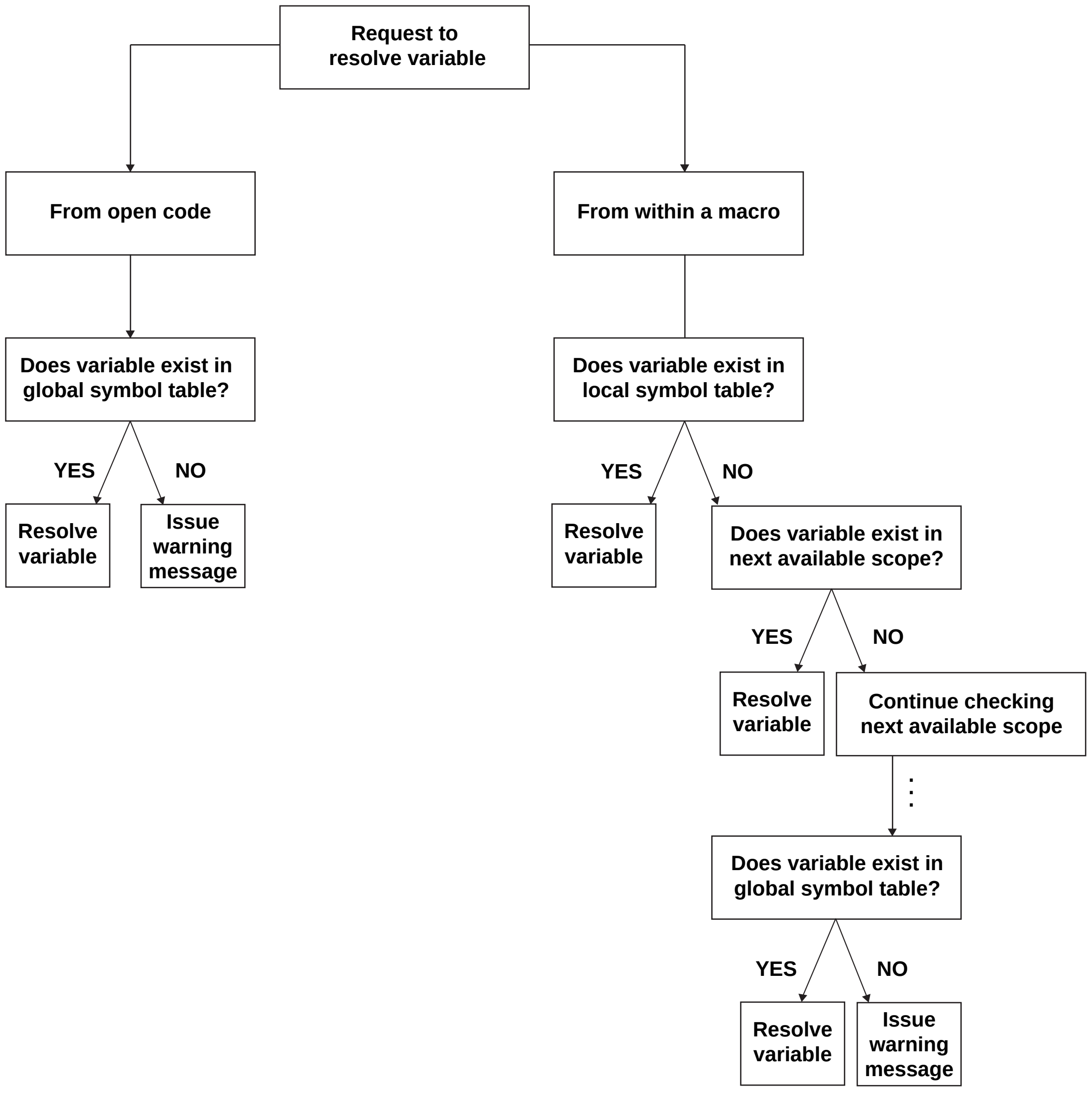

Request to create variable or assign a variable value
From open code
From within a macro
Does variable exist in global symbol table?
YES
NO
Change variable value in global symbol table
Create variable in global symbol table
Does variable exist in local symbol table?
YES
NO
Change variable value in local symbol table
Does variable exist in next available scope?
YES
NO
Change variable value
Continue checking next available scope
Does variable exist in global symbol table?
YES
NO
Change variable value in global symbol table
Create variable in local symbol table

Figure 5.4 Search Order When Resolving Macro Variable References


Request to resolve variable
From open code
From within a macro
Does variable exist in global symbol table?
YES
NO
Resolve variable
Issue warning message
Does variable exist in local symbol table?
YES
NO
Resolve variable
Does variable exist in next available scope?
YES
NO
Resolve variable
Continue checking next available scope
Does variable exist in global symbol table?
YES
NO
Resolve variable
Issue warning message

Examples of Macro Variable Scopes

Changing the Values of Existing Macro Variables

When the macro processor executes a macro program statement that can create a macro variable (such as a %LET statement), the macro processor attempts to change the value of an existing macro variable rather than create a new macro variable. The %GLOBAL and %LOCAL statements are exceptions.

To illustrate, consider the following %LET statements. Both statements assign values to the macro variable NEW:

```
%let new=inventry;
%macro name1;
   %let new=report;
%mend name1;
```

Suppose you submit the following statements:

```
%name1

data &new;
```

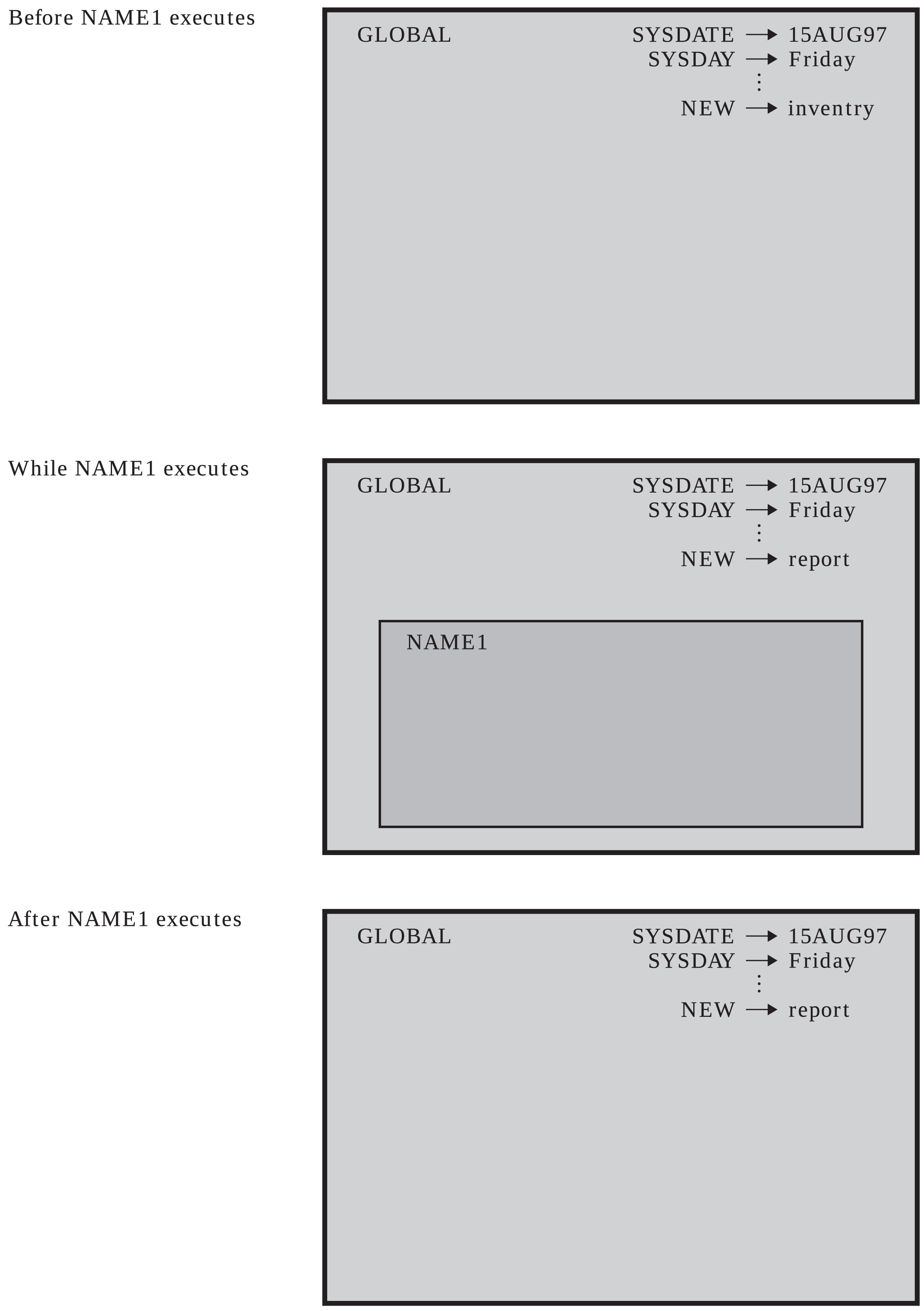
These statements produce the following statement:

```
data report;
```

Because NEW exists as a global variable, the macro processor changes the value of that variable rather than creating a new one. The macro NAME1's local symbol table remains empty.

Figure 5.5 illustrates the contents of the global and local symbol tables before, during, and after NAME1's execution.

Figure 5.5 Snapshots of Symbol Tables



Creating Local Variables

When the macro processor executes a macro program statement that can create a macro variable, the macro processor creates the variable in the local symbol table if no macro variable with the same name is available to it. Consider the following example:

```
%let new=inventry;
%macro name2;
   %let new=report;
   %let old=warehse;
%mend name2;

%name2

data &new;
   set &old;
run;
```

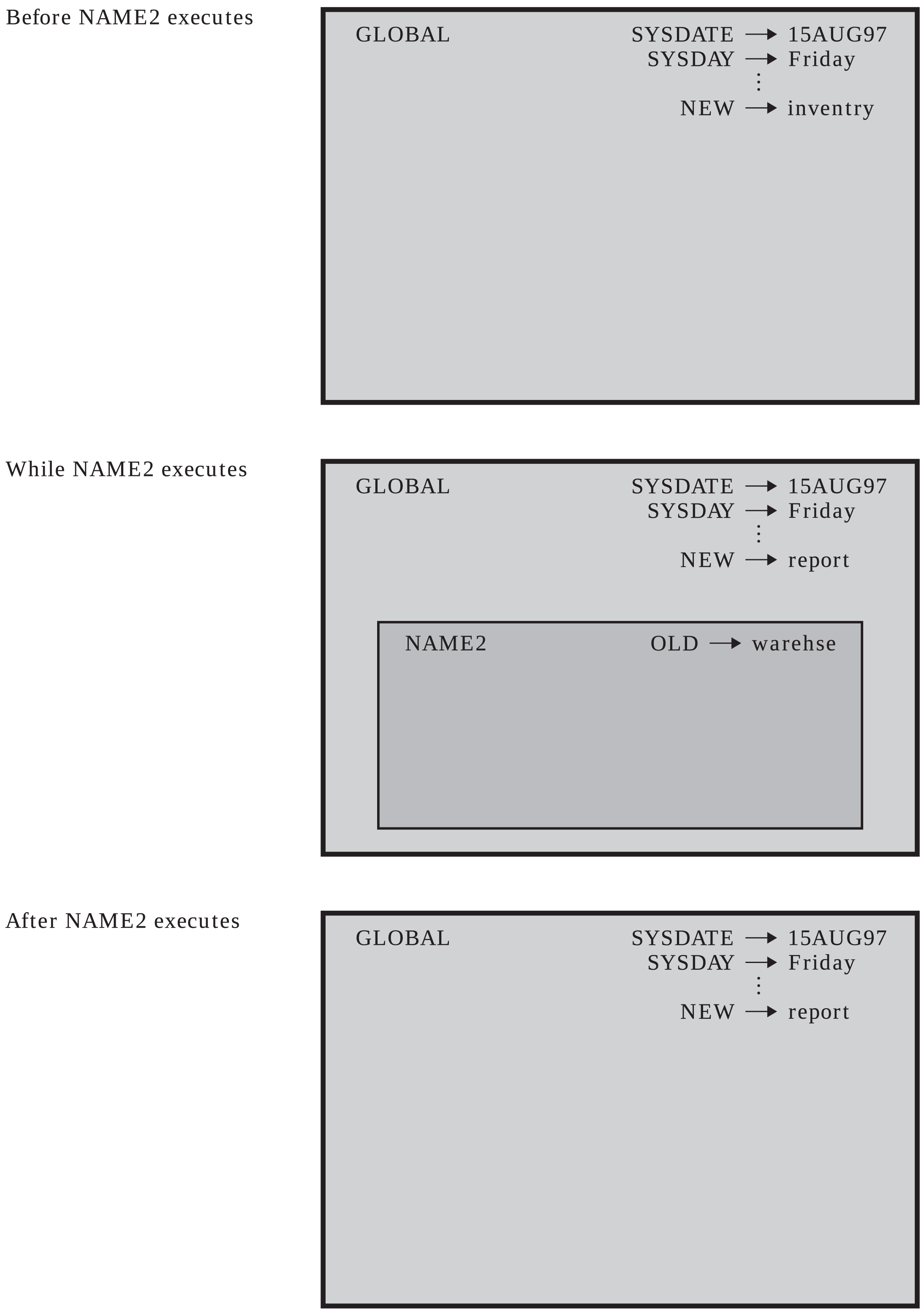
After NAME2 executes, the SAS compiler sees the following statements:

```
data report;
   set &old;
run;
```

The macro processor encounters the reference &OLD after macro NAME2 has finished executing; thus, the macro variable OLD no longer exists. The macro processor is not able to resolve the reference and issues a warning message.

Figure 5.6 on page 47 illustrates the contents of the global and local symbol tables at various stages.

Figure 5.6 Symbol Tables at Various Stages



But suppose you place the SAS statements inside the macro NAME2, as in the following program:

```
%let new=inventry;
%macro name2;
   %let new=report;
   %let old=warehse;
   data &new;
      set &old;
   run;
%mend name2;

%name2
```

In this case, the macro processor generates the SET statement during the execution of NAME2, and it locates OLD in NAME2's local symbol table. Therefore, executing the macro produces the following statements:

```
data report;
   set warehse;
run;
```

The same rule applies regardless of how many levels of nesting exist. Consider the following example:

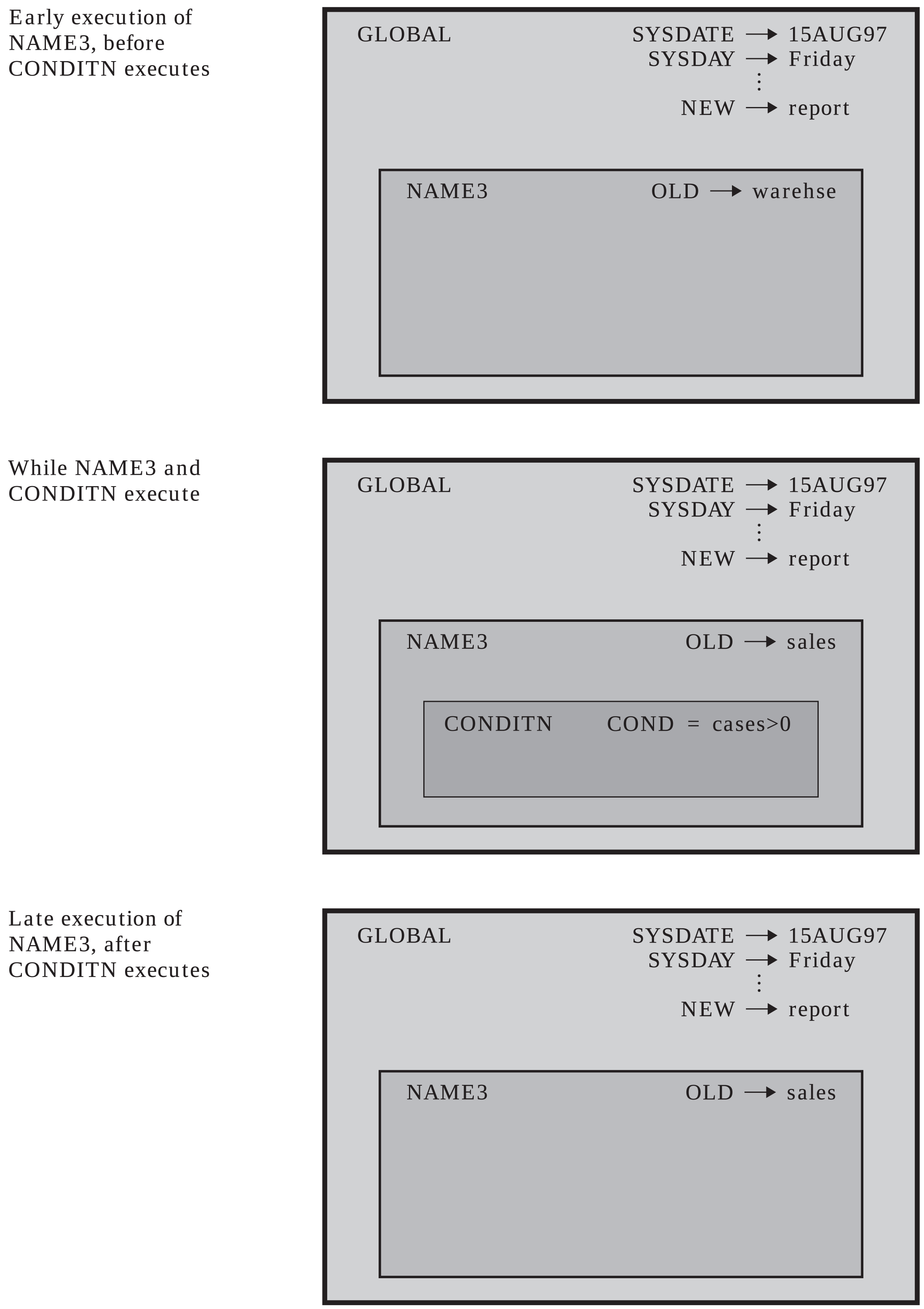
```
%let new=inventry;
%macro conditn;
   %let old=sales;
   %let cond=cases>0;
%mend conditn;

%macro name3;
   %let new=report;
   %let old=warehse;
   %conditn
      data &new;
         set &old;
         if &cond;
      run;
%mend name3;

%name3
```

The macro processor generates these statements:

```
data report;
   set sales;
   if &cond;
run;
```

CONDITN finishes executing before the macro processor reaches the reference &COND, so no variable named COND exists when the macro processor attempts to resolve the reference. Thus, the macro processor issues a warning message and generates the unresolved reference as part of the constant text and issues a warning message. Figure 5.7 on page 49 shows the symbol tables at each step.

Figure 5.7 Symbol Tables Showing Two Levels of Nesting



Notice that the placement of a macro invocation is what creates a nested scope, not the placement of the macro definition. For example, invoking CONDITN from within

NAME3 creates the nested scope; it is not necessary to define CONDITN within NAME3.

Forcing a Macro Variable to Be Local

At times you need to ensure that the macro processor creates a local macro variable rather than changing the value of an existing macro variable. In this case, use the %LOCAL statement to create the macro variable.

Explicitly make all macro variables created within macros local when you do not need their values after the macro stops executing. Debugging the large macro programs is easier if you minimize the possibility of inadvertently changing a macro variable's value. Also, local macro variables do not exist after their defining macro finishes executing, while global variables exist for the duration of the SAS session; therefore, local variables use less overall storage.

Suppose you want to use the macro NAMELST to create a list of names for a VAR statement, as shown here:

```
%macro namelst(name,number);
   %do n=1 %to &number;
      &name&n
   %end;
%mend namelst;
```

You invoke NAMELST in this program:

```
%let n=North State Industries;

proc print;
   var %namelst(dept,5);
   title "Quarterly Report for &n";
run;
```

After macro execution, the SAS compiler sees the following statements:

```
proc print;
   var dept1 dept2 dept3 dept4 dept5;
   title "Quarterly Report for 6";
run;
```

The macro processor changes the value of the global variable N each time it executes the iterative %DO loop. (After the loop stops executing, the value of N is 6, as described in " %DO" in Chapter 13, "Macro Language Dictionary.") To prevent conflicts, use a %LOCAL statement to create a local variable N, as shown here:

```
%macro namels2(name,number);
   %local n;
   %do n=1 %to &number;
      &name&n
   %end;
%mend namels2;
```

Now execute the same program:

```
%let n=North State Industries;

proc print;
   var %namels2(dept,5);
   title "Quarterly Report for &n";
```

```
run;
```

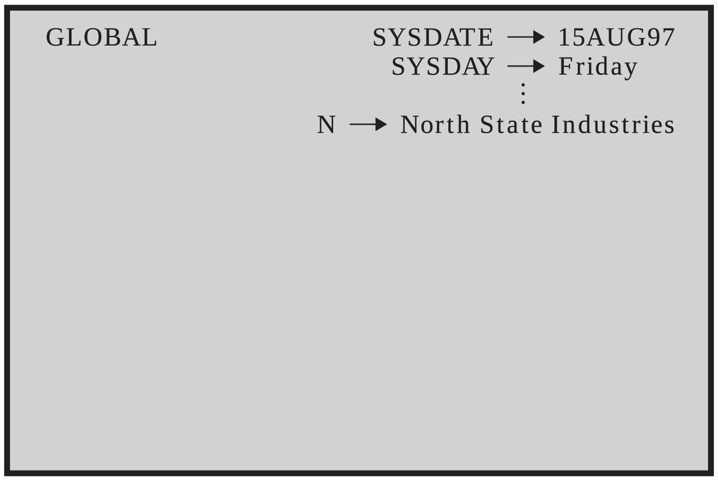
The macro processor generates the following statements:

```
proc print;
   var dept1 dept2 dept3 dept4 dept5;
   title "Quarterly Report for North State Industries";
run;
```

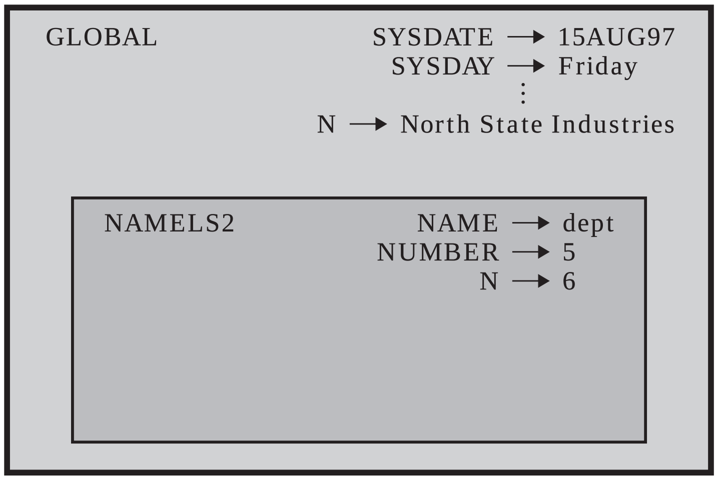
Figure 5.8 on page 52 shows the symbol tables before NAMELS2 executes, while NAMELS2 is executing, and when the macro processor encounters the reference &N in the TITLE statement.

Figure 5.8 Global and Local Variables with the Same Name

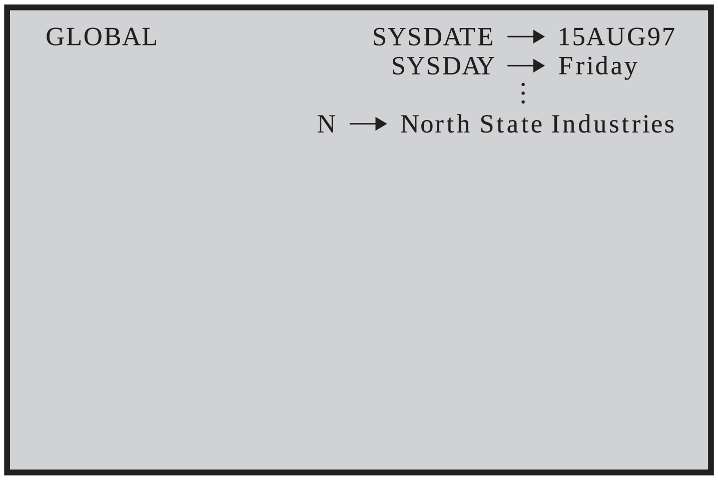
Before NAMELS2 executes



While NAMELS2 executes
(at end of last iteration
of %DO loop)



After NAMELS2 executes



Creating Global Macro Variables

The %GLOBAL statement creates a global macro variable if a variable with the same name does not already exist there, regardless of what scope is current.

For example, in the macro NAME4, the macro CONDITN contains a %GLOBAL statement that creates the macro variable COND as a global variable:

```
%macro conditn;
   %global cond;
   %let old=sales;
   %let cond=cases>0;
%mend conditn;
```

Here is the rest of the program:

```
%let new=inventry;

%macro name4;
   %let new=report;
   %let old=warehse;
   %conditn
   data &new;
      set &old;
      if &cond;
   run;
%mend name4;

%name4
```

Invoking NAME4 generates these statements:

```
data report;
   set sales;
   if cases>0;
run;
```

Suppose you want to put the SAS DATA step statements outside NAME4. In this case, all the macro variables must be global for the macro processor to resolve the references. You cannot add OLD to the %GLOBAL statement in CONDITN because the %LET statement in NAME4 has already created OLD as a local variable to NAME4 by the time CONDITN begins to execute. (You cannot use the %GLOBAL statement to make an existing local variable global.)

Thus, to make OLD global, use the %GLOBAL statement before the variable reference appears anywhere else, as shown here in the macro NAME5:

```
%let new=inventry;

%macro conditn;
   %global cond;
   %let old=sales;
   %let cond=cases>0;
%mend conditn;

   %macro name5;
      %global old;
      %let new=report;
      %let old=warehse;
      %conditn
   %mend name5;

%name5

data &new;
   set &old;
   if &cond;
run;
```

Now the %LET statement in NAME5 changes the value of the existing global variable OLD rather than creating OLD as a local variable. The SAS compiler sees the following statements:

```
data report;
   set sales;
   if cases>0;
run;
```

Creating Global Variables Based on the Value of Local Variables

To use a local variable such as a parameter outside a macro, use a %LET statement to assign the value to a global variable with a different name, as in this program:

```
%macro namels3(name,number);
   %local n;
   %global g_number;
   %let g_number=&number;
   %do n=1 %to &number;
      &name&n
   %end;
%mend namels3;
```

Now invoke the macro NAMELS3 in the following the program:

```
%let n=North State Industries;

proc print;
   var %namels3(dept,5);
   title "Quarterly Report for &n";
   footnote "Survey of &g_number Departments";
run;
```

The compiler sees the following statements:

```
proc print;
   var dept1 dept2 dept3 dept4 dept5;
   title "Quarterly Report for North State Industries";
   footnote "Survey of 5 Departments";
run;
```

Special Cases of Scope with the CALL SYMPUT Routine

Most problems with CALL SYMPUT involve the lack of an explicit step boundary between the CALL SYMPUT statement that creates the macro variable and the macro variable reference that uses that variable. (See Chapter 8, "Interfaces with the Macro

Facility," for details on CALL SYMPUT.) However, a few special cases exist that involve the scope of a macro variable created by CALL SYMPUT.

Two rules control where CALL SYMPUT creates its variables:

1 CALL SYMPUT creates the macro variable in the current symbol table available while the DATA step is executing, provided that symbol table is not empty. If it is empty (contains no local macro variables), usually CALL SYMPUT creates the variable in the closest nonempty symbol table.

2 However, if the macro variable SYSPBUFF is created at macro invocation time or the executing macro contains a computed %GOTO statement, CALL SYMPUT creates the variable in the local symbol table, even if that symbol table is empty. A computed %GOTO statement is one that uses a label that contains an & or a % in it. That is, a computed %GOTO statement contains a macro variable reference or a macro call that produces a text expression. Here is an example of a computed %GOTO statement:

```
%goto &home;
```

The symbol table that is currently available to a DATA step is the one that exists when the SAS System determines that the step is complete. (The SAS System considers a DATA step to be complete when it encounters a RUN statement, a semicolon after data lines, or the beginning of another step).

In simplest terms, if an executing macro contains a computed %GOTO statement, or if the macro variable SYSPBUFF is created at macro invocation time, but the local symbol table is empty, CALL SYMPUT behaves as though the local symbol table was not empty, and creates a local macro variable.

You may find it helpful to use the %PUT statement with the _USER_ option to determine what symbol table the CALL SYMPUT routine has created the variable in.

Example Using CALL SYMPUT with Complete DATA Step and a Nonempty Local Symbol Table

Consider the following example, which contains a complete DATA step with a CALL SYMPUT statement inside a macro:

```
%macro env1(param1);
   data _null_;
      x = 'a token';
      call symput('myvar1',x);
   run;
%mend env1;

%env1(10)

data temp;
   y = "&myvar1";
run;
```

When you submit these statements, you receive an error message:

```
WARNNG:   Apparent symbolic reference MYVAR1 not resolved.
```

This message appears because the DATA step is complete within the environment of ENV1 (that is, the RUN statement is within the macro) and because the local symbol table of ENV1 is not empty (it contains parameter PARAM1). Therefore, the CALL SYMPUT routine creates MYVAR1 as a local variable for ENV1, and the value is not available to the subsequent DATA step, which expects a global macro variable.

To see the scopes, add a %PUT statement with the _USER_ option to the macro, and a similar statement in open code. Now invoke the macro as before:

```
%macro env1(param1);
   data _null_;
      x = 'a token';
      call symput('myvar1',x);
   run;

   %put ** Inside the macro: **;
   %put _user_;
%mend env1;

%env1(10)

%put ** In open code: **;
%put _user_;

data temp;
   y = "&myvar1";  /* ERROR - MYVAR1 is not available in open code. */
run;
```

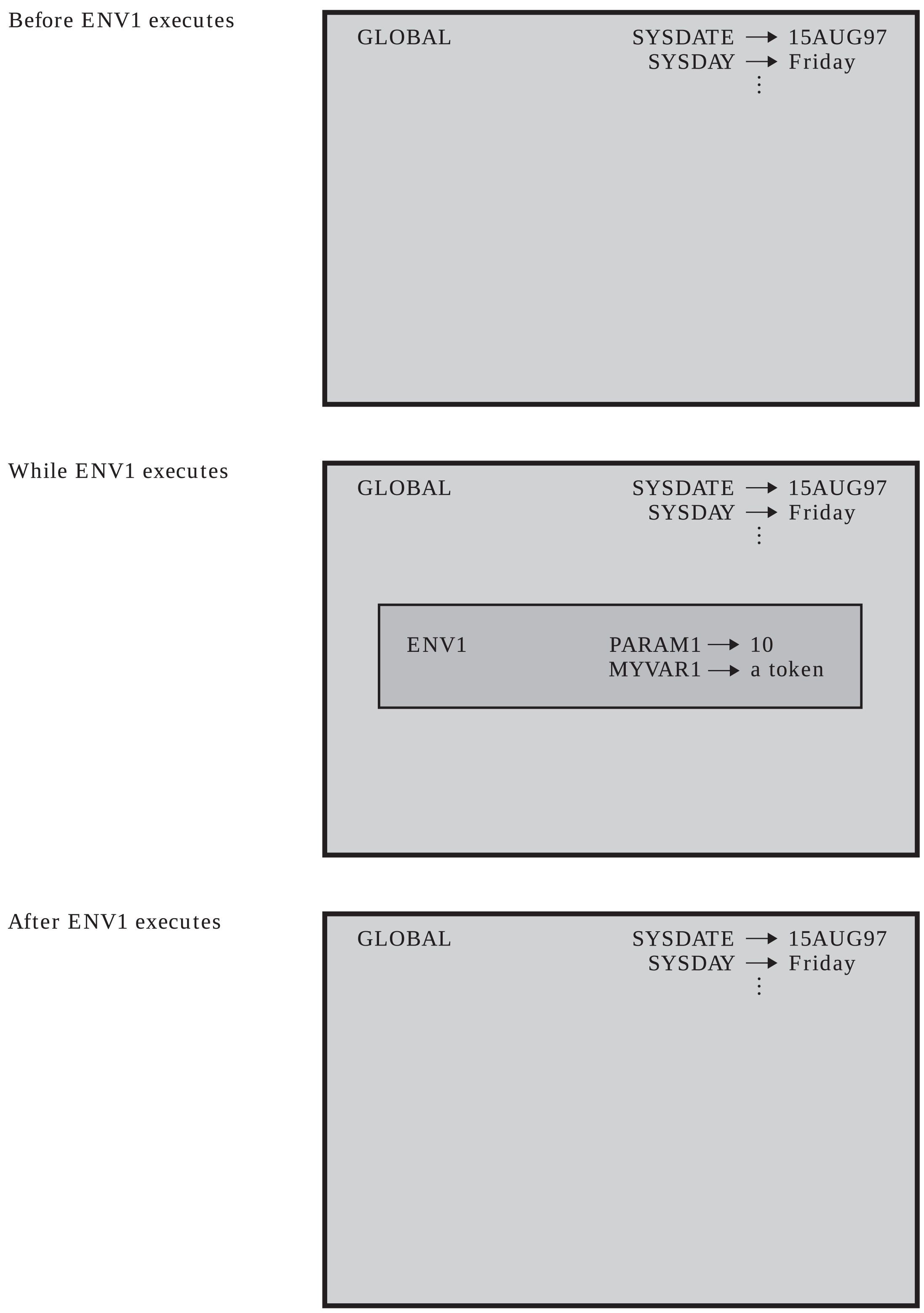
When the %PUT _USER_ statements execute, they write the following information to the SAS log:

```
** Inside the macro: **
ENV1    MYVAR1    a token
ENV1    PARAM1    10

** In open code: **
```

The MYVAR1 macro variable is created by CALL SYMPUT in the local ENV1 symbol table. The %PUT _USER_ statement in open code writes nothing to the SAS log, because no global macro variables are created.

Figure 5.9 on page 57 shows all of the symbol tables in this example.

Figure 5.9 The CALL SYMPUT Routine in a Macro Generating a Complete DATA Step



Example Using CALL SYMPUT with an Incomplete DATA Step

In the macro ENV2, shown here, the DATA step is not complete within the macro because there is no RUN statement:

```
%macro env2(param2);
   data _null_;
      x = 'a token';
      call symput('myvar2',x);
%mend env2;

%env2(20)
run;

data temp;
   y="&myvar2";
run;
```

These statements execute without errors. The DATA step is complete only when the SAS System encounters the RUN statement (in this case, in open code); thus, the current scope of the DATA step is the global scope. CALL SYMPUT creates MYVAR2 as a global macro variable, and the value is available to the subsequent DATA step.

Again, use the %PUT statement with the _USER_ option to illustrate the scopes:

```
%macro env2(param2);
   data _null_;
      x = 'a token';
      call symput('myvar2',x);

   %put ** Inside the macro: **;
   %put _user_;
%mend env2;

%env2(20)

run;

%put ** In open code: **;
%put _user_;

data temp;
   y="&myvar2";
run;
```

When the %PUT _USER_ statement within ENV2 executes, it writes the following to the SAS log:

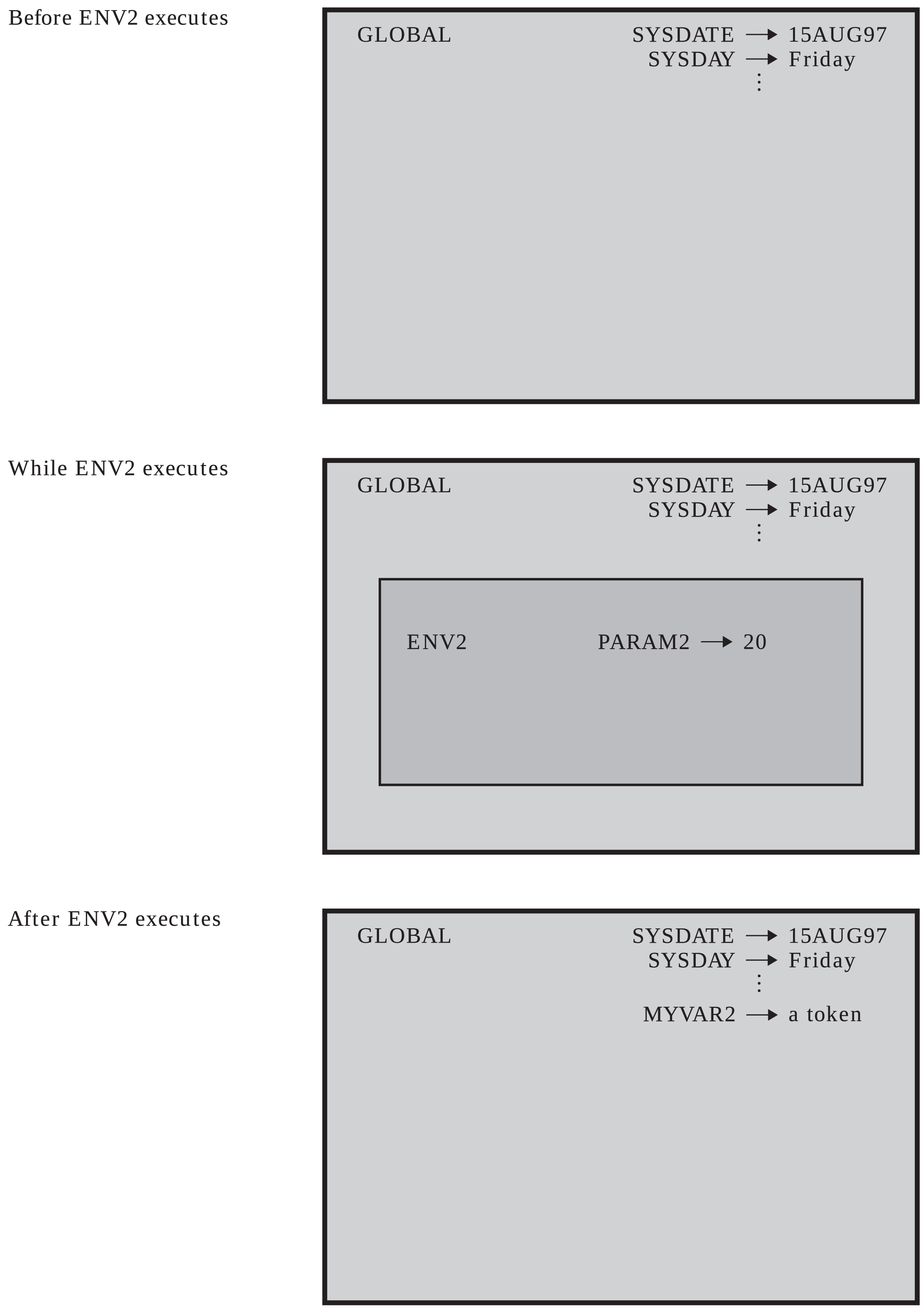
```
** Inside the macro: **
ENV2   PARAM2   20
```

The %PUT _USER_ statement in open code writes the following to the SAS log:

```
** In open code: **
GLOBAL   MYVAR2   a token
```

Figure 5.10 on page 59 shows all the scopes in this example.

Figure 5.10 The CALL SYMPUT Routine in a Macro Generating an Incomplete DATA Step



Example Using CALL SYMPUT with a Complete DATA Step and an Empty Local Symbol Table

In the following example, ENV3 does not use macro parameters. Therefore, its local symbol table is empty:

```
%macro env3;
   data _null_;
      x = 'a token';
      call symput('myvar3',x);
   run;

   %put ** Inside the macro: **;
   %put _user_;
%mend env3;

%env3

%put ** In open code: **;
%put _user_;

data temp;
   y="&myvar3";
run;
```

In this case, the DATA step is complete and executes within the macro, but the local symbol table is empty. So, CALL SYMPUT creates MYVAR3 in the closest available nonempty symbol table–the global symbol table. Both %PUT statements show that MYVAR3 exists in the global symbol table:

```
** Inside the macro: **
GLOBAL    MYVAR3   a token

** In open code: **
GLOBAL    MYVAR3   a token
```

Example Using CALL SYMPUT with SYSPBUFF and an Empty Local Symbol Table

In the following example, the presence of the SYSPBUFF automatic macro variable causes CALL SYMPUT to behave as though the local symbol table were not empty, even though the macro ENV4 has no parameters or local macro variables:

```
%macro env4 /parmbuff;
 data _null_;
      x = 'a token';
      call symput('myvar4',x);
   run;

   %put ** Inside the macro: **;
   %put _user_;
   %put &syspbuff;
%mend env4;

%env4

%put ** In open code: **;
```

```
%put _user_;
%put &syspbuff;

data temp;
   y="&myvar4";  /* ERROR - MYVAR4 is not available in open code */
run;
```

The presence of the /PARMBUFF specification causes the SYSPBUFF automatic macro variable to be created. So, when you call macro ENV4, CALL SYMPUT creates the macro variable MYVAR4 in the local symbol table (that is, in ENV4's), even though the macro ENV4 has no parameters and no local variables.

The results of the %PUT statements prove this–the score of MYVAR4 is listed as ENV4, and the reference to SYSPBUFF does not resolve in the open code %PUT statement because SYSPBUFF is local to ENV4:

```
** Inside the macro: **
ENV4     MYVAR4    a token

** In open code: **
WARNING: Apparent symbolic reference SYSPBUFF not resolved.
```

For more information about SYSPBUFF, see Chapter 13.

The correct bibliographic citation for this manual is as follows: SAS Institute Inc., *SAS Macro Language: Reference, Version 8*, Cary, NC: SAS Institute Inc., 1999. 310 pages.

SAS Macro Language: Reference, Version 8

1-58025-522-1

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

1st printing, October 1999