9

Data Step Functions

9.1 Introduction to Functions and CALL Routines

SAS® functions and built-in CALL routines perform data manipulation, mathematical calculations, and descriptive statistical computations on argument(s) typically supplied by the user. SAS functions return a single value that can be used in an assignment statement or in additional expressions within the DATA step. In contrast, a CALL routine does not return a value; instead, it only alters the values of its argument. You cannot use a CALL routine in an assignment statement or in an expression.

As of the 9.2 release, SAS provides over 500 functions and CALL routines. Compared to CALL routines, SAS functions are more commonly used in daily programming tasks. Thus, only the introductory material on the CALL routine is presented in this chapter. In addition to SAS documentation, *SAS® Functions by Example* (Cody, 2004) is a good reference for learning SAS functions and CALL routines.

9.1.1 Functions

SAS functions can be used in the DATA step programming statements, WHERE expressions, macro language statements, and the REPORT or SQL procedures. A SAS function typically takes the following form:

```
function-name (argument-1<,...argument-n>)
```

The *argument* in a SAS function can be a variable name, constant, or any valid SAS expression. When a function requires more than one argument, each argument is separated by a comma. Two additional forms are available for programming SAS functions:

```
function-name (OF variable-list)
function-name (<argument | OF variable-list | OF array-name[*]>
       <..., <argument | OF variable-list | OF array-name[*]>>)
```

Based on the syntax above, you can include *variable-list* or *array-name* as the function argument. When using either a variable-list or the name of an array as a function argument, you need to use the OF operator. Variable-lists are described in Chapter 1; Chapter 6 covers arrays.

Program 9.1 illustrates the use of the MEAN function in all its different forms. Notice that the result from mean(of x1–x3) is 4, which is the mean value of X1, X2, and X3. However, the result from mean(x1–x3) is 1 because the mean is calculated for the difference between X1 and X3. When an OF operator is presented in the MEAN function, x1–x3 is treated as a variable-list by SAS. When it is absent, x1–x3 becomes an algebraic expression where X3 is subtracted from X1.

Program 9.1:

```
data ex9_1;
    x1 = 5;
    x2 = 3;
    x3 = 4;
    y1 = 3;
    y2 =.;
    array x[3];
    array y[2];

    m1 = mean(of _numeric_);
    m2 = mean(of x:, 1);
    m3 = mean(of x1— y1);
    m4 = mean(of x1-x3);
    m5 = mean(x1-x3);
    m6 = mean(of x1-x3, of y:);
    m7 = mean(of x[*], of y[*]);
run;

title 'The use of the MEAN function';
proc print data = ex9_1;
    var m1-m7;
run;
```

Output from Program 9.1:

```
             The use of the MEAN function
      Obs     m1      m2     m3    m4   m5    m6     m7
       1     3.75    3.25   3.75   4     1   3.75   3.75
```

9.1.2 CALL Routines

A CALL routine is used to modify variable values. A SAS CALL routine starts with the keyword CALL followed by the name of the routine. A CALL routine takes one of the following forms:

```
CALL routine-name (argument-1<,...argument-n>);
CALL routine-name (OF variable-list);
```

```
CALL routine-name (argument-1 | OF variable-list-1 <,...
     argument-n | OF variable-list-n>);
```

The *argument* in a CALL routine can be a variable name, a constant, or any SAS expression. You can also use variable-lists as the argument of a CALL routine.

The NOISE data set, which was first introduced in Chapter 1, contains the ID, NOISE1, NOISE2, and NOISE3 variables. The first five observations of the data are listed below. Suppose you would like to set all noise scores to missing if their sum is greater than 20.

NOISE:

	ID	NOISE1	NOISE2	NOISE3
1	629F	5	6	9
2	656F	6	10	9
3	711F	0	.	3
4	511F	9	4	10
5	478F	.	5	3
…	…	…	…	…

If the sum of the noise scores is greater than 20, instead of assigning NOISE1 to NOISE3 to missing by using three assignment statements, you can use the CALL MISSING routine to assign these variables to missing at once. The syntax of the CALL MISSING routine is as follows:

```
CALL MISSING(varname1<, varname2,...>);
```

The arguments in the CALL MISSING routine are the names of the variables that you would like to set to missing. The variables can be either character or numeric. The CALL MISSING routine assigns an ordinary numeric value (.) or a character missing value (a blank) to each numeric or character variable in the argument list. You can even mix character and numeric variables in the argument list. Program 9.2 uses CALL MISSING to assign missing values to NOISE1 to NOISE3 when their sum is greater than 20.

Program 9.2:

```
libname ch9 'W:\SAS Book\dat';
data ex9_2;
    set ch9.noise;
    if sum(of noise:)> 20 then call missing (of noise:);
run;

title 'Set SCORES to missing if their sum is Greater than 20';
proc print data = ex9_2 (obs = 5);
run;
```

Output from Program 9.2:

```
Set SCORES to missing if their sum is Greater than 20
    Obs    ID   noise1  noise2  noise3
     1    629F     5       6       9
     2    656F     .       .       .
     3    711F     0       .       3
     4    511F     .       .       .
     5    478F     .       5       3
```


9.1.3 Categories of Functions and CALL Routines

SAS functions and CALL routines can be grouped into more than 20 categories, such as array, descriptive statistics, mathematical, random number, character, date and time, special functions, etc. Three of the array functions (DIM, HBOUND, LBOUND) were presented in Chapter 6. This section covers a few descriptive statistics, mathematical, and random number functions.

SAS provides a large selection of functions for calculating descriptive statistics. The name of the function captures what the function does. For example, the SUM function calculates the sum of its argument or arguments (as the name implies). Some commonly used functions for calculating descriptive statistics are listed in Table 9.1.

Mathematical functions are also available, such as ABS (returns the absolute value), EXP (returns the value of the exponential function),

TABLE 9.1

Descriptive Statistics Functions

Function Name	Returned Value
CMISS	The number of missing arguments
CV	The coefficient of variation
IQR	The interquartile range
LARGEST	The *k*th largest non-missing value
MAX	The largest value
MEAN	The average value
MEDIAN	The median value
MIN	The smallest value
N	The number of non-missing numeric values
NMISS	The number of missing numeric values
PCTL	The percentile that corresponds to the percentage
RANGE	The range of the non-missing values
SUM	The sum of the non-missing arguments
SMALLEST	The *k*th smallest non-missing value
STD	The standard deviation of the non-missing arguments
VAR	The variance of the non-missing arguments

FACT (computes a factorial), GCD (returns the greatest common divisor), LOG (returns the natural logarithm), and SQRT (returns the square root of a value), etc.

The RANUNI function, a random number function, was introduced in Chapter 5. The RANUNI function is used to generate a random number that follows the uniform distribution. In addition, you can generate random numbers from other types of distributions, such as binomial (RANBIN), exponential (RANEXP), normal (RANNOR), Poisson (RANPOI), etc.

9.2 Date and Time Functions

Date and time values are stored as numeric values in SAS. Please refer to Chapter 8 for date and time values, along with date and time formats and informats. Because the dates and times are stored as numeric values internally, you can easily perform computations on date and time values, such as calculating the number of days between two specified dates. However, computations on date and time values can be easily accomplished by using SAS date and time functions.

9.2.1 Creating Date and Time Values

SAS date, time, and datetime values can be created automatically after you read date and time values with appropriate informats. You can create date, time, and datetime constant values directly in the DATA step. The syntax and examples for creating date, time, and datetime constants are listed in Table 9.2.

To obtain the current date, time, and datetime values, you can use the TODAY (or DATE), TIME, and DATETIME functions, respectively, without specifying any arguments. You can also construct a date value by using the MDY and YYQ functions, a time value by using the HMS

TABLE 9.2

Syntaxes and Examples of Date, Time, and Datetime Constants

Constant	Syntax	Example
Date	'ddmmm<yy>yy'D	`'18jul2012'd`
	"ddmmm<yy>yy"D	`"18jul2012"d`
Time	'hh:mm<:ss.s>'T	`'7:34't`
	"hh:mm<:ss.s>"T	`"7:34:30am"t`
Datetime	'ddmmm<yy>yy:hh:mm<:ss.s>'DT	`'18jul2012:7:34'dt`
	"ddmmm<yy>yy:hh:mm<:ss.s>"DT	`"18jul2012:7:34:30am"dt`

TABLE 9.3

Functions for Creating Date and Time Values

Syntax	Value Returned
TODAY()	Current date as a numeric date value
DATE()	Current date as a numeric date value
TIME()	Current time of day as a numeric time value
DATETIME()	Current date and time of day as a datetime value
MDY(*month,day,year*)	Date value from month, day, and year values
YYQ(*year,quarter*)	Date value from year and quarter year values
HMS(*hour,minute,second*)	Time value from hour, minute, and second values
DHMS(*date,hour,minute,second*)	Datetime value from date, hour, minute, and second

function, or the datetime value by using the DHMS function. The syntax of these functions is listed in Table 9.3. Program 9.3 illustrates the use of these functions.

Program 9.3:

```
title 'Constructing date and time values';
data ex9_3;
    current_date = today();
    current_time = time();
    current_datetime = datetime();
    date1 = mdy(8,31,2012);
    date2 = yyq(2012,3);
    time = hms(6,12,30);
    datetime = dhms(current_date, 6,12,30);
    file print;
    put 'current_date:     ' current_date date9.;
    put 'current_time:     ' current_time time.;
    put 'current_datetime: ' current_datetime datetime.;
    put 'date1:            ' date1 date9.;
    put 'date2:            ' date2 date9.;
    put 'time:             ' time time.;
    put 'datetime:         ' datetime datetime.;
run;
```

Output from Program 9.3:

```
             Constructing date and time values
current_date:      18JUL2012
current_time:      9:47:25
current_datetime:  18JUL12:09:47:25
date1:             31AUG2012
date2:             01JUL2012
time:              6:12:30
datetime:          18JUL12:06:12:30
```

9.2.2 Extracting Components from Date and Time Values

In some applications, you might need to know the day, weekday, month, quarter, or year that is based on a single SAS date. These values can be extracted from a date value by using the DAY, WEEKDAY, MONTH, QTR, or YEAR function. Furthermore, you can extract the hour, minute, or second value from either a SAS time or a datetime value. The syntaxes for these functions are listed in Table 9.4.

The next example uses the TENANT data set described in Table 9.5. Program 9.4 shows how to use the YEAR and MONTH functions along with a subsetting IF statement to create a new data set that displays information about those tenants who moved in during August 1987.

TABLE 9.4

Functions for Extracting Components from Data and Time Values

Syntax	Value Returned
YEAR(*date*)	A four-digit numeric value
QTR(*date*)	1–4
MONTH(*date*)	1–12
DAY(*date*)	1–31
WEEKDAY(*date*)	1 (Sunday)–7 (Saturday)
HOUR(<*time* | *datetime*>)	0–23
MINUTE(<*time* | *datetime*>)	0–59
SECOND(<*time* | *datetime*>)	0–<60

TABLE 9.5

Variable Description of TENANT Data

Variable Name	Date Type	Description
ID	Character	Subject ID with four characters. The last character represents gender. For example, 115M
NAME	Character	Subject's full name
AREA_CODE	Character	Three-digit area code
PHONE1	Numeric	The first three digits of the phone number
PHONE2	Numeric	The last four digits of the phone number
MOVE_IN_DATE	Numeric	A date value representing moving date
CP_MONTH	Numeric	The month (1–12) in which complaints were made about the noise
CP_DAY	Numeric	The day (1–31) on which complaints were made about the noise
COMMENTS	Character	Tenants' comments

Program 9.4:

```
data ex9_4;
    set ch9.tenant;
    if year(move_in_date) = 1987 and month(move_in_date) = 8;
run;

title 'People who moved in Aug 1987';
proc print data = ex9_4;
    var id name move_in_date;
run;
```

Output from Program 9.4:

```
             People who moved in Aug 1987
                                            move_in_
      Obs      id       name                  date
       1      483F   LISA WALKER    10AUG1987
       2      904F   LINDA THOMPSON  05AUG1987
```

9.2.3 Date and Time Interval Functions

You can calculate the number of days in a time interval by subtracting the starting date from the ending date. To obtain the number of years in a time interval, you can divide the number of days in a time interval by 365. Alternatively, you can use the INTCK function to count the number of interval boundaries between two dates, two times, or two datetime values. The basic syntax for the INTCK function is as follows:

```
INTCK(interval, start-from, increment)
```

The *interval* argument is used to specify a character constant, a variable, or an expression that contains an interval name. You can use either uppercase or lowercase for the *interval* argument. Examples of the *interval* values are listed in Table 9.6. The *start-from* and *increment* arguments are used to

TABLE 9.6

Examples of Interval Values Used in the INTCK and INTNX Functions

Interval Value	Definition	Default Starting Point
DAY	Daily intervals	Each day
WEEK	Weekly intervals of 7 days	Each Sunday
MONTH	Monthly intervals	First of each month
YEAR	Yearly intervals	January 1
SECOND	Second intervals	Start of the day (midnight)
MINUTE	Minute intervals	Start of the day (midnight)
HOUR	Hourly intervals	Start of the day (midnight)

specify a SAS expression that represents the starting and ending SAS date, time, or datetime values, respectively.

Another useful time-interval function is the INTNX function. The INTNX function is used for generating a date, time, or datetime value by incrementing a date, time, or datetime value by a given time interval. The syntax for the INTNX function is as follows:

```
INTNX(interval, start-from, increment<, 'alignment'>)
```

The use of the *interval* argument is the same as in the INTCK function. The *start-from* argument is used to specify a SAS expression that represents the starting SAS date, time, or datetime value. Unlike the *increment* in the INTCK function, the *increment* in the INTNX function is used to specify a negative, positive, or zero integer that represents the number of intervals to shift the value from the *start-from* argument. The optional *alignment* argument must be enclosed in quotation marks; it is used to control the position of the returned SAS dates within the interval. The *alignment* argument can take one of the following values:

- BEGINNING or B (the default value): The returned date value is aligned to the beginning of the interval.
- MIDDLE or M: The returned date value is aligned to the midpoint of the interval.
- END or E: The returned date value is aligned to the end of the interval.
- SAME or S: The returned date has the same alignment as the input date.

Program 9.5 calculates the number of years since the tenants moved in by using the INTCK function. The INTNX function is used to create a two-year anniversary date from the move-in date.

Program 9.5:

```
data ex9_5;
    set ch9.tenant;
    move_in_years = intck('year', move_in_date, today());
    anniversary2 = intnx('year', move_in_date, 2, 's');
run;

title 'Illustrating the use of INTCK and INTNX functions';
proc print data = ex9_5 (obs = 5);
    var move_in_date move_in_years anniversary2;
    format anniversary2 date9.;
run;
```

Output from Program 9.5:

```
      Illustrating the use of INTCK and INTNX functions
            move_in_ move_in_
     Obs       date      years  anniversary2
      1      09SEP1989    23     09SEP1991
      2      05MAY1995    17     05MAY1997
      3      08NOV1995    17     08NOV1997
      4      03FEB1991    21     03FEB1993
      5      15FEB2000    12     15FEB2002
```

9.3 Character Functions

Often you need to change the case of a text string in SAS. You also might want to concatenate two strings or extract a substring of characters from a larger character string. All these tasks, plus character string searches, can easily be handled by character functions in SAS. Many are available. A few are described below.

9.3.1 Functions for Changing Character Cases

The UPCASE and the LOWCASE functions can change all letters in its argument to uppercase and lowercase, respectively. The syntax for these two functions is as follows:

```
UPCASE(argument)
LOWCASE(argument)
```

The *argument* in these two functions is used to specify a character constant, variable, or expression. If the UPCASE or LOWCASE functions return a value to a variable that has not been previously assigned with a length, the variable will be given the length of the *argument*.

The PROPCASE function can be used to change the words in its argument to proper case. The syntax for the PROPCASE function is as follows:

```
PROPCASE(argument <,delimiters>)
```

The PROPCASE function first converts all uppercase letters in the argument to lowercase and then converts the first character of a *word* to uppercase. A word is defined as a character string preceded by the value specified in the *delimiters* argument. The default value for the delimiter is a blank, forward slash, hyphen, open parenthesis, period, or tab. If the PROPCASE function returns a value to a variable that has not been previously assigned a length, that variable is given a length of 200 bytes.

In Program 9.6, the NEW_NAME variable is created by changing the NAME variable to its proper case by using the PROPCASE function. NEW_COMMENTS is generated by changing each value for COMMENTS to uppercase with the UPCASE function.

Program 9.6:

```
data ex9_6;
    set ch9.tenant;
    new_name = propcase(name);
    new_comments = upcase(comments);
run;

title 'Illustrating the use of PROPCASE and UPCASE functions';
proc print data = ex9_6 (obs = 5);
    var name new_name comments new_comments;
run;
```

Output from Program 9.6:

```
      Illustrating the use of PROPCASE and UPCASE functions
Obs      name           new_name          comments
 1    DAVID DAVIS     David Davis    pounding; complains often
 2    DANIEL THOMAS   Daniel Thomas  sleep OK
 3    JESSICA SCOTT   Jessica Scott  Booming; complains often
 4    THOMAS TAYLOR   Thomas Taylor  Pounding
 5    MARGARET LEWIS  Margaret       Lewis Boom

Obs         new_comments
1      POUNDING; COMPLAINS OFTEN
2      SLEEP OK
3      BOOMING; COMPLAINS OFTEN
4      POUNDING
5      BOOM
```

9.3.2 Functions for Concatenating Character Strings

One way to concatenate characters is to use the concatenation operator (||). The concatenation operator is positioned between the two character variables being concatenated. You can use an assignment statement to store the concatenated strings into a new variable. The length of the resulting variable is the sum of the lengths of each variable or constant in the concatenation operation. Note that the concatenation operator does not trim the leading and trailing blanks in the variables or constants being concatenated. Program 9.7 uses the concatenation operator to concatenate the variables AREA_CODE, PHONE1, PHONE2, and dash characters ('-'). Results are stored in the PHONE variable.

Program 9.7:

```
data ex9_7;
    set ch9.tenant;
    phone = area_code||'-'||phone1||'-'||phone2;
run;

title 'Illustrating the use of the concatenating operator';
proc print data = ex9_7(obs = 5);
    var area_code phone1 phone2 phone;
run;
```

Output from Program 9.7:

```
       Illustrating the use of the concatenating operator
       area_
Obs    code   phone1    phone2          phone
 1      714    545       3799      714-   545-   3799
 2      714    179       5944      714-   179-   5944
 3      714    640       6869      714-   640-   6869
 4      310    165       7540      310-   165-   7540
 5      714    426       2065      714-   426-   2065
```

The variable AREA_CODE is a character variable, but PHONE1 and PHONE2 are numeric variables in the TENANT data set. Because the concatenation operator requires the variables on both sides of the concatenation operator to be characters, SAS performs automatic numeric-to-character conversion by using the BEST12. format. The resulting character values will have leading blanks after the conversion because BEST12. supports right justification. The large gaps within the newly created PHONE variable can be removed by using one or a combination of character alignment functions described in Table 9.7.

Instead of using the concatenation operator, you can use the more convenient CAT, CATT, CATS, and CATX functions to concatenate character strings. The CAT function does not remove leading and trailing blanks in the resulting concatenated character string. The CATT function removes only the trailing blanks, and the CATS function removes both leading and

TABLE 9.7

Functions for Aligning Character Strings and Trimming Blanks

Syntax	Description
LEFT(*argument*)	Left-aligns a character string
RIGHT(*argument*)	Right-aligns a character string
TRIM(*argument*)	Removes trailing blanks from a character string and returns one blank if a string is missing
STRIP(*argument*)	Returns a character string with all leading and trailing blanks removed

TABLE 9.8

Functions for Aligning Character Strings and Trimming Blanks

Example	Equivalent Code
`CAT(OF V1-V3)`	`V1||V2||V3`
`CATT(OF V1-V3)`	`TRIM(V1)||TRIM(V2)||TRIM(V3)`
`CATS(OF V1-V3)`	`TRIM(LEFT(V1))||TRIM((LEFT(V2))|| TRIM(Left(V3))`
`CATX('-',OF V1-V3)`	`TRIM(LEFT(V1))||'-'||TRIM((LEFT(V2))|| '-'||TRIM(Left(V3))`

trailing blanks. The CATX function not only removes both the leading and trailing blanks, it also inserts a *delimiter* between each concatenating item. The syntax for these functions is listed below; notice that they are almost identical. The *item*(s) are used to specify a constant, a variable, or an expression; they can be either character or numeric. If *item* is numeric, it will be converted to a character by using the BEST*w*. format. Some examples for using these functions are listed in Table 9.8.

```
CAT(item-1 <,..., item-n>)
CATT(item-1 <,..., item-n>)
CATS(item-1 <,..., item-n>)
CATX(delimiter, item-1 <,...item-n>)
```

Program 9.8 illustrates how to eliminate the gaps in the concatenated result. The first method uses both the LEFT and TRIM functions along with the concatenation operator. The second method uses the STRIP function with the concatenation operator. The last method uses the CATX function only. All these methods return the same result.

Program 9.8:

```
data ex9_8;
    set ch9.tenant;
    phone_number1 = area_code||'-'||trim(left(phone1))
                    ||'-'||trim(left(phone2));
    phone_number2 = area_code||'-'||strip(phone1)||'-'||
                    strip(phone2);
    phone_number3 = catx('-', area_code, phone1, phone2);
run;

title 'Eliminating the gaps in the PHONE variable';
proc print data = ex9_8 (obs = 5);
    var area_code phone1 phone2 phone_number1
       phone_number2 phone_number3;
run;
```

Output from Program 9.8:

```
       Eliminating the gaps in the PHONE variable
    area_                     phone_        phone_         phone_
Obs code phone1 phone2   number1       number2        number3
1    714   545    3799  714-545-3799 714-545-3799 714-545-3799
2    714   179    5944  714-179-5944 714-179-5944 714-179-5944
3    714   640    6869  714-640-6869 714-640-6869 714-640-6869
4    310   165    7540  310-165-7540 310-165-7540 310-165-7540
5    714   426    2065  714-426-2065 714-426-2065 714-426-2065
```


9.3.3 Functions for Searching, Exacting, and Replacing Character Strings

There is a large selection of functions that can be used for searching, extracting, and replacing character strings. For example, to search a character expression for a string of characters, you can use the INDEX function, which has the following form:

```
INDEX(source, excerpt)
```

Both the *source* and *excerpt* arguments in the INDEX function are used to specify a character constant, variable, or expression. The INDEX function searches *source* from left to right for the first occurrence of the character string that is specified in the *excerpt* argument. If the character string is found, the INDEX function will return the position in *source* of the string's first character. If the character string is not found, the INDEX function will return 0. Note that if there are multiple occurrences of the string, the INDEX function returns only the position of the first occurrence. When a variable is used in the *excerpt* argument, both leading and trailing spaces are also considered part of the *excerpt* argument. You can remove the spaces by using the TRIM or STRIP function with the *excerpt* argument within the INDEX function.

To extract a substring from a character string, you can use either the SCAN or SUBSTR(right of =) functions. The SCAN function is used to extract the *n*th word from a character string. A word is defined as a substring that is separated by delimiters. The default length of the variable that stores the results from a SCAN is 200 bytes unless the variable has been previously defined by the LENGTH statement. The basic form of the SCAN function is as follows:

```
SCAN(string, count <,charlist >)
```

The *string* argument is used to specify a character constant, variable, or expression. The *count* argument is a nonzero integer, variable, or expression that has a nonzero integer value. You use *count* to identify the word by number in *string* that you want to extract. For example, when *count* is set to 1, the first word is returned, 2 returns the second word, and so on. If *count* is positive, the SCAN function counts words from left to right in *string*, and if *count* is negative, the SCAN function will count from right to left. The optional *charlist* argument in the SCAN function is used to identify delimiters that separate *string* into words. If the *charlist* is not specified, the default delimiter (in ASCII environments) is used, including blank ! $% & () * +, -./; < ^ |.

Unlike the SCAN function, the SUBSTR(right of =) function requires that you supply a starting position plus a number of characters to extract a designated substring. The syntax is as follows:

```
<variable=>SUBSTR(string, position<,length>)
```

The reason for listing the *variable* = on the left side of the SUBSTR(right of =) function is to distinguish the SUBSTR(right of =) function from the SUBSTR(left of =) function that is presented next. The *string* argument is used to specify a character constant, variable, or expression from which you want to extract a substring. The *position* and the *length* argument can be a numeric constant, variable, or expression. The *position* is the starting position that you want to extract, and the *length* argument is the length of the substring to extract. If *length* is omitted, SAS will extract the remainder of *string*. The length of the target *variable* will be the length of *string* unless a length has been previously assigned to *variable*.

Program 9.9 illustrates the use of the SCAN, SUBSTR(right of =), and INDEX functions. The SCAN function is used to create the FNAME (first name) and LNAME (last name) variables by extracting the first and second words from the NAME variable. Because the first and last names are separated by blanks, which is a default delimiter, the *charlist* argument is not specified in the SCAN function. Next, the NEWID and GENDER variables are created by using the SUBSTR(right of =) function. NEWID is created by subtracting the first three characters from the ID variable. GENDER is created by subtracting the last character. The INDEX function is used with the subsetting IF statement to keep only the observations with COMMENTS variables that contain 'sleep ok' character strings. The LOWCASE function nested within the INDEX function ensures 'sleep ok' in COMMENTS will retain its original case setting.

Program 9.9:

```
data ex9_9;
    set ch9.tenant;
    length fname lname $ 20 newid $ 3 gender $ 1;
    fname = scan(name, 1);
    lname = scan(name, 2);
    newid = substr(id, 1, 3);
    gender = substr(id, 4);
    if index(lowcase(comments),'sleep ok') > 0;
run;
```

```
title 'The use of SCAN, SUBSTR(right of =) and INDEX functions';
proc print data = ex9_9;
    var name fname lname id newid gender comments;
run;
```

Output from Program 9.9:

```
The use of SCAN, SUBSTR(right of =) and INDEX functions
      Obs         name             fname      lname
       1      DANIEL THOMAS     DANIEL     THOMAS
       2      WILLIAM BROWN     WILLIAM    BROWN
       3      MICHAEL JONES     MICHAEL    JONES
       4      CHARLES WILSON    CHARLES    WILSON
       5      BETTY YOUNG       BETTY      YOUNG
       6      DOROTHY LEE       DOROTHY    LEE
       7      DEBORAH HILL      DEBORAH    HILL
       8      MARK WHITE        MARK       WHITE
       9      MARIA CLARK       MARIA      CLARK
      10      LINDA MILLER      LINDA      MILLER
      11      SANDRA KING       SANDRA     KING

      Obs      id     newid     gender     comments
       1      135M     135         M        sleep OK
       2      478M     478         M        Booming; sleep OK
       3      511M     511         M        sleep OK
       4      590M     590         M        sleep OK
       5      713F     713         F        sleep OK
       6      792F     792         F        sleep OK
       7      793F     793         F        sleep OK
       8      798M     798         M        Booming; sleep OK
       9      823F     823         F        sleep OK
      10      904F     904         F        sleep OK
      11      927F     927         F        Booming; sleep OK
```

In some applications, you may need to replace specific characters or a substring within a string of characters with some character values. For example, you can use the SUBSTR (left of =) function to replace character value contents, which has the following form:

```
SUBSTR(variable, position<,length>) = characters-to-replace
```

Using the SUBSTR(left of =) function is similar to the SUBSTR(right of =) function except that SUBSTR(left of =) is placed on the left side of the equal sign. The *character-to-replace*, which is on the right side of the equal sign, can be a character constant, variable, or expression for replacing the contents of *variable.*

Another useful character replacement function is TRANWRD, which replaces all occurrences of a substring in a character string. The syntax for the TRANWRD function is as follows:

```
TRANWRD(source, target, replacement)
```

All three arguments in the TRANWRD function can be a character constant, variable, or expression. The *source* argument is the string that you want to translate. The *target* argument, with length greater than 0, is a substring that you search for in the *source* argument. The *replacement* is used to replace the *target* in the *source* argument. If the *replacement* string has zero length, the TRANWRD function will use a single blank to replace the *target*. By default, if the TRANWRD function returns a value and is assigned to a variable, this variable will have a default length of 200 bytes.

In the TENANT data set, the last character of the ID is either 'M' or 'F'. Suppose that you would like to replace 'M' with '1' and 'F' with '0'. You can first use the SUBSTR(right of =) function to compare the last character in the ID variable to see if it equals 'M'. If the last character of the ID equals 'M', then you can use the SUBSTR(left of =) function to replace the last character in the ID variable with '1'; otherwise, replace the last character with '0'. For comparison purposes, the ID2 variable is created by copying the values from ID in Program 9.10 first, and then the last character in ID2 is replaced with either '1' or '0'. Program 9.10 also creates a variable, NEW_COMMENTS, by replacing the string 'Booming' with 'Boom'.

Program 9.10:

```
data ex9_10;
    set ch9.tenant;
    id2 = id;
    if substr(id2, 4) = "M" then substr(id2, 4) = "1";
    else substr(id2, 4) = "0";
    new_comments = tranwrd(comments, 'Booming', 'Boom');
run;

title 'The use of SUBSTR(left of =) and TRANWRD functions';
proc print data = ex9_10 (obs = 5);
    var id id2 comments new_comments;
run;
```

Output from Program 9.10:

```
       The use of SUBSTR(left of =) and TRANWRD functions
Obs  id    id2        comments                   new_comments
1   115M  1151   pounding; complains often  pounding; complains
                                            often
2   135M  1351   sleep OK                   sleep OK
3   184F  1840   Booming; complains often   Boom; complains
                                            often
4   188M  1881   Pounding                   Pounding
5   244F  2440   Boom                       Boom
```

9.4 Functions for Converting Variable Types

Sometimes a variable that produces what appears to be numeric output is actually typed as a character. In this situation, you may want to use the INPUT *function* to convert a string into a number. On other occasions you may want to reverse the direction of your conversion so that a number is recast as a character string. PUT, the inverse of INPUT, is what is needed in this instance to do the conversion.

9.4.1 The INPUT Function

In Program 9.11, both INCOME1 and INCOME2 are entered as character variables. In the second DATA step, MONTH_INCOME1 and MONTH_INCOME2 are calculated from INCOME1 and INCOME2. Based on the output generated from Program 9.11, values for MONTH_INCOME1 are correct, whereas MONTH_INCOME2 contains only missing values.

Program 9.11:

```
data income;
    input name $ income1 $ income2 $;
datalines;
John 123000 123,000
Mary 131000 131,000
;
data ex9_11;
    set income;
    month_income1 = income1/12;
    month_income2 = income2/12;
run;

title 'Automatic character-to-numeric conversion';
proc print data = ex9_11;
run;
```

Output from Program 9.11:

```
      Automatic character-to-numeric conversion
                                         month_    month_
  Obs  name    income1    income2    income1   income2
   1   John    123000     123,000   10250.00      .
   2   Mary    131000     131,000   10916.67      .
```

Partial Log from Program 9.11:

```
2515 data ex9_11;
2516   set income;
```

```
2517   month_income1 = income1/12;
2518   month_income2 = income2/12;
2519 run;

NOTE: Character values have been converted to numeric
       values at the places given by: (Line):(Column).
       2517:21 2518:21
NOTE: Invalid numeric data, income2 = '123,000', at line 2518
       column 21.
name = John income1 = 123000 income2 = 123,000
       month_income1 = 10250
month_income2 =. _ERROR_ = 1 _N_ = 1
NOTE: Invalid numeric data, income2 = '131,000', at line 2518
      column 21.
name = Mary income1 = 131000 income2 = 131,000
month_income1 = 10916.666667 month_income2 =. _ERROR_ = 1 _N_ = 2
NOTE: Missing values were generated as a result of performing
      an operation on missing values.
       Each place is given by:
       (Number of times) at (Line):(Column).
       2 at 2518:28
NOTE: There were 2 observations read from the data set
       WORK.INCOME.
NOTE: The data set WORK.EX9_11 has 2 observations and 5
       variables.
NOTE: DATA statement used (Total process time):
       real time     0.00 seconds
       cpu time      0.01 seconds
```

When the second DATA step in Program 9.11 executes, SAS automatically converts the character value in INCOME1 to a numeric value first before dividing it by 12. When the automatic conversion occurs, SAS will issue a message in the log indicating that the conversion has occurred. (See the partial log from Program 9.11.) However, automatic conversion doesn't work for INCOME2.

Automatic conversion occurs only when a character value is assigned to a previously defined numeric variable. Automatic conversion also occurs when a character value is used in an arithmetic operation, compared to a numeric value by using a comparison operator, or specified in a function that requires a numeric argument.

Automatic conversion uses the *w.d* informat. Therefore, if a character value does not conform to standard numeric notation, like the values for INCOME2 with their embedded commas, the conversion will yield to a missing value. In this instance, use the INPUT function to perform the proper character-to-numeric conversion. The INPUT *function* has the following form:

```
INPUT(source, informat)
```

The *source* argument is used to specify a character constant, variable, or expression to which you want to apply an *informat*. The INPUT statement requires that *source* be typed as a character. The *informat* argument is a SAS *informat* that is applied to *source*. Program 9.12 uses the INPUT function to convert both INCOME1 and INCOME2 from character to numeric before any calculations are made. This time, the correct output is generated.

Program 9.12:

```
data ex9_12;
    set income;
    month_income1 = input(income1, 6.)/12;
    month_income2 = input(income2, comma7.)/12;
run;

title 'Using INPUT function to convert character values to
numeric';
proc print data = ex9_12;
run;
```

Output from Program 9.12:

```
Using INPUT function to convert character values to numeric
                                               month_       month_
   Obs    name    income1     income2       income1       income2
    1     John     123000     123,000      10250.00      10250.00
    2     Mary     131000     131,000      10916.67      10916.67
```

When using the INPUT function, you can specify either a numeric or character *informat*. If a character informat is assigned to *informat*, character output is generated, meaning that a character-to-character conversion takes place. For example, in Program 9.13, the INPUT function is used with the *$UPCASEw.* informat to convert mixed-case character strings to uppercase. Note that a character *informat* is referenced because the argument has a leading dollar sign ($).

Program 9.13:

```
data ex9_13;
    set ch9.tenant;
    new_comments = input(comments, $upcase30.);
run;

title 'Using INPUT function to convert character values to
uppercase';
proc print data = ex9_13 (obs = 5);
    var comments new_comments;
run;
```

Output from Program 9.13:

```
Using INPUT function to convert character values to uppercase
 Obs  comments                                 new_comments
  1 pounding; complains often         POUNDING; COMPLAINS OFTEN
  2 sleep OK                          SLEEP OK
  3 Booming; complains often BOOMING; COMPLAINS OFTEN
  4 Pounding                          POUNDING
  5 Boom                              BOOM
```

9.4.2 The PUT Function

SAS can also perform automatic numeric-to-character conversion as illustrated in Program 9.7 where values for PHONE1 and PHONE2 are converted to character strings during the concatenation operation. Similar to the automatic character-to-numeric conversion, SAS will also issue a message in the log when automatic numeric-to-character conversion occurs.

Automatic numeric-to-character conversion occurs when a numeric value is assigned to a previously defined character variable. Automatic conversion also occurs when an *operator* requires a character value (like the concatenation operator '||'), or when a *function* requires a character argument. Instead of relying upon an automatic numeric-to-character conversion, use the PUT function to convert numeric values to character strings. The PUT function has the following form:

```
PUT(source, format)
```

The *source* argument is a constant, variable, or expression whose value you want to reformat. The *source* argument can be either character or numeric. The *format* argument is used to specify the SAS format that you want applied to the value in the *source* argument. *Format* must have the same type as the *source.* For example, if the *source* is character, you will need to use a character format denoted by a leading dollar sign ($); on the other hand, if the *source* is numeric, *format* also has to be numeric. Output from a PUT statement is always character, however, because character and numeric formats always generate character output. Program 9.14 uses the PUT function to apply a numeric-to-character conversion on PHONE1 and PHONE2 before the concatenation operator is applied.

Program 9.14:

```
data ex9_14;
    set ch9.tenant;
    phone_number = area_code||'-'||put(phone1, 3.)||'-'
                   ||put(phone2, 4.);
run;
```

```
title 'Using PUT function to convert numeric values to
character';
proc print data = ex9_14 (obs = 5);
    var area_code phone1 phone2 phone_number;
run;
```

Output from Program 9.14:

```
Using PUT function to convert numeric values to character
             area_
      Obs    code    phone1   phone2    phone_number
       1      714     545      3799     714-545-3799
       2      714     179      5944     714-179-5944
       3      714     640      6869     714-640-6869
       4      310     165      7540     310-165-7540
       5      714     426      2065     714-426-2065
```

In Program 9.13, the INPUT function is used with the *$UPCASEw. informat* to create a new variable, NEW_COMMENTS. You can also use the PUT function to achieve the same result with the *$UPCASEw.* format. (See Program 9.15.)

Program 9.15:

```
data ex9_15;
    set ch9.tenant;
    new_comments = put(comments, $upcase30.);
run;

title 'Using PUT function to reformat the COMMENTS variable';
proc print data = ex9_15 (obs = 5);
    var comments new_comments;
run;
```

Output from Program 9.15:

```
      Using PUT function to reformat the COMMENTS variable
Obs  comments                             new_comments
 1  pounding; complains often            POUNDING; COMPLAINS OFTEN
 2  sleep OK                             SLEEP OK
 3  Booming; complains often BOOMING; COMPLAINS OFTEN
 4  Pounding                             POUNDING
 5  Boom                                 BOOM
```

Knowing when to use INPUT and PUT functions can be confusing. Table 9.9 summarizes the data types used in the *source* arguments and *returned* values for the INPUT and PUT functions. The *source* for INPUT must be in character form, whereas the *source* for PUT can be either numeric or character. On the other hand, *returned values* from INPUT can be either numeric or character, whereas PUT always returns a character value.

TABLE 9.9

Type of *Source, Informat/Format*, and Returned Values in the INPUT/PUT Functions

Function	*Source*	*Informat/Format*	Returned Value
INPUT(*source*, *informat*)	Character	Numeric informat	Numeric
	Character	Character informat	Character
PUT(*source*, *format*)	Numeric	Numeric format	Character
	Character	Character format	Character

TABLE 9.10

Field Description for the ID Variable

Field	Contents
First field	CONSENT: 1 or 0
Second field	GENDER: M or F
Varies: Immediate after the second ID field with length equals 3 to 6	ID
Last field	COUNTY: LA, SB, or OC (Case Varies) Some observations don't have county information

Exercises

Exercise 9.1. In CH9_Q1.SAS7BDAT, there is only one variable named ID. Each field of ID is described in Table 9.10. For this exercise, create the following variables:

- CONSENT: With values equaling 1 or 0.
- GENDER: With values equaling M or F.
- PATIENTID: With length equaling 6. Because the Patient ID with length varies from 3 to 6 from the original ID variable, add zeros at the very beginning. For example, if the original Patient ID is 123, then the newly created variable will be 000123; if the original Patient ID is 1234, then the new value will be 001234; etc.
- COUNTY: With values equaling LA, SB, or OC.

Exercise 9.2. The LAG function is used to retrieve values from previous observations, whereas DIF calculates the difference between the value from the current and previous observations. Often these two special functions are used together to simplify DATA step programming. In this exercise, read the SAS documentation on the LAG and DIFF functions and then create two variables, HEIGHT_LAST and HEIGHT_DIFF, by processing the HEIGHT. SAS7BDAT data set. HEIGHT_LAST will contain the heights of students in their previous grade, and HEIGHT_DIFF will contain the differences in height between their current and previous grade. The new output data set will look like the one below:

	NAME	GRADE	HEIGHT	HEIGHT_LAST	HEIGHT_DIFF
1	John	6	58.5	.	.
2	John	7	58.8	58.5	0.3
3	John	8	59.2	58.8	0.4
4	Mary	6	57.3	.	.
5	Mary	7	58.0	57.3	0.7
6	Mary	8	60.1	58.0	2.1