CHAPTER 3
SCL Fundamentals

- *Introduction* **18**
- *SCL Data Types* **18**
  - *Declaring Data Types* **18**
  - *Numeric (NUM) Variables* **18**
  - *Character (CHAR) Variables* **19**
  - *Lists* **19**
  - *Objects* **19**
    - *Specific Objects (CLASS and INTERFACE)* **20**
    - *Generic OBJECTs* **20**
    - *Specifying the Object Type at Run Time* **20**
- *Names in SCL* **21**
- *SCL Keywords* **22**
- *Variables* **22**
  - *Window Variables* **22**
    - *Name* **22**
    - *Data Type* **23**
    - *Length* **23**
  - *Nonwindow Variables* **23**
    - *Name* **23**
    - *Data Type* **23**
    - *Length* **23**
    - *Scope* **24**
  - *System Variables* **24**
- *Constants* **26**
  - *Numeric Constants* **26**
  - *Character Constants* **27**
  - *Numeric-to-Character Conversion* **27**
- *Operators* **28**
  - *Arithmetic Operators* **28**
  - *Comparison Operators* **28**
    - *Colon Modifier* **29**
    - *IN Operator* **29**
  - *Logical (Boolean) Operators* **30**
    - *AND Operator* **30**
    - *OR Operator* **30**
    - *NOT Operator* **31**
- *Expressions* **31**
  - *Boolean Numeric Expressions* **31**
  - *Using Functions in Expressions* **32**
- *SCL Statements* **32**
  - *Executable and Declarative Statements* **32**

The Assignment Statement **33**
Program Comments **33**
SCL Functions **33**
SCL CALL Routines **34**
Passing Arguments to SCL Functions and CALL Routines **34**
Input, Output, and Update Parameters **34**
Combining Language Elements into Program Statements **36**

Introduction

Like any language, SAS Component Language has its own vocabulary and syntax. An SCL program consists of one or more SCL statements, which can include keywords, expressions, constants, and operators.

SCL Data Types

SCL has the following data types:

- NUM
- CHAR
- LIST
- generic OBJECT
- specific object (CLASS or INTERFACE).

Declaring Data Types

You can use the DECLARE statement for declaring any type of SCL variable. You can use the LENGTH statement for declaring numeric and character variables.

You can also declare data types when you use the ENTRY and METHOD statements. In these statements, you must specify a colon before a named data type; with an unnamed data type, (for example, $), the colon is optional. For example:

```
ENTRY: name :$20
       location $20
       zipcode :num
       mybutton :mylib.mycat.button.class
       return=char;
```

For details, see “DECLARE” on page 330, “LENGTH” on page 510, “ENTRY” on page 369, and “METHOD” on page 540.

Numeric (NUM) Variables

Numeric variables contain numbers and are stored as doubles. They are declared with the keyword NUM.

```
   /* declare a numeric variable AGE           */
declare num age;

   /* declare the numeric variables AGE and YEARS*/
```

```
declare num age, years;

   /* declare numeric variables X and Y.         */
   /* Initialize X to 1 and Y to 20 plus the     */
   /* value of X.                                */
declare num x, y=20+x;
```

Character (CHAR) Variables

Character variables can contain up to 32,767 characters and are declared with the keyword CHAR. A variable that is declared as CHAR without a specified length is assigned a default length of 200.

```
   /* declare a character variable NAME and   */
   /* assign the value ABC to it              */
declare char name='abc';

   /* declare a character variable NAME       */
   /* with a length of 20                     */
declare char(20) name;
```

Lists

SCL lists are ordered collections of data. Lists are dynamic; they grow and shrink to accommodate the number or size of the items that you store in them. Lists can contain items of different data types.

To declare an SCL list, use the keyword LIST. The following example declares the list MYLIST:

```
declare list mylist;
```

The function that creates the list (for example, MAKELIST) assigns the identifier for the list to the variable, as shown below.

```
declare list mylist;
   ...more SCL statements...
mylist=makelist();
```

Note: To maintain compatibility with previous releases, the SCL compiler does not generate error messages if a list is not declared or if it is declared as a numeric variable. However, it is recommended that you declare lists so that the compiler can identify errors. A list must be declared as type List when it is passed to a method that requires an argument of type List. See “Overloading and List, Object, and Numeric Types” on page 111. △

For information about using lists, see Chapter 5, “SCL Lists,” on page 47.

Objects

Objects can be declared in either of two ways:

- as a specific object of type Class or Interface. When an object is declared with the name of the class, the compiler can validate attributes and methods for the object and can return error messages if incorrect attributes or methods are used.

- as a generic object of type Object. The specific object class that is associated with the generic object cannot be resolved until run time. The compiler reserves space in the SAS Data Vector (SDV) for the object, but it cannot validate attributes or methods for the object, because it does not know the names of classes. Instead, this validation is deferred until program execution. Consequently, you should use the OBJECT keyword to declare an object only when necessary, so that you can obtain optimal run-time performance.

You can use dot notation for accessing attributes and methods for both specific objects and generic objects.

Note: If you want to use dot notation to access the attributes or methods of a Version 6 widget, then you need to declare its widget ID of OBJECT type, and you must obtain its widget ID with the _getWidget method. For example, Text is a Version 6 text entry widget. To access its methods or attributes with dot notation, you should use code that looks like this:

```
dcl object obj;
/* dcl sashelp.fsp.efield.class obj; */
call notify ( 'Text', '_getWidget', obj );
obj.backgroundColor = 'blue';
```

See "Accessing Object Attributes and Methods With Dot Notation" on page 119 for more information. △

Specific Objects (CLASS and INTERFACE)

The following example declares an object named DataHolder as an instance of the Collection class, which is provided with SAS software:

```
declare sashelp.fsp.collection.class DataHolder;
```

When you declare a class, you can also use the IMPORT statement to reference the class and then use an abbreviated form of the class name in the DECLARE statement. For example:

```
import sashelp.fsp.collection.class;
declare collection DataHolder;
```

Generic OBJECTs

In the following example, MyObject is recognized by the compiler as an object, but the compiler has no information about the type of class instance that the object will actually be:

```
declare object MyObject;
```

Specifying the Object Type at Run Time

The following example declares an object named PgmObj2 and then specifies one condition under which PgmObj2 will be a collection object and another condition under which PgmObj2 will be an object that is created from a class named Foo. The _NEW_ operator creates the object.

```
declare object PgmObj2,
        num    x;
if x=1 then
   PgmObj2=_new_ sashelp.fsp.collection.class;
else
```

```
   PgmObj2=_new_ sashelp.fsp.foo.class;
```

As described above, you can use the IMPORT statement to reference a class definition and then use an abbreviated class name when you create the class.

```
import sashelp.fsp.collection.class;
import sashelp.fsp.foo.class;
declare object PgmObj2,
        num    x;
if x=1 then
   PgmObj2=_new_ collection();
   end;
else
   PgmObj2=_new_ foo();
```

Any errors that result from using incorrect methods or attributes for PgmObj2 and Foo will cause the program to halt.

Names in SCL

In SCL, the rules for names are

1. Librefs and filerefs can have a maximum length of 8 characters. Other names — including names of SCL variables, arrays, SCL lists, SAS tables, views, indexes, catalogs, catalog entries, macros, and macro variables — can be 32 characters long.
2. The first character must be a letter (A, B, C, . . . , Z) or an underscore (_). Subsequent characters can be letters, numeric digits (0, 1, . . . , 9), or underscores.
3. Names are stored in the case in which they are entered, which can be lower case, mixed case, or upper case.
4. Names cannot contain blanks.
5. SCL honors the names that are reserved by SAS software for automatic variables, lists of variables, SAS tables, and librefs. Thus, you cannot use these names in your SCL programs.

   a. When creating variables, do not use the names of special SAS automatic variables (for example, _N_ and _ERROR_) nor the names of lists of variables (for example, _CHARACTER_, _NUMERIC_, and _ALL_).
   b. Do not use any of the following names as a libref:

      - SASCAT
      - SASHELP
      - SASMSG
      - SASUSER
      - USER
      - WORK

      Use LIBRARY only as the libref to point to a SAS data library containing a FORMATS catalog that was created with PROC FORMAT.
   c. Do not assign any of the following names to a SAS table:

      - _NULL_
      - _DATA_
      - _LAST_

Just as SCL recognizes keywords from position and context, it also recognizes names in the same way. If SCL sees a word that meets the requirements for a user-supplied

SAS name and that is not used in a syntax that defines it as anything else, it interprets the word as a variable name.

SCL Keywords

An SCL keyword is a word or symbol in an SCL statement that defines the statement type to SAS software. Keywords are a fixed part of the SCL, and their form and meaning are also fixed. Generally, keywords define the function or CALL routine that you are using in an SCL program statement. For example, OPEN is the keyword in

```
table-id=OPEN('MYLIB.HOUSES');
```

Variables

SCL variables have most of the same attributes as variables in the base SAS language:

- name
- data type
- length.

However, SCL variables do not have labels.

SCL provides three categories of variables:

window variables
: are linked to a control (widget) or field in a window. They pass values between an SCL program and the associated window.

nonwindow variables
: are defined in an SCL program. They hold temporary values that users do not need to see.

system variables
: are provided by SCL. They hold information about the status of an application.

As in the base SAS language, you can group variables into arrays to make it easier to apply the same process to all the variables in a group. Arrays in SCL are described in Chapter 4, “SCL Arrays,” on page 37.

Window Variables

Most SCL programs are associated with a window for interacting with users. An SCL program for a window has variables that are associated with the controls and fields in the window. These variables are called window variables, and they are the means by which users and SCL programs communicate with each other. You can use these variables in the SCL program without explicitly declaring them.

Name

The name of a window variable is the same as the name that is assigned to the control or field. The SCL program for the window cannot change that name.

Data Type

A window variable also has a data type, which can be character, numeric, or an object data type. The type is determined by the value of the **`Type`** attribute, which is displayed in the Properties window (for a control) or in the Attributes window (for a field). For more information about data types that are used in SAS/AF applications, see the SAS/AF online Help and *SAS Guide to Applications Development*.

Length

Lengths of window variables are determined as follows:

- Numeric and object variables are stored internally as doubles.
- Character variables have a maximum length that equals the width of the corresponding field in the application window. For example, if a field occupies 20 columns in the window, then the maximum length of the associated window variable is 20.

SCL programs can use methods to alter the lengths of window variables for some FRAME entry controls. Otherwise, you cannot alter the length of a window variable in an SCL program. Specifying a length for a window variable in a DECLARE or LENGTH statement produces an error message when you compile the program.

Nonwindow Variables

SCL programs can define and use variables that do not have associated controls or fields in the window. These variables are called nonwindow variables, and they are used to hold values that users do not need to see. SCL programs that do not have an associated window use only nonwindow variables. Nonwindow variables are also referred to as program variables. Because nonwindow variables are used only within an SCL program, they have no informat or format.

Name

The name of a nonwindow variable is determined by the first assignment statement that uses the variable, unless the variable is explicitly defined with a DECLARE or LENGTH statement. Names of nonwindow variables can be up to 32 characters long.

Data Type

Nonwindow variables are numeric unless they are explicitly declared as a different data type.

Length

Lengths of nonwindow variables are determined as follows:

- Numeric and object variables are stored as doubles.
- Character variables have a default length of 200. However, you can use the DECLARE statement to change the length from a minimum length of 1 to a maximum of 32K.

You can use the DECLARE or LENGTH statement to specify a different maximum length for nonwindow character variables. This can significantly reduce memory requirements if your program uses many nonwindow variables.

Scope

The scope of a variable determines when a value can be assigned to it and when its value is available for use. In general, variables in an SCL program have program scope. That is, their scope is local to the program. They are available for use within the SCL program but not to other parts of SAS software. When the program finishes, the variables no longer exist, so their values are no longer available.

SCL provides a feature for defining variables as local to a DO or SELECT block. To define a variable with this type of scope, use a DECLARE statement inside a DO or SELECT block. Any variable that you declare in this way exists only for the duration of that DO or SELECT block, and its value is available only during that time. For example, the following program uses two variables named SECOND. One variable is numeric by virtue of the first assignment statement. The other is a character variable that is local to the DO block. After the DO block ends, only the numeric SECOND variable is available.

```
INIT:
   first=10;
   second=5;
   put 'Before the DO block: ' first= second=;
   do;
         /* Declare variable THIRD and new      */
         /* variable SECOND, which is local to  */
         /* the DO block and is CHAR data type  */
      declare char(3) second third;
      second='Jan';
      third ='Mar';
         /* FIRST is available because   */
         /* it comes from parent scope.  */
      put 'Inside the DO block: '
          first= second= third=;
   end;
       /* THIRD is not available because  */
       /* it ended when the DO block ended. */
   put 'After the DO block:  '
       first= second= third=;
return;
```

The example produces the following output:

```
Before the DO block: first=10 second=5
Inside the DO block: first=10 second=Jan third=Mar
After the DO block:  first=10 second=5 third=.
```

Although program variables are available only while an SCL program is running, SCL provides features for passing variables to other programs and also for receiving returned values. For more information, see “ENTRY” on page 369 and “METHOD” on page 539.

You can also use global macro variables to make variables available outside the SCL program. See “Using Macro Variables” on page 87 for details.

System Variables

System variables are created automatically when an SCL program compiles. These variables communicate information between an SCL program and an application, and you can use them in programs. System variables can be Character, Numeric, or Object data type variables. The Object data type facilitates compile-time checking for SCL programs that use dot notation to invoke methods and to access attributes for objects. Although the system variables _CFRAME_, _FRAME_, and _SELF_ are designated as object variables in Version 8 of SAS software, applications that were built with earlier releases and that use these variables will continue to work with Version 8.

Do not declare the _SELF_, _FRAME_, _CFRAME_, _METHOD_, or _EVENT_ system variables inside a CLASS or USECLASS block. SCL automatically sets these values when it is running methods that are defined in CLASS or USECLASS blocks. Redefining any of these system variables can introduce unexpected behavior.

With the exceptions of _EVENT_, _METHOD_, and _VALUE_, you can simply reference a system variable in an SCL program without explicitly declaring it.

BLANK

reports whether a window variable contains a value or sets a variable value to blank.

Type: Character

CFRAME

contains the identifier of the FRAME entry that is currently executing, when a control is executing a method. Otherwise, it stores the identifier of the FRAME entry that is executing.

Type: Object

CURCOL

contains the value of the first column on the left in an extended table object in a FRAME entry. It is used to control horizontal scrolling.

Type: Numeric

CURROW

contains the number of the current row in an extended table.

Type: Numeric

ERROR

contains a code for the application's error status.

Type: Numeric

EVENT

returns the type of event that occurred on a control. It is useful only during a _select method. At other times, it may not exist as an attribute or it is blank. _EVENT_ can have one of the following values:

''	modification or selection
C	command
D	double click
P	pop-up menu request
S	selection or single click.

EVENT must be explicitly declared in an SCL program. For example:

```
declare char(1) _event_;
```

Type: Character.

FRAME

contains the identifier of the FRAME entry that contains a control, when the object is a FRAME entry control. Otherwise, it contains the identifier of the FRAME entry that is currently executing. You can use this variable to send methods to a FRAME entry from a control's method. For example, a control method can send a _refresh method to the FRAME entry, causing the FRAME entry to refresh its display.

Type: Object

METHOD
: contains the name of the method that is currently executing.

METHOD must be explicitly declared in an SCL program. In the declaration statement, specify the maximum length for the name of a method. For example:

```
declare char(40) _method_;
```

Type: Character.

MSG
: assigns text to display on the message line, or contains the text to be displayed on the window's message line the next time the window is refreshed.

Type: Character

SELF
: contains the identifier of the object that is currently executing a method.

Type: Object

STATUS
: contains a code for the status of program execution. You can check for the value of _STATUS_, and you can also set its value.

Type: Character

VALUE
: contains the value of a control.

When _VALUE_ contains the value of a character control, it must be explicitly declared in an SCL program. In the declaration statement, specify the maximum length for a character window control. For example:

```
declare char(80) _value_;
```

Type: Character or Numeric.

Constants

In SCL, a constant (or literal) is a fixed value that can be either a number or a character string. Constants can be used in many SCL statements, including assignment and IF-THEN statements. They can also be used as values for certain options.

Numeric Constants

A numeric constant is a number that appears in a SAS statement, and it can be presented in the following forms:

- standard syntax, in which numeric constants are expressed as integers, can be specified with or without a plus or minus sign, and can include decimal places.
- scientific (E) syntax, in which the number that precedes the E is multiplied by the power of ten indicated by the number that follows the E.
- hexadecimal syntax, in which a numeric hex constant starts with a numeric digit (usually 0), can be followed by more hexadecimal digits, and ends with the letter X. The constant can contain up to 16 hexadecimal digits (0 to 9, A to F).

- special SAS date and time values, in which the date or time is enclosed in single or double quotation marks, followed by a D (date), T (time), or DT (datetime) to indicate the type of value (for example, '15jan99'd).

Character Constants

A character constant can consist of 1 to 32,767 characters and must be enclosed in quotation marks. Character constants can be represented in the following forms:

- hexadecimal form, in which a string of an even number of hex characters is enclosed in single or double quotation marks, followed immediately by an X, as in this example:

  ```
  '534153'x
  ```

- bit form, in which a string of 0s, 1s, and periods is surrounded by quotation marks and is immediately followed by a B. Zero tests whether a bit is off, 1 tests whether a bit is on, and a period ignores a bit. Commas and blanks can be inserted in the bit mask for readability without affecting its meaning.

  In the following example, if the third bit of A (counting from the left) is on, and the fifth through eighth bits are off, then the comparison is true and the expression results in 1. Otherwise, the comparison is false and the expression results in 0.

  ```
  if a='..1.0000'b then do;
  ```

  Bit constants cannot be used as literals in assignment statements. For example, the following statement is not valid:

  ```
  x='0101'b;   /* incorrect */
  ```

If a character constant includes a single quotation mark, then either write the quotation mark as two consecutive single quotation marks or surround the entire value with double quotation marks, as shown in the following examples:

```
possession='Your''s';
company="Your's and Mine"
company="Your""s and Mine"
```

To use a null character value as an argument to a function in SCL, either use `''`(without a space) or use a blank value with `' '`(with a space).

Numeric-to-Character Conversion

If a value is inconsistent with the variable's data type, SCL attempts to convert the value to the expected type. SCL automatically converts character variables to numeric variables and numeric variables to character variables, according to the following rules:

- A character variable is converted to numeric when the character variable is used
  - with an operator that requires numeric operands (for example, the plus sign)
  - with a comparison operator (for example, the equal sign) to compare a character variable and a numeric variable
  - on the right side of an assignment statement, when a numeric variable is on the left side.
- A numeric variable is converted to character when the numeric variable is used
  - with an operator that requires a character value (for example, the concatenation operator)

- on the right side of an assignment statement, when a character variable is on the left side.

When a variable is converted automatically, a message in the LOG window warns you that the conversion took place. If a conversion from character to numeric produces invalid numeric values, then a missing value is assigned to the result, an error message appears in the LOG window, and the value of the automatic variable _ERROR_ is set to 1.

Operators

Operators are symbols that request an arithmetic calculation, a comparison, or a logical operation. SCL includes the same operators that are provided in the base SAS language. The only restrictions on operators in SCL are for the minimum and maximum value operators. For these SAS operators, you must use the operator symbols (> < and < >, respectively) rather than the mnemonic equivalents (MIN and MAX, respectively).

Arithmetic Operators

The arithmetic operators, which designate that an arithmetic calculation is performed, are shown here:

Symbol	Definition
+	addition
/	division
**	exponentiation
*	multiplication
-	subtraction

Comparison Operators

Comparison operators propose a relationship between two quantities and ask whether that relationship is true. Comparison operators can be expressed as symbols or written with letters. An operator that is written with letters, such as EQ for =, is called a mnemonic operator. The symbols for comparison operators and their mnemonic equivalents are shown in the following table:

Symbol	Mnemonic Equivalent	Definition
=	EQ	equal to
^= *	NE	not equal to
¬= *	NE	not equal to
>	GT	greater than
<	LT	less than
>= **	GE	greater than or equal to

Symbol	Mnemonic Equivalent	Definition
<= **	LE	less than or equal to
<>		maximum
><		minimum
||		concatenation
	IN	equal to one item in a list

* The symbol that you use for NE depends on your keyboard.
** The symbols =< and => are also accepted for compatibility with previous releases of SAS.

Colon Modifier

You can add a colon (:) modifier after any operator to compare only a specified prefix of a character string. For example, the following code produces the message **`pen found`**, because the string **`pen`** occurs at the beginning (as a prefix) of **`pencil`**:

```
var='pen';
if var =: 'pencil'
   then put var 'found';
else
   put var 'not found';
```

The following code produces the message **`phone not found`** because **`phone`** occurs at the end (as a suffix) of **`telephone`**:

```
var='phone';
if var =: 'telephone';
   then put var 'found';
else put var 'not found';
```

The code produces these messages:

```
pen found
phone not found
```

IN Operator

The IN operator compares a value produced by an expression on the left side of the operator to a list of values on the right. For example:

```
if age in (16, 21, 25);
```

If the IN operator returns 0 if the value on the left does not match a value in the list. The result is 1 if the value on the left matches a value in the list. In the case of arrays, the IN operator returns the index of the element if it finds a match.

The form of the comparison is

expression IN <*value-1*<, . . . ,*value-n*>)

The elements of the comparison are

expression
: can be any valid SAS expression, but it is usually a variable name when used with the IN operator.

value
: must be a SAS constant. *Value* can be an array of constants.

Suppose you have the following program section:

```
init:
declare a[5] = (2 4 6 8 10);
b = 6;
if b in a then put 'B is in array A';
c=b in a;
put c=;
return;
```

This code produces the following output:

```
B in in array A
c=3
```

Logical (Boolean) Operators

Logical operators (also called Boolean operators) are usually used in expressions to link sequences of comparisons. The logical operators are shown in the following table:

Symbol	Mnemonic Equivalent	Definition
&	AND	AND comparison
|	OR	OR comparison
¬ *	NOT	NOT comparison
^ *	NOT	NOT comparison
~ *	NOT	NOT comparison

* The symbol that you use for NOT depends on your keyboard.

AND Operator

If both conditions compared by an AND operator are true, then the result of the AND operation is true. Two comparisons with a common variable linked by AND can be condensed with an implied AND. For example, the following two subsetting IF statements produce the same result:

```
if 16<=age and age<=65;
if 16<=age<=65;
```

OR Operator

If either condition compared by an OR operator is true, then the result of the OR operation is true.

Be careful when using the OR operator with a series of comparisons (in an IF, SELECT, or WHERE statement, for example). Remember that only one comparison in a series of OR comparisons needs to be true in order to make a condition true. Also, any nonzero, nonmissing constant is always evaluated as true. Therefore, the following subsetting IF statement is always true:

```
if x=1 or 2;
```

Although X=1 may be either true or false, the 2 is evaluated as nonzero and nonmissing, so the entire expression is true. In the following statement, however, the condition is not necessarily true, because either comparison can evaluate as true or false:

```
if x=1 or x=2;
```

You can also use the IN operator with a series of comparisons. The following statements are equivalent:

```
if x in (2, 4, 6);
if x=2 or x=4 or x=6;
```

NOT Operator

Putting NOT in front of a quantity whose value is false makes that condition true. That is, negating a false statement makes the statement true. Putting NOT in front of a quantity whose value is missing is also true. Putting NOT in front of a quantity that has a nonzero, nonmissing value produces a false condition. That is, the result of negating a true statement is false.

Expressions

An SCL expression can be a sequence of operands and operators forming a set of instructions that are performed to produce a result value, or it can be a single variable name, constant, or function. Operands can be variable names or constants, and they can be numeric, character, or both. Operators can be symbols that request a comparison, a logical operation, or an arithmetic calculation. Operators can also be SAS functions and grouping parentheses.

Expressions are used for calculating and assigning new values, for conditional processing, and for transforming variables. These examples show SAS expressions:

- 3
- x
- age<100
- (abc)/2
- min(2,-3,1)

SCL expressions can resolve to numeric, character, or Boolean values. In addition, a numeric expression that contains no logical operators can serve as a Boolean expression.

Boolean Numeric Expressions

In SCL programs, any numeric value other than 0 or missing is true, whereas a value of 0 or missing is false. Therefore, a numeric variable or expression can stand alone in a condition. If the value is a number other than 0 or missing, then the condition is true; if the value is 0 or missing, then the condition is false.

A numeric expression can be simply a numeric constant, as follows:

```
if 5 then do;
```

The numeric value returned by a function is also a valid numeric expression:

```
if index(address,'Avenue') then do;
```

Using Functions in Expressions

You can use functions almost any place in an SCL program statement where you can use variable names or literal values. For example, the following example shows a way to perform an operation (in this case, the FETCH function) and take an action, based on the value of the return code from the function:

```
rc=fetch(dsid);
      /* The return code -1 means the  */
      /* end of the file was reached.  */
   if (rc=-1) then
      do;
      ...SCL statements to handle the
      end-of-file condition...
      end;
```

To eliminate the variable for the return code, you can use the function directly in the IF statement's expression, as shown in the following example:

```
if (fetch(dsid)=-1) then
      do;
       ...SCL statements to handle the
       end-of-file condition...
      end;
```

In this case, the FETCH function is executed, and then the IF expression evaluates the return code to determine whether to perform the conditional action.

As long as you do not need the value of the function's return code for any other processing, the latter form is more efficient because it eliminates the unnecessary variable assignment.

SCL Statements

SCL provides all of the program control statements of the base SAS language. However, many base SAS language statements that relate to the creation and manipulation of SAS tables and external files are absent in SCL. In their place, SCL provides an extensive set of language elements for manipulating SAS tables and external files. These elements are described in Chapter 11, “Using SAS Tables,” on page 165 and in Chapter 12, “Using External Files,” on page 179.

SCL also provides CLASS and INTERFACE statements, which enable you to design and build true object-oriented applications. CLASS statements enable you to define classes from which you can create new objects. The INTERFACE statement enables you to define how applications can communicate with these objects.

Executable and Declarative Statements

As in the base SAS language, SCL statements are either executable or declarative.

executable statements
: are compiled into intermediate code and result in some action when the SCL program is executed. (Examples of executable statements are the CURSOR, IF-THEN/ELSE, and assignment statements.)

declarative statements
: provide information to the SCL compiler but do not result in executable code unless initial values are assigned to the declared variables. (Examples of declarative statements are the DECLARE, LENGTH, and ARRAY statements.)

You can place declarative statements anywhere in an SCL program, but they typically appear at the beginning of the program before the first labeled section.

CAUTION:

Do not place executable statements outside the program modules. Executable statements outside a program module (labeled section, class definition file, method implementation file, and so on) are never executed. See Chapter 2, “The Structure of SCL Programs,” on page 9 for more information about program modules. △

The Assignment Statement

The assignment statement in SCL works like the assignment statement in base SAS except:

- □ You can specify an array name (without the subscript) in the left side of the assignment statement. See “Using Assignment Statements” on page 42 and “Returning Arrays From Methods” on page 45 for more information.
- □ You can use the assignment statement to initialize the values of an SCL list. See “Initializing the Values in a List” on page 51 for more information.

Program Comments

You can include comments anywhere in your SCL programs. Comments provide information to the programmer, but they are ignored by the compiler, and they produce no executable code.

SCL allows the following two forms of comments:

- □ /* *comment* */

```
     /* sort the data set and */
     /* then do something else */
  sysrc=sort(dsid,'year month');
```

- □ * *comment* ;

```
    * sort the data set and  ;
    * then do something else ;
  sysrc=sort(dsid,'year month');
```

SCL Functions

Like the functions in the base SAS language, each SCL function returns a value that is based on one or more arguments that are supplied with the function. Most of the special features of SCL are implemented as functions. In addition, SCL provides all of the functions of the base SAS language except for the DIF and LAG functions. (The DIF and LAG functions require a queue of previously processed rows that only the DATA step maintains.)

SCL functions can be divided into the following groups according to the type of information they return:

- functions that return a value representing the result of a manipulation of the argument values. For example, the MLENGTH function returns the maximum length of a variable.
- functions that perform an action and return a value indicating the success or failure of that action. For these functions, the value that the function returns is called a *return code*. For example, the LIBNAME function returns the value 0 if it successfully assigns a libref to a SAS data library or directory. If the function cannot assign the libref, it returns a nonzero value that reports the failure of the operation. The SYSMSG function returns the text of the error message that is associated with the return code.

  Note: Some functions use a return code value of 0 to indicate that the requested operation was successful, whereas other functions use a return code of 0 to indicate that the operation failed. △

SCL CALL Routines

Like functions, CALL routines perform actions, based on the values of arguments that are supplied with the routine name. However, unlike functions, CALL routines do not return values. Many halt the program if the call is unsuccessful. Use CALL routines to implement features that do not require return codes.

SCL has a variety of CALL routines of its own. It also supports all of the CALL routines that are provided by the base SAS language.

Passing Arguments to SCL Functions and CALL Routines

Some additional restrictions apply to the values that you pass as arguments to SCL functions and CALL routines. Some SCL functions and CALL routines accept only names of variables as arguments, but for most arguments you can specify either a literal value or the name of a variable that contains the desired value.

Note: For some functions, passing missing values for certain arguments causes the SCL program to stop executing and to display an error message. Restrictions on argument values are described in the entries in Chapter 16, “SAS Component Language Dictionary,” on page 249. △

Input, Output, and Update Parameters

Parameters to functions and methods can be one of three types:

input
: The value of the parameter is passed into the function, but even if the function modifies the value, it cannot pass the new value out to the calling function.

output
: Output parameters are used to return a value from a function.

update
: Update parameters can be used to pass a value into a function, and the function can modify its value and return the new value out to the calling function.

Note: If you use dot notation to specify a parameter to a method, then the parameter is treated as an update parameter if the method does not have a signature or if the object is declared as a generic object. SCL executes the _setAttributeValue method for all update parameters, which could cause unwanted effects. See "What Happens When Attribute Values Are Set or Queried" on page 122 for complete information. △

If you do not use dot notation to pass parameters to the functions and routines documented in Chapter 16, "SAS Component Language Dictionary," on page 249, then all parameters are input parameters except for those listed in Table 3.1 on page 35.

Table 3.1 Functions With Update Parameters

Function Name	Update Parameters
DELNITEM	*index*
DIALOG	all *parameters* other than *entry*
DISPLAY	all *parameters* other than *entry*
FGET	*cval*
FILEDIALOG	*filename*
FILLIST	*description*
LVARLEVEL	*n-level*
CALL METHOD	all *parameters* except *entry* and *label*
NAMEDIVIDE	all parameters except *name*
NOTIFY	all *parameters* except *control-name* and *method-name*
RGBDM	*RGB-color*
SAVEENTRYDIALOG	*description*
SEND	all *parameters* except *object-id* and *method-name*
SETNITEMC	*index*
SETNITEML	*index*
SETNITEMN	*index*
SETNITEMO	*index*
SUPER	all *parameters* except *object-id* and *method-name*
VARLEVEL	*n-level*
VARSTAT	*varlist-2*

Note: The *argument* parameter of the DATA step SUBSTR (left of =) function is also an update parameter. △

For all methods that you define with the METHOD statement, all parameters are assumed to be update parameters unless either you specify input or output when you define the method or you invoke the method with SEND, NOTIFY, SUPER, or CALL METHOD. If you invoke the method with SEND, NOTIFY, SUPER, or CALL METHOD, then the first two parameters (listed in Table 3.1 on page 35) are assumed to be input parameters.

Combining Language Elements into Program Statements

The statements that you use in SCL programs must conform to the following rules:

- You must end each SCL program statement with a semicolon.
- You can place any number of SCL program statements on a single line as long as you separate the individual statements with semicolons. If you plan to use the SCL debugger, it is helpful to begin each statement on a separate line.
- You can continue an SCL program statement from one line to the next as long as no keyword is split.
- You can begin SCL program statements in any column.
- You must separate words in SCL program statements with blanks or with special characters such as the equal sign (=) or another operator.
- You must place arguments for SCL functions and CALL routines within parentheses.
- If a function or CALL routine takes more than one argument, you must separate the arguments with commas.
- Character arguments that are literal values must be enclosed in either single or double quotation marks (for example, **'Y'** or **''N''**).
- Numeric arguments cannot be enclosed in quotation marks.

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

SAS® Component Language: Reference, Version 8

Copyright © 1999 by SAS Institute Inc., Cary, NC, USA.

ISBN 1–58025–495–0

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

1st printing, October 1999