Chapter 1
Character Functions

Introduction

A major strength of SAS is its ability to work with character data. The SAS character functions are essential to this. The collection of functions and call routines in this chapter allow you to do extensive manipulation on all sorts of character data.

SAS users who are new to Version 9 will notice the tremendous increase in the number of SAS character functions. You will also want to review the next chapter on Perl regular expressions, another way to process character data.

Before delving into the realm of character functions, it is important to understand how SAS stores character data and how the length of character variables gets assigned.

Storage Length for Character Variables

It is in the compile stage of the DATA step that SAS variables are determined to be character or numeric, that the storage lengths of SAS character variables are determined, and that the descriptor portion of the SAS data set is written. The program below will help you to understand how character storage lengths are determined:

Program 1.1: How SAS determines storage lengths of character variables

```
DATA EXAMPLE1;
   INPUT GROUP $
     @10 STRING $3.;
   LEFT  = 'X    '; *X AND 4 BLANKS;
   RIGHT = '    X'; *4 BLANKS AND X;
   SUB = SUBSTR(GROUP,1,2);
   REP = REPEAT(GROUP,1);
DATALINES;
ABCDEFGH 123
XXX        4
Y        5
;
```

Explanation

The purpose of this program is not to demonstrate SAS character functions. That is why the functions in this program are not highlighted as they are in all the other programs in this book. Let's look at each of the character variables created in this DATA step. To see the storage length for each of the variables in data set EXAMPLE1, let's run PROC CONTENTS. Here is the program:

Program 1.2: Running PROC CONTENTS to determine storage lengths

```
PROC CONTENTS DATA=EXAMPLE1 VARNUM;
   TITLE "PROC CONTENTS for Data Set EXAMPLE1";
RUN;
```

The VARNUM option requests the variables to be in the order that they appear in the SAS data set, rather than the default, alphabetical order. The output is shown next:

```
                -----Variables Ordered by Position-----

                     #    Variable    Type    Len
                     ____________________________
                     1    GROUP       Char      8
                     2    STRING      Char      3
                     3    LEFT        Char      5
                     4    RIGHT       Char      5
                     5    SUB         Char      8
                     6    REP         Char    200
```

First, GROUP is read using list input. No informat is used, so SAS will give the variable the default length of 8. Since STRING is read with an informat, the length is set to the informat width of 3. LEFT and RIGHT are both created with an assignment statement. Therefore the length of these two variables is equal to the number of bytes in the literals following the equal sign. Note that if a variable appears several times in a DATA step, its length is determined by the **first** reference to that variable.

For example, beginning SAS programmers often get in trouble with statements such as:

```
IF SEX = 1 THEN GENDER = 'MALE';
ELSE IF SEX = 2 THEN GENDER = 'FEMALE';
```

The length of GENDER in the two lines above is 4, since the statement in which the variable first appears defines its length.

There are several ways to make sure a character variable is assigned the proper length. Probably the best way is to use a LENGTH statement. So, if you precede the two lines above with the statement:

```
LENGTH GENDER $ 6;
```

the length of GENDER will be 6, not 4. Some lazy programmers will "cheat" by adding two blanks after MALE in the assignment statement (me, never!). Another trick is to place the line for FEMALE first.

So, continuing on to the last two variables. You see a length of 8 for the variable SUB. As you will see later in this chapter, the SUBSTR (substring) function can extract some or all of one string and assign the result to a new variable. Since SAS has to determine variable lengths in the compile stage and since the SUBSTR arguments that define the starting point and the length of the substring could possibly be determined in the execution stage (from data values, for example), SAS does the logical thing: it gives the variable defined by the SUBSTR function the longest length it possibly could—the length of the string from which you are taking the substring.

Finally, the variable REP is created by using the REPEAT function. As you will find out later in this chapter, the REPEAT function takes a string and repeats it as many times as directed by the second argument to the function. Using the same logic as the SUBSTR function, since the length of REP is determined in the compile stage and since the number of repetitions could vary, SAS gives it a default length of 200. A note of historical interest: Prior to Version 7, the maximum length of character variables was 200. With the coming of Version 7, the maximum length of character variables was increased to 32,767. SAS made a very wise decision to leave the default length for situations such as the REPEAT function described here, at 200. The take-home message is that you should always be sure that you know the storage lengths of your character variables.

Functions That Change the Case of Characters

Two old functions, UPCASE and LOWCASE, change the case of characters. A new function (as of Version 9), PROPCASE (proper case) capitalizes the first letter of each word.

Function: UPCASE

Purpose: To change all letters to uppercase.
Note: The corresponding function LOWCASE changes uppercase to lowercase.

Syntax: **UPCASE(*character-value*)**

character-value is any SAS character expression.

If a length has not been previously assigned, the length of the resulting variable will be the length of the argument.

Examples

For these examples CHAR = "ABCxyz"

Function	Returns
UPCASE(CHAR)	"ABCXYZ"
UPCASE("a1%m?")	"A1%M?"

Program 1.3: Changing lowercase to uppercase for all character variables in a data set

```
***Primary function: UPCASE
***Other function: DIM;

DATA MIXED;
   LENGTH A B C D E $ 1;
   INPUT A B C D E X Y;
DATALINES;
M f P p D 1 2
m f m F M 3 4
;
DATA UPPER;
   SET MIXED;
   ARRAY ALL_C[*] _CHARACTER_;
   DO I = 1 TO DIM(ALL_C);
      ALL_C[I] = UPCASE(ALL_C[I]);
   END;
```

```
   DROP I;
RUN;

PROC PRINT DATA=UPPER NOOBS;
   TITLE 'Listing of Data Set UPPER';
RUN;
```

Explanation

Remember that upper- and lowercase values are represented by different internal codes, so if you are testing for a value such as Y for a variable and the actual value is y, you will not get a match. Therefore it is often useful to convert all character values to either upper- or lowercase before doing your logical comparisons. In this program, _CHARACTER_ is used in the array statement to represent all the character variables in the data set MIXED. Inspection of the listing below verifies that all lowercase values were changed to uppercase.

```
                        Listing of Data Set UPPER

                A    B    C    D    E    X    Y

                M    F    P    P    D    1    2
                M    F    M    F    M    3    4
```

Function: LOWCASE

Purpose: To change all letters to lowercase.

Syntax: **LOWCASE(*character-value*)**

character-value is any SAS character expression.

Note: The corresponding function UPCASE changes lowercase to uppercase.

If a length has not been previously assigned, the length of the resulting variable will be the length of the argument.

Examples

For these examples CHAR = "ABCxyz"

Function	Returns
LOWCASE(CHAR)	"abcxyz"
LOWCASE("A1%M?")	"a1%m?"

Program 1.4: Program to capitalize the first letter of the first and last name (using SUBSTR)

```
***Primary functions: LOWCASE, UPCASE
***Other function: SUBSTR (used on the left and right side of the equal
sign);

DATA CAPITALIZE;
   INFORMAT FIRST LAST $30.;
   INPUT FIRST LAST;
   FIRST = LOWCASE(FIRST);
   LAST = LOWCASE(LAST);
   SUBSTR(FIRST,1,1) = UPCASE(SUBSTR(FIRST,1,1));
   SUBSTR(LAST,1,1) = UPCASE(SUBSTR(LAST,1,1));
DATALINES;
ronald cODy
THomaS eDISON
albert einstein
;
PROC PRINT DATA=CAPITALIZE NOOBS;
   TITLE "Listing of Data Set CAPITALIZE";
RUN;
```

Explanation

Before we get started on the explanation, I should point out that as of Version 9, the PROPCASE function capitalizes the first letter of each word in a string. However, it provides a good demonstation of the LOWCASE and UPCASE functions and this method will still be useful for SAS users using earlier versions of SAS software.

This program capitalizes the first letter of the two character variables FIRST and LAST. The same technique could have other applications. The first step is to set all the letters to lowercase using the LOWCASE function. The first letter of each name is then turned back to uppercase using the SUBSTR function (on the right side of the equal sign) to select the first letter in the first and last names, and the UPCASE function to capitalize it. The

SUBSTR function on the left side of the equal sign is used to place this letter in the first position of each of the variables. The listing below shows that this program worked as desired:

```
                    Listing of Data Set CAPITALIZE

                    FIRST     LAST

                    Ronald    Cody
                    Thomas    Edison
                    Albert    Einstein
```

SAS9.1 Function: PROPCASE

Purpose: To capitalize the first letter of each word in a string.

Syntax: **PROPCASE(*character-value*)**

character-value is any SAS character expression.

If a length has not been previously assigned, the length of the resulting variable will be the length of the argument.

Examples

For these examples CHAR = "ABCxyz"

Function	Returns
PROPCASE(CHAR)	"Abcxyz"
PROPCASE("a1%m?")	"A1%m?"
PROPCASE("mr. george w. bush")	"Mr. George W. Bush"

Program 1.5: Capitalizing the first letter of each word in a string

```
***Primary function: PROPCASE;

DATA PROPER;
   INPUT NAME $60.;
   NAME = PROPCASE(NAME);
```

```
DATALINES;
ronald cODy
THomaS eDISON
albert einstein
;
PROC PRINT DATA=PROPER NOOBS;
   TITLE "Listing of Data Set PROPER";
RUN;
```

Explanation

In this program, you use the PROPCASE function to capitalize the first letter of the first and last names. The listing is shown below:

```
                    Listing of Data Set PROPER

                    NAME

                    Ronald Cody
                    Thomas Edison
                    Albert Einstein
```

Program 1.6: Alternative program to capitalize the first letter of each word in a string

```
***First and last name are two separate variables.

DATA PROPER;
   INFORMAT FIRST LAST $30.;
   INPUT FIRST LAST;
   LENGTH NAME $ 60;
   CALL CATX(' ', NAME, FIRST, LAST);
   NAME = PROPCASE(NAME);
DATALINES;
ronald cODy
THomaS eDISON
albert einstein
;
PROC PRINT DATA=PROPER NOOBS;
   TITLE "Listing of Data Set PROPER";
RUN;
```

Explanation

In this alternative program, the CATX call routine is used to concatenate the first and last name with a blank as the separator character. The PROPCASE function is then used the same way as above. The listing is identical to the listing above.

Functions That Remove Characters from Strings

COMPBL (compress blanks) can replace multiple blanks with a single blank. The COMPRESS function can remove not only blanks, but also any characters you specify from a string.

Function: COMPBL

Purpose: To replace all occurrences of two or more blanks with a single blank character. This is particularly useful for standardizing addresses and names where multiple blanks may have been entered.

Syntax: **COMPBL(*character-value*)**

character-value is any SAS character expression.

If a length has not been previously assigned, the length of the resulting variable will be the length of the argument.

Example

For these examples `CHAR = "A C  XYZ"`

Function	Returns
`COMPBL(CHAR)`	`"A C XYZ"`
`COMPBL("X Y  Z  LAST")`	`"X Y Z LAST"`

Program 1.7: Using the COMPBL function to convert multiple blanks to a single blank

```
***Primary function: COMPBL;

DATA SQUEEZE;
   INPUT #1 @1  NAME    $20.
         #2 @1  ADDRESS $30.
         #3 @1  CITY    $15.
            @20 STATE    $2.
            @25 ZIP      $5.;
   NAME = COMPBL(NAME);
   ADDRESS = COMPBL(ADDRESS);
   CITY = COMPBL(CITY);
DATALINES;
RON CODY
89 LAZY BROOK ROAD
FLEMINGTON         NJ   08822
BILL     BROWN
28   CATHY   STREET
NORTH   CITY       NY   11518
;
PROC PRINT DATA=SQUEEZE;
   TITLE 'Listing of Data Set SQUEEZE';
   ID NAME;
   VAR ADDRESS CITY STATE ZIP;
RUN;
```

Explanation

Each line of the addresses was passed through the COMPBL function to replace any sequence of two or more blanks to a single blank. A listing of data set SQUEEZE is shown below:

Listing of Data Set SQUEEZE

NAME	ADDRESS	CITY	STATE	ZIP
RON CODY	89 LAZY BROOK ROAD	FLEMINGTON	NJ	08822
BILL BROWN	28 CATHY STREET	NORTH CITY	NY	11518

Function: COMPRESS

Purpose: To remove specified characters from a character value.

Syntax: **COMPRESS(*character-value* <,'*compress-list*'>)**

character-value is any SAS character expression.

compress-list is an optional list of the characters you want to remove. If this argument is omitted, the default character to be removed is a blank. If you include a list of values to remove, only those characters will be removed. If a blank is not included in the list, blanks will not be removed.

If a length has not been previously assigned, the length of the resulting variable will be the length of the argument.

Examples

In the examples below, `CHAR = "A C123XYZ"`

Function	Returns
`COMPRESS("A C XYZ")`	`"ACXYZ"`
`COMPRESS("(908) 777-1234"," (-)")`	`"9087771234"`
`COMPRESS(CHAR,"0123456789")`	`"A CXYZ"`

Program 1.8: Removing dashes and parentheses from phone numbers

```
***Primary function: COMPRESS;

DATA PHONE_NUMBER;
   INPUT PHONE $ 1-15;
   PHONE1 = COMPRESS(PHONE);
   PHONE2 = COMPRESS(PHONE,'(-) ');
DATALINES;
(908)235-4490
(201) 555-77 99
;
PROC PRINT DATA=PHONE_NUMBER;
   TITLE 'Listing of Data Set PHONE_NUMBER';
RUN;
```

Explanation

For the variable PHONE1, the second argument is omitted from the COMPRESS function; therefore, only blanks are removed. For PHONE2, left and right parentheses, dashes, and blanks are listed in the second argument so all of these characters are removed from the character value. You can verify this by inspecting the listing below:

Listing of Data Set PHONE_NUMBER

Obs	PHONE	PHONE1	PHONE2
1	(908)235-4490	(908)235-4490	9082354490
2	(201) 555-77 99	(201)555-7799	2015557799

Converting Social Security Numbers to Numeric Form

Here is another example where the COMPRESS function makes it easy to convert a standard social security number, including the dashes, to a numeric value.

Program 1.9: Converting social security numbers from character to numeric

```
***Primary function: COMPRESS
***Other function:  INPUT;

DATA SOCIAL;
   INPUT @1 SS_CHAR $11.
         @1 MIKE_ZDEB COMMA11.;
   SS_NUMERIC = INPUT(COMPRESS(SS_CHAR,'-'),9.);
   SS_FORMATTED = SS_NUMERIC;
   FORMAT SS_FORMATTED SSN.;
DATALINES;
123-45-6789
001-11-1111
;
PROC PRINT DATA=SOCIAL NOOBS;
   TITLE "Listing of Data Set SOCIAL";
RUN;
```

Explanation

The COMPRESS function is used to remove the dashes from the social security number and the INPUT function does the character to numeric conversion.

It should be noted here that the social security number, including dashes, can be read directly into a numeric variable using the comma11. informat. This trick was brought to light by Mike Zdeb in a NESUG workshop in Buffalo in the Fall of 2002. Here, the variable SS_FORMATTED is set equal to the variable SS_NUMERIC so that you can see the effect of adding the SSN. format. (Note: SSN. is equivalent to SSN11.) This format prints numeric values with leading zeros and dashes in the proper places, as you can see in the listing below:

```
                    Listing of Data Set SOCIAL

     SS_CHAR       MIKE_ZDEB     SS_NUMERIC     SS_FORMATTED

  123-45-6789     123456789      123456789     123-45-6789
  001-11-1111       1111111        1111111     001-11-1111
```

Counting the Number of Digits in a Character String

This program computes the number of numerals (i.e., digits) in a string by a novel method. It uses the COMPRESS function to remove all digits from the string and then subtracts the resulting length from the original length for the computation.

Program 1.10: Counting the number of numerals in a string

```
***Primary functions: COMPRESS, LENGTHN;

DATA COUNT;
   INPUT STRING $20.;
   ONLY_LETTERS = COMPRESS(STRING,'0123456789');
   NUM_NUMERALS = LENGTHN(STRING) - LENGTHN(ONLY_LETTERS);
DATALINES;
ABC123XYZ
XXXXX
12345
1234X
;
```

```
PROC PRINT DATA=COUNT NOOBS;
   TITLE "Listing of Data Set COUNT";
RUN;
```

Explanation

This is an interesting application of the COMPRESS function. By computing the length of the string before and after removing the numerals, this program sets the difference in the lengths to the number of numerals in the original string. Notice the use of the LENGTHN function instead of the LENGTH function. When the COMPRESS function operates on the third observation (all digits), the result is a null string. The LENGTH function returns a value of 1 in this situation; the LENGTHN function returns a value of 0. See LENGTH and LENGTHN function descriptions for a detailed explanation.

Listing of Data Set COUNT

STRING	ONLY_ LETTERS	NUM_ NUMERALS
ABC123XYZ	ABCXYZ	3
XXXXX	XXXXX	0
12345		5
1234X	X	4

Functions That Search for Characters

Functions in this category allow you to search a string for specific characters or for a character category (such as a digit). Some of these functions can also locate the first position in a string where a character does not meet a particular specification. Quite a few of the functions in this section are new to Version 9 and they provide some new and useful capabilities.

The "ANY" functions (ANYALNUM, ANYALPHA, ANYDIGIT, ANYPUNCT, and ANYSPACE)

This group of functions is described together because of the similarity of their use. New as of Version 9, these functions return the location of the first alphanumeric, letter, digit, punctuation, or space in a character string. Note that there are other "ANY" functions

besides those presented here—these are the most common ones (see the *SAS OnlineDoc 9.1* for a complete list). The functionality of this group of functions is similar to many of the Perl regular expressions that are also available in Version 9.

It is important to note that it may be necessary to use the TRIM function (or STRIP function) with the ANY and NOT functions since leading or, especially, trailing blanks will affect the results. For example, if X = "ABC " (ABC followed by three blanks), Y = NOTALNUM(X) will be 4, the location of the first blank. Therefore, you may want to routinely use TRIM (or STRIP) like this:

```
Y = NOT or ANY function(TRIM(X));
```

Note that there are a group of similar functions NOTALPHA, NOTDIGIT, etc. that work in a similar manner and are described together later in the next section. One program example follows the description of these five functions.

SAS9.1 Function: ANYALNUM

Purpose: To locate the first occurrence of an alphanumeric character (any upper- or lowercase letter or number) and return its position. If none is found, the function returns a 0. With the use of an optional parameter, this function can begin searching at any position in the string and can also search from right to left, if desired.

Syntax: **ANYALNUM(*character-value* <,*start*>)**

character-value is any SAS character expression.

start is an optional parameter that specifies the position in the string to begin the search. If it is omitted, the search starts at the beginning of the string. If it is non-zero, the search begins at the position in the string of the absolute value of the number (starting from the left-most position in the string). If the start value is positive, the search goes from left to right; if the value is negative, the search goes from right to left. A negative value larger than the length of the string results in a scan from right to left, starting at the end of the string. If the value of *start* is a positive number longer than the length of the string, or if it is 0, the function returns a 0.

Examples

For these examples, `STRING = "ABC 123 ?xyz_n_"`

Function	Returns
`ANYALNUM(STRING)`	`1` (the position of `"A"`)
`ANYALNUM("??$$%%")`	`0` (no alpha-numeric characters)
`ANYALNUM(STRING,5)`	`5` (the position of `"1"`)
`ANYALNUM(STRING,-4)`	`3` (the position of `"C"`)
`ANYALNUM(STRING,6)`	`6` (the position of `"2"`)

SAS9.1 Function: ANYALPHA

Purpose: To locate the first occurrence of an alpha character (any upper- or lowercase letter) and return its position. If none is found, the function returns a 0. With the use of an optional parameter, this function can begin searching at any position in the string and can also search from right to left, if desired.

Syntax: **`ANYALPHA(`*`character-value`* **`<,`***`start`***`>)`**

`character-value` is any SAS character expression.

`start` is an optional parameter that specifies the position in the string to begin the search. If it is omitted, the search starts at the beginning of the string. If it is non-zero, the search begins at the position in the string of the absolute value of the number (starting from the left-most position in the string). If the start value is positive, the search goes from left to right; if the value is negative, the search goes from right to left. A negative value larger than the length of the string results in a scan from right to left, starting at the end of the string. If the value of *`start`* is a positive number longer than the length of the string, or if it is 0, the function returns a 0.

Examples

For these examples, `STRING = "ABC 123 ?xyz_n_"`

Function	Returns
ANYALPHA(STRING)	1 (position of "A")
ANYALPHA("??$$%%")	0 (no alpha characters)
ANYALPHA(STRING,5)	10 (position of "x")
ANYALPHA(STRING,-4)	3 (position of "C")
ANYALPHA(STRING,6)	10 (position of "x")

SAS9.1 Function: ANYDIGIT

Purpose: To locate the first occurrence of a digit (numeral) and return its position. If none is found, the function returns a 0. With the use of an optional parameter, this function can begin searching at any position in the string and can also search from right to left, if desired.

Syntax: **ANYDIGIT(*character-value* <,*start*>)**

character-value is any SAS character expression.

start is an optional parameter that specifies the position in the string to begin the search. If it is omitted, the search starts at the beginning of the string. If it is non-zero, the search begins at the position in the string of the absolute value of the number (starting from the left-most position in the string). If the start value is positive, the search goes from left to right; if the value is negative, the search goes from right to left. A negative value larger than the length of the string results in a scan from right to left, starting at the end of the string. If the value of *start* is a positive number longer than the length of the string, or if it is 0, the function returns a 0.

Examples

For these examples, `STRING = "ABC 123 ?xyz_n_"`

Function	Returns
ANYDIGIT(STRING)	5 (position of "1")
ANYDIGIT("??$$%%")	0 (no digits)
ANYDIGIT(STRING,5)	5 (position of "1")
ANYDIGIT(STRING,-4)	0 (no digits from position 4 to 1)
ANYDIGIT(STRING,6)	6 (position of "2")

SAS9.1 Function: ANYPUNCT

Purpose: To locate the first occurrence of a punctuation character and return its position. If none is found, the function returns a 0. With the use of an optional parameter, this function can begin searching at any position in the string and can also search from right to left, if desired.

In the ASCII character set, the following characters are considered punctuation:

```
!  "  #  $  %  &  '  (  )  *  +  ,  -  .  /  :  ;
<  =  >  ?  @  [  \ ]  ^  _  `  {  |  }  ~
```

Syntax: **ANYPUNCT(*character-value* <,*start*>)**

character-value is any SAS character expression.

start is an optional parameter that specifies the position in the string to begin the search. If it is omitted, the search starts at the beginning of the string. If it is non-zero, the search begins at the position in the string of the absolute value of the number (starting from the left-most position in the string). If the start value is positive, the search goes from left to right; if the value is negative, the search goes from right to left. A negative value larger than the length of the string results in a scan from right to left, starting at the end of the string. If the value of *start* is a positive number longer than the length of the string, or if it is 0, the function returns a 0.

Examples

For these examples, `STRING = "A!C 123 ?xyz_n_"`

Function	Returns
`ANYPUNCT(STRING)`	`2` (position of "!")
`ANYPUNCT("??$$%%")`	`1` (position of "?")
`ANYPUNCT(STRING,5)`	`9` (position of "?")
`ANYPUNCT(STRING,-4)`	`2` (starts at position 4 and goes left, position of "!")
`ANYPUNCT(STRING,-3)`	`2` (starts at "C" and goes left, position of "!")

SAS9.1 Function: ANYSPACE

Purpose: To locate the first occurrence of a white space character (a blank, horizontal or vertical tab, carriage return, linefeed, and form-feed) and return its position. If none is found, the function returns a 0. With the use of an optional parameter, this function can begin searching at any position in the string and can also search from right to left, if desired.

Syntax: **`ANYSPACE(`*`character-value`*` <,`*`start`*`>)`**

`character-value` is any SAS character expression.

`start` is an optional parameter that specifies the position in the string to begin the search. If it is omitted, the search starts at the beginning of the string. If it is non-zero, the search begins at the position in the string of the absolute value of the number (starting from the left-most position in the string). If the start value is positive, the search goes from left to right; if the value is negative, the search goes from right to left. A negative value larger than the length of the string results in a scan from right to left, starting at the end of the string. If the value of *`start`* is a positive number longer than the length of the string, or if it is 0, the function returns a 0.

Examples

For these examples, `STRING = "ABC 123 ?xyz_n_"`

Function	Returns
ANYSPACE(STRING)	4 (position of the first blank)
ANYSPACE("??$$%%")	0 (no spaces)
ANYSPACE(STRING,5)	8 (position of the second blank)
ANYSPACE(STRING,-4)	4 (position of the first blank)
ANYSPACE(STRING,6)	8 (position of the second blank)

Program 1.11: Demonstrating the "ANY" character functions

```
***Primary functions: ANYALNUM, ANYALPHA, ANYDIGIT, ANYPUNCT, and
ANYSPACE;

DATA ANYWHERE;
   INPUT STRING $CHAR20.;
   ALPHA_NUM   = ANYALNUM(STRING);
   ALPHA_NUM_9 = ANYALNUM(STRING,-999);
   ALPHA       = ANYALPHA(STRING);
   ALPHA_5     = ANYALPHA(STRING,-5);
   DIGIT       = ANYDIGIT(STRING);
   DIGIT_9     = ANYDIGIT(STRING,-999);
   PUNCT       = ANYPUNCT(STRING);
   SPACE       = ANYSPACE(STRING);
DATALINES;
Once upon a time 123
HELP!
987654321
;
PROC PRINT DATA=ANYWHERE NOOBS HEADING=H;
   TITLE "Listing of Data Set ANYWHERE";
RUN;
```

Explanation

Each of these "ANY" functions works in a similar manner, the only difference being in the types of character values it is searching for. The two statements using a starting value of –999 demonstrate an easy way to search from right to left, without having to know the length of the string (assuming that you don't have any strings longer than 999, in which case

you could choose a larger number). Functions such as ANYALPHA and ANYDIGIT can be very useful for extracting values from strings where the positions of digits or letters are not fixed. An alternative to using this group of functions would be the Perl regular expressions. See the following chapter for a complete discussion of regular expressions. Notice in the listing below that the position of the first space in lines two and three are 6 and 10, respectively. These are the positions of the first trailing blank in each of the two strings (remember that the length of STRING is 20).

Listing of Data Set ANYWHERE

STRING	ALPHA_ NUM	ALPHA_ NUM_9	ALPHA	ALPHA_5	DIGIT	DIGIT_9	PUNCT	SPACE
Once upon a time 123	1	20	1	4	18	20	0	5
HELP!	1	4	1	4	0	0	5	6
987654321	1	9	0	0	1	9	0	10

Program 1.12: Using the functions ANYDIGIT and ANYSPACE to find the first number in a string

```
***Primary functions: ANYDIGIT and ANYSPACE
***Other functions: INPUT and SUBSTR;

DATA SEARCH_NUM;
   INPUT STRING $60.;
   START = ANYDIGIT(STRING);
   END = ANYSPACE(STRING,START);
   IF START NE 0 THEN
      NUM = INPUT(SUBSTR(STRING,START,END-START),9.);
DATALINES;
This line has a 56 in it
two numbers 123 and 456 in this line
No digits here
;
PROC PRINT DATA=SEARCH_NUM NOOBS;
   TITLE "Listing of Data Set SEARCH_NUM";
RUN;
```

Explanation

This program identifies the first number in any line of data that contains a numeric value (followed by one or more blanks). The ANYDIGIT function determines the position of the first digit of the number; the ANYSPACE function searches for the first blank following the number (the starting position of this search is the position of the first digit). The SUBSTR function extracts the digits (starting at the value of START with a length determined by the difference between END and START). Finally, the INPUT function performs the character to numeric conversion. Inspect the listing below to see that this program works as expected.

Listing of Data Set SEARCH_NUM

STRING	START	END	NUM
This line has a 56 in it	17	19	56
two numbers 123 and 456 in this line	13	16	123
No digits here	0	0	.

The "NOT" functions (NOTALNUM, NOTALPHA, NOTDIGIT, and NOTUPPER)

This group of functions is similar to the "ANY" functions (such as ANYALNUM, ANYALPHA, etc.) except that the function returns the position of the first character value that is **not** a particular value (alphanumeric, character, digit, or uppercase character). Note that this is not a complete list of the "NOT" functions. See the *SAS OnlineDoc 9.1* for a complete list.

As with the "ANY" functions, there is an optional parameter that specifies where to start the search and in which direction to search.

SAS9.1 Function: NOTALNUM

Purpose: To determine the position of the first character in a string that is **not** an alphanumeric (any upper- or lowercase letter or a number). If none is found, the function returns a 0. With the use of an optional parameter, this function can begin searching at any position in the string and can also search from right to left, if desired.

Syntax: **NOTALNUM(*character-value <,start>*)**

character-value is any SAS character expression.

start is an optional parameter that specifies the position in the string to begin the search. If it is omitted, the search starts at the beginning of the string. If it is non-zero, the search begins at the position in the string of the absolute value of the number (starting from the left-most position in the string). If the start value is positive, the search goes from left to right; if the value is negative, the search goes from right to left. A negative value larger than the length of the string results in a scan from right to left, starting at the end of the string. If the value of *start* is a positive number longer than the length of the string, or if it is 0, the function returns a 0.

Examples

For these examples, `STRING = "ABC 123 ?xyz_n_"`

Function	Returns
`NOTALNUM(STRING)`	`4` (position of the 1st blank)
`NOTALNUM("Testing123")`	`0` (all alpha-numeric values)
`NOTALNUM("??$$%%")`	`1` (position of the "?")
`NOTALNUM(STRING,5)`	`8` (position of the 2nd blank)
`NOTALNUM(STRING,-6)`	`4` (position of the 1st blank)
`NOTALNUM(STRING,8)`	`9` (position of the "?")

SAS9.1 Function: NOTALPHA

Purpose: To determine the position of the first character in a string that is **not** an upper- or lowercase letter (alpha character). If none is found, the function returns a 0. With the use of an optional parameter, this function can begin searching at any position in the string and can also search from right to left, if desired.

Syntax: **NOTALPHA**(*character-value* <,*start*>)

character-value is any SAS character expression.

start is an optional parameter that specifies the position in the string to begin the search. If it is omitted, the search starts at the beginning of the string. If it is non-zero, the search begins at the position in the string of the absolute value of the number (starting from the left-most position in the string). If the start value is positive, the search goes from left to right; if the value is negative, the search goes from right to left. A negative value larger than the length of the string results in a scan from right to left, starting at the end of the string. If the value of *start* is a positive number longer than the length of the string, or if it is 0, the function returns a 0.

Examples

For these examples, `STRING = "ABC 123 ?xyz_n_"`

Function	Returns
`NOTALPHA(STRING)`	`4` (position of 1st blank)
`NOTALPHA("ABCabc")`	`0` (all alpha characters)
`NOTALPHA("??$$%%")`	`1` (position of first "?")
`NOTALPHA(STRING,5)`	`5` (position of "1")
`NOTALPHA(STRING,-10)`	`9` (start at position 10 and search left, position of "?")
`NOTALPHA(STRING,2)`	`4` (position of 1st blank)

SAS9.1 Function: NOTDIGIT

Purpose: To determine the position of the first character in a string that is **not** a digit. If none is found, the function returns a 0. With the use of an optional parameter, this function can begin searching at any position in the string and can also search from right to left, if desired.

Syntax: **NOTDIGIT**(*character-value* <,*start*>)

character-value is any SAS character expression.

start is an optional parameter that specifies the position in the string to begin the search. If it is omitted, the search starts at the beginning of the string. If it is non-zero, the search begins at the position in the string of the absolute value of the number (starting from the left-most position in the string). If the start value is positive, the search goes from left to right; if the value is negative, the search goes from right to left. A negative value larger than the length of the string results in a scan from right to left, starting at the end of the string. If the value of *start* is a positive number longer than the length of the string, or if it is 0, the function returns a 0.

Examples

For these examples, `STRING = "ABC 123 ?xyz_n_"`

Function	Returns
`NOTDIGIT(STRING)`	`1` (position of "A")
`NOTDIGIT("123456")`	`0` (all digits)
`NOTDIGIT("??$$%%")`	`1` (position of "?")
`NOTDIGIT(STRING,5)`	`8` (position of 2nd blank)
`NOTDIGIT(STRING,-6)`	`4` (position of 1st blank)
`NOTDIGIT(STRING,6)`	`8` (position of 2nd blank)

SAS9.1 Function: NOTUPPER

Purpose: To determine the position of the first character in a string that is **not** an uppercase letter. If none is found, the function returns a 0. With the use of an optional parameter, this function can begin searching at any position in the string and can also search from right to left, if desired.

Syntax: **NOTUPPER**(*character-value <,start>*)

character-value is any SAS character expression.

start is an optional parameter that specifies the position in the string to begin the search. If it is omitted, the search starts at the beginning of the string. If it is non-zero, the search begins at the position in the string of the absolute value of the number (starting from the left-most position in the

string). If the start value is positive, the search goes from left to right; if the value is negative, the search goes from right to left. A negative value larger than the length of the string results in a scan from right to left, starting at the end of the string. If the value of *start* is a positive number longer than the length of the string, or if it is 0, the function returns a 0.

Examples

For these examples, `STRING = "ABC 123 ?xyz_n_"`

Function	Returns
`NOTUPPER("ABCDabcd")`	`5` (position of "a")
`NOTUPPER("ABCDEFG")`	`0` (all uppercase characters)
`NOTUPPER(STRING)`	`4` (position of 1st blank)
`NOTUPPER("??$$%%")`	`1` (position of "?")
`NOTUPPER(STRING,5)`	`5` (position of "1")
`NOTUPPER(STRING,-6)`	`6` (position of "2")
`NOTUPPER(STRING,6)`	`6` (position of "2")

Program 1.13: Demonstrating the "NOT" character functions

```
***Primary functions: NOTALNUM, NOTALPHA, NOTDIGIT, AND NOTUPPER;

DATA NEGATIVE;
   INPUT STRING $5.;
   NOT_ALPHA_NUMERIC = NOTALNUM(STRING);
   NOT_ALPHA         = NOTALPHA(STRING);
   NOT_DIGIT         = NOTDIGIT(STRING);
   NOT_UPPER         = NOTUPPER(STRING);
DATALINES;
ABCDE
abcde
abcDE
12345
:#$%&
ABC
;
PROC PRINT DATA=NEGATIVE NOOBS;
   TITLE "Listing of Data Set NEGATIVE";
RUN;
```

Explanation

This straightforward program demonstrates each of the "NOT" character functions. As with most character functions, be careful with trailing blanks. Notice that the last observation ("ABC") contains only three characters but since STRING is read with a $5. informat, there are two trailing blanks following the letters 'ABC'. That is the reason you obtain a value of 4 for all the functions except NOTDIGIT, which returns a 1 (the first character is not a digit). A listing of the data set NEGATIVE is shown next:

Listing of Data Set NEGATIVE

STRING	NOT_ALPHA_ NUMERIC	NOT_ ALPHA	NOT_ DIGIT	NOT_ UPPER
ABCDE	0	0	1	0
abcde	0	0	1	1
abcDE	0	0	1	1
12345	0	1	0	1
:#$%&	1	1	1	1
ABC	4	4	1	4

FIND and FINDC

This pair of functions shares some similarities to the INDEX and INDEXC functions. FIND and INDEX both search a string for a given substring. FINDC and INDEXC both search for individual characters. However, both FIND and FINDC have some additional capability over their counterparts. For example, this pair of functions has the ability to declare a starting position for the search, the direction of the search, and to ignore case or trailing blanks.

SAS9.1 Function: FIND

Purpose: To locate a substring within a string. With optional arguments, you can define the starting point for the search, the direction of the search, and ignore case or trailing blanks.

Syntax: **FIND(*character-value, find-string* <,'*modifiers*'> <,*start*>)**

character-value is any SAS character expression.

find-string is a character variable or string literal that contains one or more characters that you want to search for. The function returns the first position in the *character-value* that contains the *find-string*. If the *find-string* is not found, the function returns a 0.

The following modifiers (in upper- or lowercase), placed in single or double quotation marks, may be used with FIND:

`i` ignore case.

`t` ignore trailing blanks in both the character variable and the *find-string*.

start is an optional parameter that specifies the position in the string to begin the search. If it is omitted, the search starts at the beginning of the string. If it is non-zero, the search begins at the position in the string of the absolute value of the number. If the value is positive, the search goes from left to right; if the value is negative, the search goes from right to left. A negative value larger than the length of the string results in a scan from right to left, starting at the end of the string. If the value of *start* is a positive number longer than the length of the string, or if it is 0, the function returns a 0.

Examples

For these examples `STRING1 = "Hello hello goodbye"` and `STRING2 = "hello"`

Function	Returns
`FIND(STRING1, STRING2)`	`7`
`FIND(STRING1, STRING2, 'I')`	`1`
`FIND(STRING1,"bye")`	`17`
`FIND("abcxyzabc","abc",4)`	`7`
`FIND(STRING1, STRING2, "i", -99)`	`7`

SAS9.1 Function: FINDC

Purpose: To locate a character that appears or does not appear within a string. With optional arguments, you can define the starting point for the search, the direction of the search, to ignore case or trailing blanks, or to look for characters except the ones listed.

Syntax: **FINDC(*character-value*, *find-characters* <,'*modifiers*'> <,*start*>)**

character-value is any SAS character expression.

find-characters is a list of one or more characters that you want to search for.

The function returns the first position in the *character-value* that contains one of the *find-characters*. If none of the characters are found, the function returns a 0. With an optional argument, you can have the function return the position in a character string of a character that is not in the *find-characters* list.

modifiers (in upper- or lowercase), placed in single or double quotation marks, may be used with FINDC as follows:

- i ignore case.
- t ignore trailing blanks in both the character variable and the *find-characters.*
- v count only characters that are not in the list of find characters.
- o process the modifiers and find characters only once to a specific call to the function. In subsequent calls, changes to these arguments will have no effect.

start is an optional parameter that specifies the position in the string to begin the search. If it is omitted, the search starts at the beginning of the string. If it is non-zero, the search begins at the position in the string of the absolute value of the number. If the value is positive, the search goes from

left to right; if the value is negative, the search goes from right to left. A negative value larger than the length of the string results in a scan from right to left, starting at the end of the string. If the value of *start* is a positive number longer than the length of the string, or if it is 0, the function returns a 0.

Note: You can switch the positions of *start* and *modifiers* and the function will work the same.

Examples

For these examples `STRING1 = "Apples and Books"` and `STRING2 = "abcde"`

Function	Returns
`FINDC(STRING1, STRING2)`	5
`FINDC(STRING1, STRING2, 'i')`	1
`FINDC(STRING1,"aple",'vi')`	6
`FINDC("abcxyzabc","abc",4)`	7

Program 1.14: Using the FIND and FINDC functions to search for strings and characters

```
***Primary functions: FIND and FINDC;

DATA FIND_VOWEL;
   INPUT @1 STRING $20.;
   PEAR = FIND(STRING,"Pear");
   POS_VOWEL = FINDC(STRING,"aeiou",'I');
   UPPER_VOWEL = FINDC(STRING,"aeiou");
   NOT_VOWEL = FINDC(STRING,"AEIOU",'IV');
DATALINES;
XYZABCabc
XYZ
Apple and Pear
;
PROC PRINT DATA=FIND_VOWEL NOOBS;
   TITLE "Listing of Data Set FIND_VOWEL";
RUN;
```

Explanation

The FIND function returns the position of the characters "Pear" in the variable STRING. Since the i modifier is not used, the search is case-sensitive. The first use of the FINDC function looks for any upper- or lowercase vowel in the string (because of the i modifier). The next statement, without the i modifier, locates only lowercase vowels. Finally, the v modifier in the last FINDC function reverses the search to look for the first character that is not a vowel (upper- or lowercase because of the i modifier).

Program 1.15: Demonstrating the o modifier with FINDC

```
***Primary function: FINDC;

DATA O_MODIFIER;
   INPUT STRING      $15.
         @16 LOOK_FOR $1.;
   POSITION = FINDC(STRING,LOOK_FOR,'IO');
DATALINES;
Capital A here A
Lower a here   X
Apple          B
;
PROC PRINT DATA=O_MODIFIER NOOBS HEADING=H;
   TITLE "Listing of Data Set O_MODIFIER";
RUN;
```

Explanation

In the first call to FINDC, the value of LOOK_FOR is an uppercase A. Since the o modifier was used, changing the value of LOOK_FOR in the next two observations has no effect—the function continues to look for the letter A. Note that another use of FINDC in this DATA step would not be affected by the previous use of the o modifier, even if the name of the variable (in this case POSITION) were the same. The o modifier is most likely useful in reducing processing time when looping through multiple strings, looking for the same string with the same modifiers. The listing of data set O_MODIFIER below shows that, even though the LOOK_FOR value was changed to X in the second observation and B in the third observation, the function continues to search for the letter A.

Listing of Data Set O_MODIFIER

STRING	LOOK_FOR	POSITION
Capital A here	A	2
Lower a here	X	7
Apple	B	1

INDEX, INDEXC, and INDEXW

This group of functions all search a string for a substring of one or more characters. INDEX and INDEXW are similar, the difference being that INDEXW looks for a word (defined as a string bounded by spaces or the beginning or end of the string) while INDEX simply searches for the designated substring. INDEXC searches for one or more individual characters and always searches from right to left. Note that these three functions are all case-sensitive.

Function: INDEX

Purpose: To locate the starting position of a substring in a string.

Syntax: **INDEX(*character-value, find-string*)**

character-value is any SAS character expression.

find-string is a character variable or string literal that contains the substring for which you want to search.

The function returns the first position in the *character-value* that contains the *find-string*. If the *find-string* is not found, the function returns a 0.

Examples

For these examples `STRING = "ABCDEFG"`

Function	Returns
INDEX(STRING,'C')	3 (the position of the 'C')
INDEX(STRING,'DEF')	4 (the position of the 'D')
INDEX(STRING,'X')	0 (no "X" in the string)
INDEX(STRING,'ACE')	0 (no "ACE" in the string)

Program 1.16: Converting numeric values of mixed units (e.g., kg and lbs) to a single numeric quantity

```
***Primary functions: COMPRESS, INDEX, INPUT
***Other function: ROUND;

DATA HEAVY;
   INPUT CHAR_WT $ @@;
   WEIGHT = INPUT(COMPRESS(CHAR_WT,'KG'),8.);
   IF INDEX(CHAR_WT,'K') NE 0 THEN WEIGHT = 2.22 * WEIGHT;
   WEIGHT = ROUND(WEIGHT);
   DROP CHAR_WT;
DATALINES;
60KG 155 82KG 54KG 98
;
PROC PRINT DATA=HEAVY NOOBS;
   TITLE "Listing of Data Set HEAVY";
   VAR WEIGHT;
RUN;
```

Explanation

The data lines contain numbers in kilograms, followed by the abbreviation KG or in pounds (no units used). As with most problems of this type, when you are reading a combination of numbers and characters, you usually need to first read the value as a character. Here the COMPRESS function is used to remove the letters KG from the character value. The INPUT function does its usual job of character to numeric conversion. If the INDEX function returns any value other than a 0, the letter K was found in the string and the WEIGHT value is converted from KG to pounds. Finally, the value is rounded to the nearest pound, using the ROUND function. The listing of data set HEAVY follows:

```
                          Listing of Data Set HEAVY

                          WEIGHT

                            133
                            155
                            182
                            120
                             98
```

Function: INDEXC

Purpose: To search a character string for one or more characters. The INDEXC function works in a similar manner to the INDEX function, with the difference being it can be used to search for any one in a list of character values.

Syntax: **INDEXC(*character-value*, '*char1*','*char2*','*char3*', ...)**

INDEXC(*character-value*, '*char1char2char3*. . .')

character-value is any SAS character expression.

char1, *char2*, ... are individual character values that you wish to search for in the *character-value*.

The INDEXC function returns the first occurrence of any of the *char1* , *char2*, etc., values in the string. If none of the characters is found, the function returns a 0.

Examples

For these examples `STRING = "ABCDEFG"`

Function	Returns
INDEXC(STRING,'F','C','G')	3 (position of the "C")
INDEXC(STRING, 'FCG')	3 (position of the "C")
INDEXC(STRING,'FCG')	3 (position of the "C")
INDEXC(STRING,'X','Y','Z')	0 (no "X", "Y", or "Z" in STRING)

Note: It makes no difference if you list the search characters as 'ABC' or 'A','B','C'.

Program 1.17: Searching for one of several characters in a character variable

```
***Primary function: INDEXC;

DATA CHECK;
   INPUT TAG_NUMBER $ @@;
   ***If the tag number contains an X, Y, or Z, it indicates
      an international destination, otherwise, the destination
      is domestic;
   IF INDEXC(TAG_NUMBER,'X','Y','Z') GT 0 THEN
      DESTINATION = 'INTERNATIONAL';
   ELSE DESTINATION = 'DOMESTIC';
DATALINES;
T123 TY333 1357Z UZYX 888 ABC
;
PROC PRINT DATA=CHECK NOOBS;
   TITLE "Listing of Data Set CHECK";
   ID TAG_NUMBER;
   VAR DESTINATION;
RUN;
```

Explanation

Rather than use three statements using the INDEX function, you can use the INDEXC function, which allows you to check for any one of a number of character values. Here, if an X, Y, or Z is found in the variable TAG_NUMBER, the function returns a number greater than 0 and DESTINATION will be set to INTERNATIONAL. As you can see in the listing below, this use of the INDEXC function works as advertised.

```
                        Listing of Data Set CHECK

                         TAG_
                        NUMBER    DESTINATION

                        T123      DOMESTIC
                        TY333     INTERNATIONAL
                        1357Z     INTERNATIONAL
                        UZYX      INTERNATIONAL
                        888       DOMESTIC
                        ABC       DOMESTIC
```

Program 1.18: Reading dates in a mixture of formats

```
***Primary function: INDEXC
***Other function: INPUT;

***Note: Version 9 has some enhanced date reading ability;

***Program to read mixed dates;
DATA MIXED_DATES;
   INPUT @1 DUMMY $15.;
   IF INDEXC(DUMMY,'/-:') NE 0 THEN DATE = INPUT(DUMMY,MMDDYY10.);
   ELSE DATE = INPUT(DUMMY,DATE9.);
   FORMAT DATE WORDDATE.;
   DROP DUMMY;
DATALINES;
10/21/1946
06JUN2002
5-10-1950
7:9:57
;
PROC PRINT DATA=MIXED_DATES NOOBS;
   TITLE "Listing of Data Set MIXED_DATES";
   VAR DATE;
RUN;
```

Explanation

In this somewhat trumped-up example, dates are entered either in *mm*/*dd*/*yyyy* or *ddMONyyyy* form. Also, besides a slash, dashes and colons are used. Any string that includes either a slash, dash, or colon is a date that needs the mmddyy10. informat.

Otherwise, the date9. informat is used. A list of the data set MIXED_DATES is shown below:

```
                    Listing of Data Set MIXED_DATES

                          DATE

                    October 21, 1946
                        June 6, 2002
                        May 10, 1950
                        July 9, 1957
```

Function: INDEXW

Purpose: To search a string for a word, defined as a group of letters separated on both ends by a word boundary (a space, the beginning of a string, end of the string). Note that punctuation is not considered a word boundary.

Syntax: **INDEXW(*character-value*, *find-string*)**

character-value is any SAS character expression.

find-string is the word for which you want to search.

The function returns the first position in the *character-value* that contains the *find-string*. If the *find-string* is not found, the function returns a 0.

Examples

For these examples `STRING1 = "there is a the here"` and `STRING2 = "end in the."`

Function	Result
`INDEXW(STRING1,"the")`	`12` (the word "the")
`INDEXW("ABABAB","AB")`	`0` (no word boundaries around "AB")
`INDEXW(STRING1,"er")`	`0` (not a word)
`INDEXC(STRING2,"the")`	`0` (punctuation is not a word boundary)

Program 1.19: Searching for a word using the INDEXW function

```
***Primary functions: INDEX and INDEXW;

DATA FIND_WORD;
   INPUT STRING $40.;
   POSITION_W = INDEXW(STRING,"the");
   POSITION   = INDEX(STRING,"the");
DATALINES;
there is a the in this line
ends in the
ends in the.
none here
;
PROC PRINT DATA=FIND_WORD;
   TITLE "Listing of Data Set FIND_WORD";
RUN;
```

Explanation

This program demonstrates the difference between INDEX and INDEXW. Notice in the first observation in the listing below, the INDEX function returns a 1 because the letters "the" as part of the word "there" begin the string. Since the INDEXW function needs either white space at the beginning or end of a string to delimit a word, it returns a 12, the position of the word "the" in the string. Observation 3 emphasizes the fact that a punctuation mark does not serve as a word separator. Finally, since the string "the" does not appear anywhere in the fourth observation, both functions return a 0. Here is the listing:

Listing of Data Set FIND_WORD

Obs	STRING	POSITION_W	POSITION
1	there is a the in this line	12	1
2	ends in the	9	9
3	ends in the.	0	9
4	none here	0	0

Function: VERIFY

Purpose: To check if a string contains any unwanted values.

Syntax: `VERIFY(`*`character-value, verify-string`*`)`

`character-value` is any SAS character expression.

`verify-string` is a SAS character variable or a list of character values in quotation marks.

This function returns the first position in the *`character-value`* that is **not** present in the *`verify-string`*. If the *`character-value`* does not contain any characters other than those in the *`verify-string`*, the function returns a 0. Be especially careful to think about trailing blanks when using this function. If you have an 8-byte character variable equal to 'ABC' (followed by five blanks), and if the verify string is equal to 'ABC', the VERIFY function returns a 4, the position of the first blank (which is not present in the verify string). Therefore, you may need to use the TRIM function on either the *`character-value`*, the *`verify-string`*, or both.

Examples

For these examples `STRING = "ABCXABD"` and `V = "ABCDE"`

Function	Returns
`VERIFY(STRING,V)`	`4` ("X" is not in the verify string)
`VERIFY(STRING,"ABCDEXYZ")`	`0` (no "bad" characters in STRING)
`VERIFY(STRING,"ACD")`	`2` (position of the "B")
`VERIFY("ABC  ","ABC")`	`4` (position of the 1st blank)
`VERIFY(TRIM("ABC  "),"ABC")`	`0` (no invalid characters)

Program 1.20: Using the VERIFY function to check for invalid character data values

```
***Primary function: VERIFY;

DATA VERY_FI;
   INPUT ID     $ 1-3
         ANSWER $ 5-9;
   P = VERIFY(ANSWER,'ABCDE');
   OK = P EQ 0;
DATALINES;
001 ACBED
002 ABXDE
003 12CCE
004 ABC E
;
PROC PRINT DATA=VERY_FI NOOBS;
   TITLE "listing of Data Set VERY_FI";
RUN;
```

Explanation

In this example, the only valid values for ANSWER are the uppercase letters A–E. Any time there are one or more invalid values, the result of the VERIFY function (variable P) will be a number from 1 to 5. The SAS statement that computes the value of the variable OK needs a word of explanation. First, the logical comparison `P EQ 0` returns a value of true or false, which is equivalent to a 1 or 0. This value is then assigned to the variable OK. Thus, the variable OK is set to 1 for all valid values of ANSWER and to 0 for any invalid values. This use of the VERIFY function is very handy in some data cleaning applications. A listing of data set VERI_FI is shown below:

listing of Data Set VERY_FI

ID	ANSWER	P	OK
001	ACBED	0	1
002	ABXDE	3	0
003	12CCE	1	0
004	ABC E	4	0

Functions That Extract Parts of Strings

The functions described in this section can extract parts of strings. When used on the left hand side of the equal sign, the SUBSTR function can also be used to insert characters into specific positions of an existing string.

Function: SUBSTR

Purpose: To extract part of a string. When the SUBSTR function is used on the left side of the equal sign, it can place specified characters into an existing string.

Syntax: **SUBSTR(*character-value, start <,length>*)**

character-value is any SAS character expression.

start is the starting position within the string.

length if specified, is the number of characters to include in the substring. If this argument is omitted, the SUBSTR function will return all the characters from the start position to the end of the string.

If a length has not been previously assigned, the length of the resulting variable will be the length of the *character-value*.

Examples

For these examples, let `STRING = "ABC123XYZ"`

Function	Returns
`SUBSTR(STRING,4,2)`	`"12"`
`SUBSTR(STRING,4)`	`"123XYZ"`
`SUBSTR(STRING,LENGTH(STRING))`	`"Z"` (last character in the string)

Program 1.21: Extracting portions of a character value and creating a character variable and a numeric value

```
***Primary function: SUBSTR
***Other function: INPUT;

DATA SUBSTRING;
   INPUT ID $ 1-9;
   LENGTH STATE $ 2;
   STATE = SUBSTR(ID,1,2);
   NUM = INPUT(SUBSTR(ID,7,3),3.);
DATALINES;
NYXXXX123
NJ1234567
;
PROC PRINT DATA=SUBSTRING NOOBS;
   TITLE 'Listing of Data Set SUBSTRING';
RUN;
```

Explanation

In this example, the ID contains both state and number information. The first two characters of the ID variable contain the state abbreviations and the last three characters represent numerals that you want to use to create a numeric variable. Extracting the state codes is straightforward. To obtain a numeric value from the last 3 bytes of the ID variable, it is necessary to first use the SUBSTR function to extract the three characters of interest and to then use the INPUT function to do the character to numeric conversion. A listing of data set SUBSTRING is shown next:

```
                    Listing of Data Set SUBSTRING

                       ID        STATE    NUM

                    NYXXXX123     NY      123
                    NJ1234567     NJ      567
```

Program 1.22: Extracting the last two characters from a string, regardless of the length

```
***Primary functions: LENGTH, SUBSTR;

DATA EXTRACT;
   INPUT @1 STRING $20.;
    LAST_TWO = SUBSTR(STRING,LENGTH(STRING)-1,2);
DATALINES;
ABCDE
AX12345NY
126789
;
PROC PRINT DATA=EXTRACT NOOBS;
   TITLE "Listing of Data Set EXTRACT";
    VAR STRING LAST_TWO;
RUN;
```

Explanation

This program demonstrates how you can use the LENGTH and SUBSTR functions together to extract portions of a string when the strings are of different or unknown lengths. To see how this program works, take a look at the first line of data. The LENGTH function will return a 5 and (5–1) = 4, the position of the next to the last (penultimate) character in STRING. See the listing below:

```
                        Listing of Data Set EXTRACT

                        STRING       LAST_TWO

                        ABCDE           DE
                        AX12345NY       NY
                        126789          89
```

Program 1.23: Using the SUBSTR function to "unpack" a string

```
***Primary function: SUBSTR
***Other functions: INPUT;

DATA PACK;
   INPUT STRING $ 1-5;
DATALINES;
12345
8 642
;
DATA UNPACK;
   SET PACK;
   ARRAY X[5];
   DO J = 1 TO 5;
      X[J] = INPUT(SUBSTR(STRING,J,1),1.);
   END;
   DROP J;
RUN;
PROC PRINT DATA=UNPACK NOOBS;
   TITLE "Listing of Data Set UNPACK";
RUN;
```

Explanation

There are times when you want to store a group of one-digit numbers in a compact, space-saving way. In this example, you want to store five one-digit numbers. If you stored each one as an 8-byte numeric, you would need 40 bytes of storage for each observation. By storing the five numbers as a 5-byte character string, you need only 5 bytes of storage. However, you need to use CPU time to turn the character string back into the five numbers.

The key here is to use the SUBSTR function with the starting value as the index of a DO loop. As you pick off each of the numerals, you can use the INPUT function to do the character-to-numeric conversion. Notice that the ARRAY statement in this program does not include a list of variables. When this list is omitted and the number of elements is placed in parentheses, SAS automatically uses the array name followed by the numbers from 1 to *n*, where *n* is the number in parentheses. A listing of data set UNPACK is shown below:

```
                    Listing of Data Set UNPACK

                  STRING    X1    X2    X3    X4    X5

                  12345      1     2     3     4     5
                  8 642      8     .     6     4     2
```

Function: SUBSTR (on the left-hand side of the equal sign)

As we mentioned in the description of the SUBSTR function, there is an interesting and useful way it can be used—on the left-hand side of the equal sign.

Purpose: To place one or more characters into an existing string.

Syntax: **SUBSTR(*character-value*, *start* <, *length*>) = *character-value***

character-value is any SAS character expression.

start is the starting position in a string where you want to place the new characters.

length is the number of characters to be placed in that string. If *length* is omitted, all the characters on the right-hand side of the equal sign replace the characters in *character-value*.

Examples

In these examples `EXISTING = "ABCDEFGH", NEW = "XY"`

Function	Returns
`SUBSTR(EXISTING,3,2) = NEW`	`EXISTING` is now `= "ABXYEFGH"`
`SUBSTR(EXISTING,3,1) = "*"`	`EXISTING` is now `= "AB*DEFGH"`

Program 1.24: Demonstrating the SUBSTR function on the left-hand side of the equal sign

```
***Primary function: SUBSTR
***Other function: PUT;

DATA STARS;
   INPUT SBP DBP @@;
   LENGTH SBP_CHK DBP_CHK $ 4;
   SBP_CHK = PUT(SBP,3.);
   DBP_CHK = PUT(DBP,3.);
   IF SBP GT 160 THEN SUBSTR(SBP_CHK,4,1) = '*';
   IF DBP GT 90 THEN SUBSTR(DBP_CHK,4,1) = '*';
DATALINES;
120 80 180 92 200 110
;
PROC PRINT DATA=STARS NOOBS;
   TITLE "Listing of Data Set STARS";
RUN;
```

Explanation

In this program, you want to "flag" high values of systolic and diastolic blood pressure by placing an asterisk after the value. Notice that the variables SBP_CHK and DBP_CHK are both assigned a length of 4 by the length statement. The fourth position needs to be there in case you want to place an asterisk in that position, to flag the value as abnormal. The PUT function places the numerals of the blood pressures into the first 3 bytes of the corresponding character variables. Then, if the value is above the specified level, an asterisk is placed in the fourth position of these variables.

```
                        Listing of Data Set STARS

                        SBP    DBP    SBP_CHK    DBP_CHK

                        120     80     120         80
                        180     92     180*        92*
                        200    110     200*       110*
```

SAS9.1 Function: SUBSTRN

Purpose: This function serves the same purpose as the SUBSTR function with a few added features. Unlike the SUBSTR function, the starting position and the length arguments of the SUBSTRN function can be 0 or negative without causing an error. In particular, if the length is 0, the function returns a string of 0 length. This is particularly useful when you are using regular expression functions where the length parameter may be 0 when a pattern is not found (with the PRXSUBSTR function, for example). You can use the SUBSTRN function in any application where you would use the SUBSTR function. The effect of 0 or negative parameters is discussed in the description of the arguments below.

Syntax: **SUBSTRN(*character-value*, *start* <, *length*>)**

character-value is any SAS character variable.

start is the starting position in the string. If this value is non-positive, the function returns a substring starting from the first character in *character-value* (the length of the substring will be computed by counting, starting from the value of *start*). See the program below.

length is the number of characters in the substring. If this value is non-positive (in particular, 0), the function returns a string of length 0. If this argument is omitted, the SUBSTRN function will return all the characters from the start position to the end of the string.

If a length has not been previously assigned, the length of the resulting variable will be the length of the *character-value*.

Examples

For these examples, `STRING = "ABCDE"`

Function	Returns
`SUBSTRN(STRING,2,3)`	`"BCD"`
`SUBSTRN(STRING,-1,4)`	`"AB"`
`SUBSTRN(STRING,4,5)`	`"DE"`
`SUBSTRN(STRING,3,0)`	`string of zero length`

Program 1.25: Demonstrating the unique features of the SUBSTRN function

```
***Primary function: SUBSTRN;

DATA HOAGIE;
   STRING = 'ABCDEFGHIJ';
   LENGTH RESULT $5.;
   RESULT = SUBSTRN(STRING,2,5);
   SUB1 = SUBSTRN(STRING,-1,4);
   SUB2 = SUBSTRN(STRING,3,0);
   SUB3 = SUBSTRN(STRING,7,5);
   SUB4 = SUBSTRN(STRING,0,2);
   FILE PRINT;
   TITLE "Demonstrating the SUBSTRN Function";
   PUT "Original String ="      @25 STRING  /
       "SUBSTRN(STRING,2,5) ="  @25 RESULT  /
       "SUBSTRN(STRING,-1,4) =" @25 SUB1    /
       "SUBSTRN(STRING,3,0) ="  @25 SUB2    /
       "SUBSTRN(STRING,7,5) ="  @25 SUB3    /
       "SUBSTRN(STRING,0,2) ="  @25 SUB4;
RUN;
```

Explanation

In data set HOAGIE (sub-strings, get it?) the storage lengths of the variables SUB1–SUB4 are all equal to the length of STRING (which is 10). Since a LENGTH statement was used to define the length of RESULT, it has a length of 5.

Examine the results below and the brief explanation that follows the results.

```
                    Demonstrating the SUBSTRN Function

                    Original String =       ABCDEFGHIJ
                    SUBSTRN(STRING,2,5) =   BCDEF
                    SUBSTRN(STRING,-1,4) =  AB
                    SUBSTRN(STRING,3,0) =
                    SUBSTRN(STRING,7,5) =   GHIJ
                    SUBSTRN(STRING,0,2) =   A
```

The first function call (RESULT) gives the same result as the SUBSTR function. The resulting substring starts at the second position in STRING (B) and has a length of 5. All the remaining SUBSTRN functions would have resulted in an error if the SUBSTR function had been used instead.

The starting position of –1 and a length of 4 results in the characters "AB." To figure this out, realize that you start counting from –1 (–1, 0, 1, 2) and the result is the first two characters in STRING.

When the LENGTH is 0, the result is a string of length 0.

When you start at position 7 and have a length of 5, you go past the end of the string, so that the result is truncated and the result is "GHIJ."

Finally, when the starting position is 0 and the length is 2, you get the first character in string, "A."

Functions That Join Two or More Strings Together

There are three call routines and four functions that concatenate character strings. Although you can use the || concatenation operator in combination with the STRIP, TRIM, or LEFT functions, these routines and functions make it much easier to put strings together and, if you wish, to place one or more separator characters between the strings. The three call routines are discussed first, followed by the four concatenation functions.

Call Routines

These three call routines, new with Version 9, concatenate two or more strings. Note that there are four concatenation functions as well (CAT, CATS, CATT, and CATX). The differences among these routines involve the handling of leading and/or trailing blanks as well as spacing between the concatenated strings. The traditional concatenation operator (||) is still useful, but it sometimes takes extra work to strip leading and trailing blanks (LEFT and TRIM functions, or the new STRIP function) before performing the concatenation operation. These call routines are a convenience, and you will probably want to use them in place of the older form of concatenation. There are corresponding concatenation functions described in the next section.

One advantage of using the call routines over their corresponding functions is improved performance. For example, CALL CATS(R, X, Y, Z) is faster than R = CATS(R, X, Y, Z).

We will describe all three call routines and follow with one program demonstrating all three.

SAS9.1 Function: CALL CATS

Purpose: To concatenate two or more strings, removing both leading and trailing blanks before the concatenation takes place. To help you remember that this call routine is the one that strips the leading and trailing blanks before concatenation, think of the S at the end of CATS as "strip blanks."
Note: To call our three cats, I usually just whistle loudly.

Syntax: `CALL CATS(result, string-1 <,string-n>)`

result is the concatenated string. It can be a new variable or, if it is an existing variable, the other strings will be added to it. **Be sure that the length of result is long enough to hold the concatenated results.** If not, the resulting string will be truncated, and you will see an error message in the log.

string-1 and *string-n* are the character strings to be concatenated. Leading and trailing blanks will be stripped prior to the concatenation.

Example

For these examples
`A = "Bilbo"` (no blanks)
`B = "   Frodo"` (leading blanks)
`C = "Hobbit   "` (trailing blanks)
`D = "   Gandalf   "` (leading and trailing blanks)

Function	Returns
`CALL CATS(RESULT, A, B)`	`"BilboFrodo"`
`CALL CATS(RESULT, B, C, D)`	`"FrodoHobbitGandalf"`
`CALL CATS(RESULT, "Hello", D)`	`"HelloGandalf"`

SAS9.1 Function: CALL CATT

Purpose: To concatenate two or more strings, removing only trailing blanks before the concatenation takes place. To help you remember this, think of the T at the end of CATT as "trailing blanks" or "trim blanks."

Syntax: **CALL CATT(*result, string-1 <,string-n>*)**

result is the concatenated string. It can be a new variable or, if it is an existing variable, the other strings will be added to it. **Be sure that the length of result is long enough to hold the concatenated results.** If not, the program will terminate and you will see an error message in the log.

string-1 and *string-n* are the character strings to be concatenated. Trailing blanks will be stripped prior to the concatenation.

Example

For these examples
`A = "Bilbo"` (no blanks)
`B = "   Frodo"` (leading blanks)
`C = "Hobbit   "` (trailing blanks)
`D = "   Gandalf   "` (leading and trailing blanks)

Function	Returns
`CALL CATT(RESULT, A, B)`	`"Bilbo   Frodo"`
`CALL CATT(RESULT, B, C, D)`	`"   FrodoHobbit   Gandalf"`
`CALL CATT(RESULT, "Hello", D)`	`"Hello   Gandalf"`

SAS9.1 Function: CALL CATX

Purpose: To concatenate two or more strings, removing both leading and trailing blanks before the concatenation takes place, and place a single space, or one or more characters of your choice, between each of the strings. To help you remember this, think of the X at the end of CATX as "add eXtra blank."

Syntax: **CALL CATX(*separator, result, string-1 <,string-n>*)**

separator is one or more characters, placed in single or double quotation marks, that you want to use to separate the strings

result is the concatenated string. It can be a new variable or, if it is an existing variable, the other strings will be added to it. **Be sure that the length of result is long enough to hold the concatenated results.** If not, the program will terminate and you will see an error message in the log.

String-1 and *string-n* are the character strings to be concatenated. Leading and trailing blanks will be stripped prior to the concatenation and a single blank will be placed between the strings.

Example

For these examples
`A = "Bilbo"` (no blanks)
`B = "   Frodo"` (leading blanks)
`C = "Hobbit   "` (trailing blanks)
`D = "   Gandalf   "` (leading and trailing blanks)

Function	Returns
`CALL CATX(" ", RESULT, A, B)`	`"Bilbo Frodo"`
`CALL CATX(",", RESULT, B, C, D)`	`"Frodo,Hobbit,Gandalf"`
`CALL CATX(":", RESULT, "Hello", D)`	`"Hello:Gandalf"`
`CALL CATX(", ", RESULT, "Hello", D)`	`"Hello, Gandalf"`
`CALL CATX("***", RESULT, A, B)`	`"Bilbo***Frodo"`

Program 1.26: Demonstrating the three concatenation call routines

```
***Primary functions: CALL CATS, CALL CATT, CALL CATX;

DATA CALL_CAT;
   STRING1 = "ABC";       * No spaces;
   STRING2 = "DEF   ";    * Three trailing spaces;
   STRING3 = "   GHI";    * Three leading spaces;
   STRING4 = "   JKL   "; * Three leading and trailing spaces;
   LENGTH RESULT1 - RESULT4 $ 20;
   CALL CATS(RESULT1, STRING2, STRING4);
```

```
   CALL CATT(RESULT2, STRING2, STRING1);
   CALL CATX(" ", RESULT3 ,STRING1,STRING3);
   CALL CATX(",", RESULT4,STRING3,STRING4);
RUN;
PROC PRINT DATA=CALL_CAT NOOBS;
   TITLE "Listing of Data Set CALL_CAT";
RUN;
```

Explanation

The three concatenation call routines each perform concatenation operations. The CATS call routine strips leading and trailing blanks; the CATT call routine removes trailing blanks before performing the concatenation; the CATX call routine is similar to the CATS call routine except that it inserts a separator character (specified as the first argument) between each of the concatenated strings.

According to SAS documentation, the call routines are more efficient to use than the concatenation operator combined with the TRIM and LEFT functions. For example:

```
RESULT5 = TRIM(STRING2) || " " || TRIM(LEFT(STRING3));
```

RESULT5 would be: "DEF GHI"

A listing of data set CALL_CAT is shown below:

Listing of Data Set CALL_CAT

STRING1	STRING2	STRING3	STRING4	RESULT1	RESULT2	RESULT3	RESULT4
ABC	DEF	GHI	JKL	DEFJKL	DEFABC	ABC GHI	GHI,JKL

The "CAT" Functions (CAT, CATS, CATT, and CATX)

These four concatenation functions are very similar to the concatenation call routines described above. However, since they are functions and not call routines, you need to name the new character variable to be created on the left-hand side of the equal sign and the function, along with its arguments, on the right-hand side of the equal sign. As with the concatenation call routines, we will describe the four functions together and then use a single program to demonstrate them.

SAS9.1 Function: CAT

Purpose: To concatenate (join) two or more character strings, leaving leading and/or trailing blanks unchanged. This function accomplishes the same task as the concatenation operator (||).

Syntax: **CAT(*string-1, string-2 <,string-n>*)**

string-1, string-2 <,string-n> are the character strings to be concatenated. These arguments can also be written as: `CAT(OF C1-C5)` where C1 to C5 are character variables.

Note: It is **very important to set the length of the resulting character string**, using a LENGTH statement (or other method), before using any of the concatenation functions. Otherwise, the length of the resulting string will default to 200.

Example

For these examples
`A = "Bilbo"` (no blanks)
`B = "   Frodo"` (leading blanks)
`C = "Hobbit   "` (trailing blanks)
`D = "   Gandalf   "` (leading and trailing blanks)
`C1-C5` are five character variables, with the values of `'A', 'B', 'C', 'D',` and `'E'` respectively.

Function	Returns
`CAT(A, B)`	`"Bilbo   Frodo"`
`CAT(B, C, D)`	`"   FrodoHobbit      Gandalf   "`
`CAT("Hello", D)`	`"Hello   Gandalf   "`
`CAT(OF C1-C5)`	`"ABCDE"`

SAS9.1 Function: CATS

Purpose: To concatenate (join) two or more character strings, stripping both leading and trailing blanks.

Syntax: **CATS(*string-1, string-2 <,string-n>*)**

string-1, string-2, and *string-n* are the character strings to be concatenated. These arguments can also be written as: `CATS(OF C1-C5)` where C1 to C5 are character variables.

Note: It is **very important to set the length of the resulting character string**, using a LENGTH statement or other method, before calling any of the concatenation functions. Otherwise, the length of the resulting string will default to 200.

Example

For these examples
`A = "Bilbo"` (no blanks)
`B = "   Frodo"` (leading blanks)
`C = "Hobbit   "` (trailing blanks)
`D = "   Gandalf   "` (leading and trailing blanks)
`C1-C5` are five character variables, with the values of `'A', 'B', 'C', 'D',` and `'E'` respectively.

Function	Returns
`CATS(A, B)`	`"BilboFrodo"`
`CATS(B, C, D)`	`"FrodoHobbitGandalf"`
`CATS("Hello", D)`	`"HelloGandalf"`
`CATS(OF C1-C5)`	`"ABCDE"`

SAS9.1 Function: CATT

Purpose: To concatenate (join) two or more character strings, stripping only trailing blanks.

Syntax: **CATT(*string-1, string-2 <,string-n>*)**

string1, string-2, and *string-n* are the character strings to be concatenated. These arguments can also be written as: `CATT(OF C1-C5)` where C1 to C5 are character variables.

Note: It is **very important to set the length of the resulting character string**, using a LENGTH statement or other method, before calling any of the concatenation functions. Otherwise, the length of the resulting string will default to 200.

Example:

For these examples
`A = "Bilbo"` (no blanks)
`B = "   Frodo"` (leading blanks)
`C = "Hobbit   "` (trailing blanks)
`D = "   Gandalf   "` (leading and trailing blanks)
`C1-C5` are five character variables, with the values of `'A', 'B', 'C', 'D',` and `'E'` respectively.

Function	Returns
`CATT(A, B)`	`"Bilbo   Frodo"`
`CATT(B, C, D)`	`"   FrodoHobbit   Gandalf"`
`CATT("Hello", D)`	`"Hello   Gandalf"`
`CATT(OF C1-C5)`	`"ABCDE"`

SAS9.1 Function: CATX

Purpose: To concatenate (join) two or more character strings, stripping both leading and trailing blanks and inserting one or more separator characters between the strings.

Syntax: **CATX(*separator, string-1, string-2 <,string-n>*)**

separator is one or more characters, placed in single or double quotation marks, to be used as separators between the concatenated strings.

string-1, string-2,string-n are the character strings to be concatenated. These arguments can also be written as: `CATX(" ",OF C1-C5)`, where C1 to C5 are character variables.

Note: It is **very important to set the length of the resulting character string** using a LENGTH statement (or other method), before calling any of the concatenation functions. Otherwise, the length of the resulting string will default to 200.

Example

For these examples
`A = "Bilbo"` (no blanks)
`B = "   Frodo"` (leading blanks)
`C = "Hobbit   "` (trailing blanks)
`D = "   Gandalf   "` (leading and trailing blanks)
`C1-C5` are five character variables, with the values of `'A', 'B', 'C', 'D',` and `'E'` respectively.

Function	Returns
`CATX(" ", A, B)`	`"Bilbo Frodo"`
`CATX(":"B, C, D)`	`"Frodo:Hobbit:Gandalf"`
`CATX("***", "Hello", D)`	`"Hello***Gandalf"`
`CATX("," ,OF C1-C5)`	`"A,B,C,D,E"`

Program 1.27: Demonstrating the four concatenation functions

```
***Primary functions: CAT, CATS, CATT, CATX;

DATA CAT_FUNCTIONS;
   STRING1 = "ABC";       * No spaces;
   STRING2 = "DEF   ";    * Three trailing spaces;
   STRING3 = "   GHI";    * Three leading spaces;
   STRING4 = "   JKL   "; * Three leading and trailing spaces;
   LENGTH JOIN1 - JOIN5 $ 20;
   JOIN1 = CAT(STRING2, STRING3);
   JOIN2 = CATS(STRING2, STRING4);
   JOIN3 = CATT(STRING2, STRING1);
   JOIN4 = CATX(" ",STRING1,STRING3);
   JOIN5 = CATX(",",STRING3,STRING4);
RUN;
PROC PRINT DATA=CAT_FUNCTIONS NOOBS;
   TITLE "Listing of Data Set CAT_FUNCTIONS";
RUN;
```

Explanation

Notice that each of the STRING variables differs with respect to leading and trailing blanks. The CAT function is identical to the || operator. The CATS function removes both leading and trailing blanks and is equivalent to `TRIM(LEFT(STRING2)) || TRIM(LEFT(STRING4))`. The CATT function, trims only trailing blanks. The last two statements use the CATX function, which removes leading and trailing blanks. It is just like the CATS function but adds one or more separator characters (specified as the first argument), between each of the strings to be joined. Inspection of the listing below will help make all this clear:

```
                Listing of Data Set CAT_FUNCTIONS

          STRING1    STRING2    STRING3    STRING4       JOIN1

            ABC        DEF        GHI        JKL      DEF      GHI

          JOIN2     JOIN3      JOIN4      JOIN5

          DEFJKL    DEFABC    ABC GHI    GHI,JKL
```

Functions That Remove Blanks from Strings

There are times when you want to remove blanks from the beginning or end of a character string. The two functions LEFT and RIGHT merely shift the characters to the beginning or the end of the string, respectively. The TRIM, TRIMN, and STRIP functions are useful when you want concatenate strings (although the new concatenation functions will do this for you).

LEFT and RIGHT

These two functions left- or right-align text. Remember that the length of a character variable will not change when you use these two functions. If there are leading blanks, the LEFT function will shift the first non-blank character to the first position and move the extra blanks to the end; if there are trailing blanks, the RIGHT function will shift the non-blank text to the right and move the extra blanks to the left.

Function: LEFT

Purpose: To left-align text values. A subtle but important point: LEFT doesn't "remove" the leading blanks; it moves them to the end of the string. Thus, it doesn't change the storage length of the variable, even when you assign the result of LEFT to a new variable. The LEFT function is particularly useful if values were read with the $CHAR informat, which preserves leading blanks. Note that the STRIP function removes both leading and trailing blanks from a string.

Syntax: **LEFT(*character-value*)**

character-value is any SAS character expression.

Example

In these examples `STRING = "  ABC"`

Function	Returns
`LEFT(STRING)`	`"ABC  "`
`LEFT("  123  ")`	`"123    "`

Program 1.28: Left-aligning text values from variables read with the $CHAR informat

```
***Primary function: LEFT;

DATA LEAD_ON;
   INPUT STRING $CHAR15.;
   LEFT_STRING = LEFT(STRING);
DATALINES;
ABC
   XYZ
  Ron Cody
;
PROC PRINT DATA=LEAD_ON NOOBS;
   TITLE "Listing of Data Set LEAD_ON";
   FORMAT STRING LEFT_STRING $QUOTE17.;
RUN;
```

Explanation

If you want to work with character values, you will usually want to remove any leading blanks first. The $CHARw. informat differs from the $w. informat. $CHARw. maintains leading blanks; $w. left-aligns the text. Programs involving character variables sometimes fail to work properly because careful attention was not paid to either leading or trailing blanks.

Notice the use of the $QUOTE format in the PRINT procedure. This format adds double quotation marks around the character value. This is especially useful in debugging programs involving character variables since it allows you to easily identify leading blanks in a character value.

The listing of data set LEAD_ON is shown below. Notice that the original variable STRING contains leading blanks. The length of LEFT_STRING is also 15.

```
Listing of Data Set LEAD_ON

STRING             LEFT_STRING

"ABC"              "ABC"
"   XYZ"           "XYZ"
"  Ron Cody"       "Ron Cody"
```

Function: RIGHT

Purpose: To right-align a text string. Note that if the length of a character variable has previously been defined and it contains trailing blanks, the RIGHT function will move the characters to the end of the string and add the blanks to the beginning so that the final length of the variable remains the same.

Syntax: `right(`*`character-value`*`)`

`character-value` is any SAS character expression.

Example

In these examples `STRING = "ABC  "`

Function	Returns
`RIGHT(STRING)`	`"  ABC"`
`RIGHT("  123  ")`	`"    123"`

Program 1.29: Right-aligning text values

```
***Primary function: RIGHT;

DATA RIGHT_ON;
   INPUT STRING $CHAR10.;
   RIGHT_STRING = RIGHT(STRING);
DATALINES;
   ABC
   123 456
Ron Cody
;
PROC PRINT DATA=RIGHT_ON NOOBS;
   TITLE "Listing of Data Set RIGHT_ON";
   FORMAT STRING RIGNT_STRING $QUOTE12.;
RUN;
```

Explanation

Data lines one and two both contain three leading blanks; lines one and three contain trailing blanks.

Notice the use of the $QUOTE format in the PRINT procedure. This format adds double quotation marks around the character value. This is especially useful in debugging programs involving character variables since it allows you to easily identify leading blanks in a character value.

Notice in the listing below, that the values are right-aligned and that blanks are moved to the beginning of the string:

```
                    Listing of Data Set RIGHT_ON

                    STRING          RIGHT_STRING

                    "   ABC"        "       ABC"
                    "   123 456"    "   123 456"
                    "Ron Cody"      "  Ron Cody"
```

TRIM, TRIMN, and STRIP

This group of functions trims trailing blanks (TRIM and TRIMN) and both leading and trailing blanks (STRIP).

The two functions TRIM and TRIMN are similar: they both remove trailing blanks from a string. The functions work identically except when the argument contains only blanks. In that case, TRIM returns a single blank (length of 1) and TRIMN returns a null string with a length of 0. The STRIP function removes both leading and trailing blanks.

Function: TRIM

Purpose: To remove trailing blanks from a character value. This is especially useful when you want to concatenate several strings together and each string may contain trailing blanks.

Syntax: **TRIM(*character-value*)**

character-value is any SAS character expression.

Important note: The length of the variable returned by the TRIM function will be the same length as the argument, unless the length of this variable has been previously defined. If the result of the TRIM function is assigned to a variable with a length longer than the trimmed argument, the resulting variable will be padded with blanks.

Examples

For these examples, `STRING1 = "ABC  "` and `STRING2 = "  XYZ"`

Function	Returns
`TRIM(STRING1)`	`"ABC"`
`TRIM(STRING2)`	`"  XYZ"`
`TRIM("A  B  C  ")`	`"A  B  C"`
`TRIM("A  ") || TRIM("B  ")`	`"AB"`
`TRIM("  ")`	`" "` (length = 1)

Program 1.30: Creating a program to concatenate first, middle, and last names into a single variable

```
***Primary function: TRIM;

DATA PUT_TOGETHER;
   LENGTH NAME $ 45;
   INFORMAT NAME1-NAME3 $15.;
   INFILE DATALINES MISSOVER;
   INPUT NAME1 NAME2 NAME3;
   NAME = TRIM(NAME1) || ' ' || TRIM(NAME2) || ' ' || TRIM(NAME3);
   WITHOUT = NAME1 || NAME2 || NAME3;
   KEEP NAME WITHOUT;
DATALINES;
Ronald Cody
Julia     Child
Henry    Ford
Lee Harvey Oswald
;
PROC PRINT DATA=PUT_TOGETHER NOOBS;
   TITLE "Listing Of Data Set PUT_TOGETHER";
RUN;
```

Explanation

Note that this program would be much simpler using the concatenation functions or call routines. However, this method was used to demonstrate the TRIM function.

This program reads in three names, each up to 15 characters in length. Note the use of the INFILE option MISSOVER. This options sets the value of NAME3 to missing when there are only two names.

To put the names together, you use the concatenate operator (||). The TRIM function is used to trim trailing blanks from each of the words (which are all 15 bytes in length), before putting them together. Without the TRIM function, there are extra spaces between each of the names (see the variable WITHOUT). The listing below demonstrates that the program works as desired.

```
                Listing Of Data Set PUT_TOGETHER

NAME                 WITHOUT

Ronald Cody          Ronald         Cody
Julia Child          Julia          Child
Henry Ford           Henry          Ford
Lee Harvey Oswald    Lee            Harvey         Oswald
```

Function: TRIMN

Purpose: To remove trailing blanks from a character value. This is especially useful when you want to concatenate several strings together and each string may contain trailing blanks. The difference between TRIM and TRIMN is that the TRIM function returns a single blank for a blank string while TRIMN returns a null string (zero blanks).

Syntax: `TRIMN(`*`character-value`*`)`

`character-value` is any SAS character expression.

Important note: The length of the variable returned by the TRIMN function will be the same length as the argument, unless the length of this variable has been previously defined. If the result of the TRIMN function is assigned to a variable with a length longer than the trimmed argument, the resulting variable will be padded with blanks.

Examples

For these examples, `STRING1 = "ABC  "` and `STRING2 = "  XYZ"`

Function	Returns
`TRIMN(STRING1)`	`"ABC"`
`TRIMN(STRING2)`	`"  XYZ"`
`TRIMN("A  B  C  ")`	`"A  B  C"`
`TRIMN("A  ") || TRIM("B  ")`	`"AB"`
`TRIMN("  ")`	`""` (length = 0)

Program 1.31: Demonstrating the difference between the TRIM and TRIMN functions

```
***Primary functions: TRIM, TRIMN, and LENGTHC
***Other function: COMPRESS;

DATA ALL_THE_TRIMMINGS;
   A = "AAA";
   B = "BBB";
   LENGTH_AB = LENGTHC(A || B);
   LENGTH_AB_TRIM = LENGTHC(TRIM(A) || TRIM(B));
   LENGTH_AB_TRIMN = LENGTHC(TRIMN(A) || TRIMN(B));
   LENGTH_NULL = LENGTHC(COMPRESS(A,"A") || COMPRESS(B, "B"));
   LENGTH_NULL_TRIM = LENGTHC(TRIM(COMPRESS(A,"A")) ||
                      TRIM(COMPRESS(B,"B")));
   LENGTH_NULL_TRIMN = LENGTHC(TRIMN(COMPRESS(A,"A")) ||
                       TRIMN(COMPRESS(B,"B")));
   PUT A= B= /
       LENGTH_AB= LENGTH_AB_TRIM= LENGTH_AB_TRIMN= /
       LENGTH_NULL= LENGTH_NULL_TRIM= LENGTH_NULL_TRIMN=;
RUN;
```

Explanation

First, remember that the LENGTHC function returns the length of its argument, including trailing blanks. As the listing from the SAS log (below) shows, the two functions TRIM and TRIMN yield identical results when there are no null strings involved. When you compress an 'A' from the variable A, or 'B' from variable B, the result is null. Notice that when you trim these compressed values and concatenate the results, the length is 2 (1 + 1); when you use the TRIMN function, the length is 0. Here are the lines written to the SAS log:

```
A=AAA B=BBB
LENGTH_AB=6 LENGTH_AB_TRIM=6 LENGTH_AB_TRIMN=6
LENGTH_NULL=0 LENGTH_NULL_TRIM=2 LENGTH_NULL_TRIMN=0
```

SAS9.1 Function: STRIP

Purpose: To strip leading and trailing blanks from character variables or strings. `STRIP(CHAR)` is equivalent to `TRIMN(LEFT(CHAR))`, but more convenient.

Syntax: **`STRIP(`*`character-value`*`)`**

`character-value` is any SAS character expression.

If the STRIP function is used to create a new variable, the length of that new variable will be equal to the length of the argument of the STRIP function. If leading or trailing blanks were trimmed, trailing blanks will be added to the result to pad out the length as necessary. The STRIP function is useful when using the concatenation operator. However, note that there are several new concatenation functions and call routines that also perform trimming before concatenation.

Examples

For these examples, let `STRING = "   abc   "`

Function	Returns
STRIP(STRING)	"abc" (if result was previously assigned a length of three, otherwise trailing blanks would be added)
STRIP(" LEADING AND TRAILING ")	"LEADING AND TRAILING"

Program 1.32: Using the STRIP function to strip both leading and trailing blanks from a string

```
***Primary function: STRIP;

DATA _NULL_;
   ONE = "   ONE   "; ***Note: three leading and trailing blanks;
   TWO = "   TWO   "; ***Note: three leading and trailing blanks;
   CAT_NO_STRIP = ":" || ONE || "-" || TWO || ":";
   CAT_STRIP    = ":" || STRIP(ONE) || "-" || STRIP(TWO) || ":";
   PUT ONE= TWO= / CAT_NO_STRIP= / CAT_STRIP=;
RUN;
```

Explanation

Without the STRIP function, the leading and trailing blanks are maintained in the concatenated string. The STRIP function, as advertised, removed the leading and trailing blanks. The following lines were written to the SAS log:

```
ONE=ONE TWO=TWO
CAT_NO_STRIP=:   ONE   -   TWO   :
CAT_STRIP=:ONE-TWO:
```

Functions That Compare Strings (Exact and "Fuzzy" Comparisons)

Functions in this section allow you to compare strings that are exactly alike (similar except for case) or close (not exact matches). Programmers find this latter group of functions useful in matching names that may be spelled differently in separate files.

SAS9.1 Function: COMPARE

Purpose: To compare two character strings. When used with one or more modifiers, this function can ignore case, remove leading blanks, truncate the longer string to the length of the shorter string, and strip quotation marks from SAS n-literals. While all of these tasks can be accomplished with a variety of SAS functions, use of the COMPARE function can simplify character comparisons.

Syntax: **COMPARE(*string-1, string-2* <,'*modifiers*'>)**

string-1 is any SAS character expression.

string-2 is any SAS character expression.

modifiers are one or more modifiers, placed in single or double quotation marks as follows:

i or I	ignore case.
l or L	remove leading blanks.
n or N	remove quotation marks from any argument that is an n-literal and ignore case. An n-literal is a string in quotation marks, followed by an 'n', useful for non-valid SAS names.
: (colon)	truncate the longer string to the length of the shorter string.

Note that the default is to pad the shorter string with blanks before a comparison; this is similar to the =: comparison operator.

Note that the order of the modifiers is important. See the examples below.

The function returns a 0 when the two strings match (after any modifiers are applied). If the two strings differ, a non-zero result is returned. The returned value is negative if *string-1* comes before *string-2* in a sort sequence, positive otherwise. The magnitude of this value is the position of the first difference in the two strings.

Examples

For these examples, `string1 = "AbC", string2 = "   ABC", string3 = "  'ABC'n", string4 = "ABCXYZ"`

Function	Returns
`COMPARE(string1,string4)`	`2` ("B" comes before "b")
`COMPARE(string4,string1)`	`-2`
`COMPARE(string1,string2,'i')`	`1`
`COMPARE(string1,string4,':I')`	`0`
`COMPARE(string1,string3,'nl')`	`4`
`COMPARE(string1,string3,'ln')`	`1`

Program 1.33: Comparing two strings using the COMPARE function

```
***Primary function: COMPARE
***Other function: UPCASE;

DATA COMPARE;
   INPUT @1 STRING1 $CHAR3.
         @5 STRING2 $CHAR10.;
   IF UPCASE(STRING1) = UPCASE(STRING2) THEN EQUAL = 'YES';
   ELSE EQUAL = 'NO';
   IF UPCASE(STRING1) =: UPCASE(STRING2) THEN COLON = 'YES';
   ELSE COLON = 'NO';
   COMPARE = COMPARE(STRING1,STRING2);
   COMPARE_IL = COMPARE(STRING1,STRING2,'IL');
   COMPARE_IL_COLON = COMPARE(STRING1,STRING2,'IL:');
DATALINES;
```

```
Abc    ABC
abc ABCDEFGH
123 311
;

PROC PRINT DATA=COMPARE NOOBS;
   TITLE "Listing of Data Set COMPARE";
RUN;
```

Explanation

The first two variables, EQUAL and COLON, use the UPCASE function to convert all the characters to uppercase before the comparison is made. The colon modifier following the equal sign (the variable COLON) is an instruction to truncate the longer variable to the length of the shorter variable before a comparison is made. Be careful here. If the variable STRING1 had been read with a $CHAR10. informat, the =: comparison operator would not have produced a match. The length truncation is done on the storage length of the variable, which may include trailing blanks.

The three COMPARE functions demonstrate the coding efficiency of using this function with its many modifiers. Use of this function without modifiers gives you no advantage over a simple test of equality using an equal sign. Using the IL and colon modifiers allows you to compare the two strings, ignoring case, removing leading blanks, and truncating the two strings to a length of 3 (the length of STRING1). Note the value of COMPARE_IL_COLON in the third observation is –1 since "1" comes before "3" in the ASCII collating sequence. The output from PROC PRINT is shown below:

Listing of Data Set COMPARE

STRING1	STRING2	EQUAL	COLON	COMPARE	COMPARE_IL	COMPARE_IL_COLON
Abc	ABC	NO	NO	1	0	0
abc	ABCDEFGH	NO	YES	1	-4	0
123	311	NO	NO	-1	-1	-1

CALL COMPCOST, COMPGED, and COMPLEV

The two functions, COMPGED and COMPLEV, are both used to determine the similarity between two strings. The COMPCOST call routine allows you to customize the scoring system when you are using the COMPGED function.

COMPGED computes a quantity called **generalized edit distance**, which is useful in matching names that are not spelled exactly the same. The larger the value, the more dissimilar the two strings. COMPLEV performs a similar function but uses a method called the **Levenshtein edit distance**. It is more efficient than the generalized edit distance, but may not be as useful in name matching. See the SPEDIS function for a discussion of fuzzy merging and for detailed programming examples.

SAS9.1 Function: CALL COMPCOST

Purpose: To determine the similarity between two strings, using a method called the generalized edit distance. The cost is computed based on the difference between the two strings. For example, replacing one letter with another is assigned one cost and reversing two letters can be assigned another cost. Since there is a default cost associated with every operation used by COMPGED, you can use that function without using COMPCOST at all. You need to call this function only once in a DATA step. Since this is a very advanced and complicated routine, only a few examples of its use will be explained. See the *SAS OnlineDoc 9.1* for a complete list of operations and costs.

Syntax: **CALL COMPCOST('*operation-1*', *cost-1* <,'*operation-2*', *cost-2* ...>)**

operation is a keyword, placed in quotation marks. A few keywords are listed here for explanation purposes, but see the *SAS OnlineDoc 9.1* documentation for a complete list of operations:

Partial List of Operations
DELETE=
REPLACE=
SWAP=
TRUNCATE=

cost is a value associated with the operation. Valid values for cost range from –32,767 to +32,767.

Note: The wording of arguments in this book might differ from the wording of arguments in the *SAS OnlineDoc 9.1*.

Examples

```
CALL COMPCOST('REPLACE=', 100, 'SWAP=', 200);
CALL COMPCOST('SWAP=', 150);
```

Note: Operation can be upper- or lowercase

To see how CALL COMPCOST and COMPGED are used together, see Program 1.36.

SAS9.1 Function: COMPGED

Purpose: To compute the similarity between two strings, using a method called the generalized edit distance. See SPEDIS for a discussion of the possible uses of this function.

This function can be used in conjunction with CALL COMPCOST if you want to alter the default costs for each type of spelling error.

Syntax: **COMPGED(*string-1, string-2* <,*maxcost*> <,'*modifiers*'>)**

string-1 is any SAS character expression.
string-2 is any SAS character expression.

maxcost, if specified, is the maximum cost that will be returned by the COMPLEV function. If the cost computation results in a value larger than *maxcost*, the value of *maxcost* will be returned.

modifiers placed in single or double quotation marks as follows:

`i` or `I`	ignore case.
`l` or `L`	remove leading blanks.
`n` or `N`	remove quotation marks from any argument that is an n-literal and ignore case. An n-literal is a string in quotation marks, followed by an 'n', useful for non-valid SAS names.
: (colon)	truncate the longer string to the length of the shorter string.

Note: If multiple modifiers are used, the order of the modifiers is important. They are applied in the same order as they appear.

Examples

String1	String2	Function	Returns
SAME	SAME	COMPGED(STRING1, STRING2)	0
case	CASE	COMPGED(STRING1, STRING2)	500
case	CASE	COMPGED(STRING1,STRING2,'I')	0
case	CASE	COMPGED(STRING1, STRING2, 999, 'I')	0
Ron	Run	COMPGED(STRING1, STRING2)	100

Program 1.34: Using the COMPGED function with a SAS n-literal

```
***Primary function: COMPGED;

OPTIONS VALIDVARNAME=ANY;

DATA N_LITERAL;
   STRING1 = "'INVALID#'N";
   STRING2 = 'INVALID';
   COMP1 = COMPGED(STRING1,STRING2);
   COMP2 = COMPGED(STRING1,STRING2,'N:');
RUN;
```

```
PROC PRINT DATA=N_LITERAL NOOBS;
   TITLE "Listing of Data Set N_LITERAL";
RUN;
```

Explanation

This program demonstrates the use of the COMPGED function with a SAS n-literal. Starting with Version 7, SAS variable names could contain characters not normally allowed in SAS names. The system option VALIDVARNAME is set to "ANY" and the name is placed in quotation marks, followed by the letter N. Using the N modifier (which strips quotation marks and the 'n' from the string) and the colon modifier (which truncates the longer string to the length of the shorter string) results in a value of 0 for the variable COMP2. See the listing below:

```
                    Listing of Data Set N_LITERAL

                STRING1      STRING2    COMP1    COMP2

              'INVALID#'N    INVALID     310       0
```

SAS9.1 Function: COMPLEV

Purpose: To compute the similarity between two strings, using a method called the *Levenshtein edit distance*. It is similar to the COMPGED function except that it uses less computer resources but may not do as good a job of matching misspelled names.

Syntax: **COMPLEV(*string-1, string-2* <,*maxcost*> <,'*modifiers*'>)**

string-1 is any SAS character expression.

string-2 is any SAS character expression.

maxcost, if specified, is the maximum cost that will be returned by the COMPGED function. If the cost computation results in a value larger than *maxcost*, the value of *maxcost* will be returned.

modifiers (placed in single or double quotation marks)

i or I	ignore case.
l or L	remove leading blanks.
n or N	remove quotation marks from any argument that is an n-literal and ignore case. An n-literal is a string in quotation marks, followed by an 'n', that is useful for non-valid SAS names.
:(colon)	truncate the longer string to the length of the shorter string.

Note: If multiple modifiers are used, the order of the modifiers is important. They are applied in the same order as they appear.

Examples

String1	String2	Function	Returns
SAME	SAME	COMPLEV(STRING1, STRING2)	0
case	CASE	COMPLEV(STRING1, STRING2)	4
case	CASE	COMPLEV(STRING1,STRING2,'I')	0
case	CASE	COMPLEV(STRING1, STRING2, 999, 'I')	0
Ron	Run	COMPLEV(STRING1, STRING2)	1

Program 1.35: Demonstration of the generalized edit distance (COMPGED) and Levenshtein edit distance (COMPLEV) functions

```
***Primary functions: COMPGED and COMPLEV;

DATA _NULL_;
   INPUT @1  STRING1 $CHAR10.
         @11 STRING2 $CHAR10.;
   PUT "Function COMPGED";
   DISTANCE = COMPGED(STRING1, STRING2);
   IGNORE_CASE = COMPGED(STRING1, STRING2, 'I');
   LEAD_BLANKS = COMPGED(STRING1, STRING2, 'L');
   CASE_TRUNC = COMPGED(STRING1, STRING2, ':I');
   MAX = COMPGED(STRING1, STRING2, 250);
   PUT STRING1= STRING2= /
       DISTANCE= IGNORE_CASE= LEAD_BLANKS= CASE_TRUNC= MAX= /;

   PUT "Function COMPLEV";
   DISTANCE = COMPLEV(STRING1, STRING2);
   IGNORE_CASE = COMPLEV(STRING1, STRING2, 'I');
   LEAD_BLANKS = COMPLEV(STRING1, STRING2, 'L');
   CASE_TRUNC = COMPLEV(STRING1, STRING2, ':I');
   MAX = COMPLEV(STRING1, STRING2, 3);
   PUT STRING1= STRING2= /
       DISTANCE= IGNORE_CASE= LEAD_BLANKS= CASE_TRUNC= MAX= /;
DATALINES;
SAME      SAME
cAsE      case
Longer    Long
abcdef    xyz
   lead   lead
;
```

Explanation

In this program, all the default costs were used, so it was not necessary to call COMPCOST. Notice that the strings were read in with the $CHAR. informat so that leading blanks would be preserved. If you wish to use modifiers, you must enter them in quotation marks, in the order you want the modifying operations to proceed. Careful inspection of the SAS log below demonstrates the COMPGED and COMPLEV functions and the effects of the modifiers and the *maxcost* parameter.

SAS Log from Program 1.35

```
Function COMPGED
STRING1=SAME STRING2=SAME
DISTANCE=0 IGNORE_CASE=0 LEAD_BLANKS=0 CASE_TRUNC=0 MAX=0

Function COMPLEV
STRING1=SAME STRING2=SAME
DISTANCE=0 IGNORE_CASE=0 LEAD_BLANKS=0 CASE_TRUNC=0 MAX=0

Function COMPGED
STRING1=cAsE STRING2=case
DISTANCE=200 IGNORE_CASE=0 LEAD_BLANKS=200 CASE_TRUNC=0 MAX=200

Function COMPLEV
STRING1=cAsE STRING2=case
DISTANCE=2 IGNORE_CASE=0 LEAD_BLANKS=2 CASE_TRUNC=0 MAX=2

Function COMPGED
STRING1=Longer STRING2=Long
DISTANCE=100 IGNORE_CASE=100 LEAD_BLANKS=100 CASE_TRUNC=100 MAX=100

Function COMPLEV
STRING1=Longer STRING2=Long
DISTANCE=2 IGNORE_CASE=2 LEAD_BLANKS=2 CASE_TRUNC=2 MAX=2

Function COMPGED
STRING1=abcdef STRING2=xyz
DISTANCE=550 IGNORE_CASE=550 LEAD_BLANKS=550 CASE_TRUNC=550 MAX=250

Function COMPLEV
STRING1=abcdef STRING2=xyz
DISTANCE=6 IGNORE_CASE=6 LEAD_BLANKS=6 CASE_TRUNC=6 MAX=3

Function COMPGED
STRING1=lead STRING2=lead
DISTANCE=320 IGNORE_CASE=320 LEAD_BLANKS=0 CASE_TRUNC=320 MAX=250

Function COMPLEV
STRING1=lead STRING2=lead
DISTANCE=3 IGNORE_CASE=3 LEAD_BLANKS=0 CASE_TRUNC=3 MAX=3
```

The program below demonstrates how to use CALL COMPCOST in combination with COMPGED and the resulting differences.

Program 1.36: Changing the effect of the call to COMPCOST on the result from COMPGED

```
***Primary functions: CALL COMPCOST and COMPGED;

DATA _NULL_;
   TITLE "Program without Call to COMPCOST";
   INPUT @1  STRING1 $CHAR10.
         @11 STRING2 $CHAR10.;
   DISTANCE = COMPGED(STRING1, STRING2);
   PUT STRING1= STRING2= /
       DISTANCE=;
DATALINES;
Ron       Run
ABC       AB
;
DATA _NULL_;
   TITLE "Program with Call to COMPCOST";
   INPUT @1  STRING1 $CHAR10.
         @11 STRING2 $CHAR10.;
   IF _N_ = 1 THEN CALL COMPCOST('APPEND=',33);
   DISTANCE = COMPGED(STRING1, STRING2);
   PUT STRING1= STRING2= /
       DISTANCE=;
DATALINES;
Ron       Run
ABC       AB
;
```

Explanation

The first DATA _NULL_ program is a simple comparison of STRING1 to STRING2, using the COMPGED function. The second DATA _NULL_ program makes a call to COMPCOST (note the use of `_N_= 1`) before the COMPGED function is used. In the SAS logs below, you can see that the distance in the second observation in the first program is 50, while in the second program it is 33. That is the result of overriding the default value of 50 points for an appending error and setting it equal to 33.

SAS Log from Program without CALL COMPCOST

```
STRING1=Ron STRING2=Run
DISTANCE=100
STRING1=ABC STRING2=AB
DISTANCE=50
```

SAS Log from Program with CALL COMPCOST

```
STRING1=Ron STRING2=Run
DISTANCE=100
STRING1=ABC STRING2=AB
DISTANCE=33
```

Function: SOUNDEX

The SOUNDEX function creates a phonetic equivalent of a text string to facilitate "fuzzy" matching. You can research the details of the SOUNDEX algorithm in the *SAS OnlineDoc 9.1*. Briefly, this algorithm makes all vowels equal, along with letters that sound the same (such as 'C' and 'K'). One feature of this algorithm is that the first letters in both words must be the same to obtain a phonetic match. For those readers interested in the topic of fuzzy matching, there is an algorithm called NYSIIS, similar to the SOUNDEX algorithm, that maintains vowel position information and allows mismatches on the initial letter. A copy of a macro to implement the NYSIIS algorithm is available on the companion Web site for this book, located at **support.sas.com/companionsites**. In addition, see the COMPGED and COMPLEV functions as well as the COMPCOST call routine for alternative matching algorithms.

Purpose: To create a phonetic equivalent of a text string. Often used to attempt to match names where there might be some minor spelling differences.

Syntax: **SOUNDEX(*character-value*)**

character-value is any SAS character expression.

Program 1.37: Fuzzy matching on names using the SOUNDEX function

```
***Primary function: SOUNDEX
***Prepare data sets FILE_1 and FILE_2 to be used in the match;

DATA FILE_1;
   INPUT @1  NAME  $10.
         @11 X       1.;
DATALINES;
Friedman  4
Shields   1
MacArthur 7
ADAMS     9
Jose      5
Lundquist 9
;
DATA FILE_2;
   INPUT @1  NAME  $10.
         @11 Y       1.;
DATALINES;
Freedman  5
Freidman  9
Schields  2
McArthur  7
Adams     3
Jones     6
Londquest 9
;
***PROC SQL is used to create a Cartesian Product, combinations of all
   the names in one data set against all the names in the other;

PROC SQL;
   CREATE TABLE BOTH AS
   SELECT FILE_1.NAME AS NAME1, X,
          FILE_2.NAME AS NAME2, Y
   FROM FILE_1 ,FILE_2
QUIT;

DATA POSSIBLE;
   SET BOTH;
   SOUND_1 = SOUNDEX(NAME1);
   SOUND_2 = SOUNDEX(NAME2);
   IF SOUND_1 EQ SOUND_2;
RUN;
PROC PRINT DATA=POSSIBLE NOOBS;
   TITLE "Possible Matches between two files";
   VAR NAME1 SOUND_1 NAME2 SOUND_2 X Y;
RUN;
```

Explanation

Each of the two SAS data sets (FILE_1 and FILE_2) contain names and data. One of the best ways to see which names may be possible matches is to use PROC SQL to create what is called a **Cartesian product** of the two data sets. This is a data set (table in SQLese) that matches each observation from FILE_1 to each observation from FILE_2. Since there are six observations in FILE_1 and seven observations in FILE_2, data set BOTH contains 6 x 7 = 42 observations. To be sure this is clear, see the listing of the first 15 observations from data set BOTH:

Listing of data set BOTH
First 15 Observations

Obs	NAME1	X	NAME2	Y
1	Friedman	4	Freedman	5
2	Friedman	4	Freidman	9
3	Friedman	4	Schields	2
4	Friedman	4	McArthur	7
5	Friedman	4	Adams	3
6	Friedman	4	Jones	6
7	Friedman	4	Londquest	9
8	Shields	1	Freedman	5
9	Shields	1	Freidman	9
10	Shields	1	Schields	2
11	Shields	1	McArthur	7
12	Shields	1	Adams	3
13	Shields	1	Jones	6
14	Shields	1	Londquest	9
15	MacArthur	7	Freedman	5

The next DATA step (the data set POSSIBLE) uses the SOUNDEX function to create a SOUNDEX equivalent for each of the two names. The subsetting IF statement selects all observations where the SOUNDEX values of NAME_1 and NAME_2 are the same. A more flexible approach is shown in Program 1.39, where you can choose various values of spelling distance in determining possible matches between two names.

Possible Matches between two files

NAME1	SOUND_1	NAME2	SOUND_2	X	Y
Friedman	F6355	Freedman	F6355	4	5
Friedman	F6355	Freidman	F6355	4	9
Shields	S432	Schields	S432	1	2
MacArthur	M2636	McArthur	M2636	7	7
ADAMS	A352	Adams	A352	9	3
Lundquist	L53223	Londquest	L53223	9	9

Function: SPEDIS

The SPEDIS function is a relatively new addition to the SAS arsenal of character functions. It computes a "spelling distance" between two words. If the two words are identical, the spelling distance is 0; for each type of spelling error, SPEDIS assigns penalty points. For example, if the first letters of the two words do not match, there is a relatively large penalty. If two letters are reversed (such as ie or ei for example), there is a smaller penalty. The final spelling distance is also based on the length of the words being matched. A wrong letter in a long word results in a smaller score than a wrong letter in a shorter word.

Take a look at two new functions (COMPGED and COMPLEV) that are similar to the SPEDIS function. The COMPGED function even has an associated call routine (COMPCOST) that allows you to adjust the costs for various classes of spelling errors.

There are some very interesting applications of the SPEDIS, COMPGED, and COMPLEV functions. One, described next, allows you to match names that are not spelled exactly the same. You may want to try this same program with each of the alternative functions to compare matching ability and computer efficiency.

Purpose: To compute the spelling distance between two words. The more alike the two words are, the lower the score. A score of 0 indicates an exact match.

Syntax: **SPEDIS(*word-1, word-2*);**

word-1 is any SAS character expression.

word-2 is any SAS character expression.

The function returns the spelling distance between the two words. A zero indicates the words are identical. Higher scores indicate the words are more dissimilar. Note that SPEDIS is asymmetric—the order of the values being compared makes a difference. The value returned by the function is computed as a percentage of the length of the string to be compared. Thus, an error in two short words returns a larger value than the same error in two longer words.

A list of the operations and their "cost" is shown in the table below. This will give you an idea of the severity of different types of spelling errors. Swapping (interchanging) two letters has a smaller penalty than inserting a letter, and errors involving the first character in a string have a larger penalty than errors involving letters in the middle of the string.

Operation	Cost	Explanation
MATCH	0	no change
SINGLET	25	delete one of a double letter
DOUBLET	50	double a letter
SWAP	50	reverse the order of two consecutive letters
TRUNCATE	50	delete a letter from the end
APPEND	35	add a letter to the end
DELETE	50	delete a letter from the middle
INSERT	100	insert a letter in the middle
REPLACE	100	replace a letter in the middle
FIRSTDEL	100	delete the first letter
FIRSTINS	200	insert a letter at the beginning
FIRSTREP	200	replace the first letter

Examples

For these examples `WORD1="Steven" WORD2 = "Stephen"` and `WORD3 = "STEVEN"`

Function	Returns
`SPEDIS(WORD1,WORD2)`	25
`SPEDIS(WORD2,WORD1)`	28
`SPEDIS(WORD1,WORD3)`	83
`SPEDIS(WORD1,"Steven")`	0

Program 1.38: Using the SPEDIS function to match social security numbers that are the same or differ by a small amount

```
***Primary function: SPEDIS;

DATA FIRST;
   INPUT ID_1 : $2. SS : $11.;
DATALINES;
1A 123-45-6789
2A 111-45-7654
3A 999-99-9999
4A 222-33-4567
;
DATA SECOND;
   INPUT ID_2 : $2. SS : $11.;
DATALINES;
1B 123-45-6789
2B 111-44-7654
3B 899-99-9999
4B 989-99-9999
5B 222-22-5467
;
%LET CUTOFF = 10;
PROC SQL;
   TITLE "Output from SQL when CUTOFF is set to &CUTOFF";
   SELECT ID_1,
          FIRST.SS AS FIRST_SS,
          ID_2,
          SECOND.SS AS SECOND_SS
   FROM FIRST, SECOND
   WHERE SPEDIS(FIRST.SS, SECOND.SS) LE &CUTOFF;
QUIT;
```

Explanation

In this example, you want to match social security numbers between two files, where the two numbers may be slightly different (one digit changed or two digits transposed for example). Even though social security numbers are made up of digits, you can treat the values just like any other alphabetic characters. The program is first run with the cutoff value set to 10, with the result shown next:

```
Output from SQL when CUTOFF is set to 10

 ID_1  FIRST_SS     ID_2  SECOND_SS
 ____  ___________  ____  ___________
 1A    123-45-6789  1B    123-45-6789
 2A    111-45-7654  2B    111-44-7654
 3A    999-99-9999  4B    989-99-9999
```

When you run the same program with the cutoff value set to 20, you obtain an additional match—numbers 3A and 3B. These two numbers differ in the first digit, which resulted in a larger penalty. Here is the listing with CUTOFF set to 20:

```
Output from SQL when CUTOFF is set to 20

 ID_1  FIRST_SS     ID_2  SECOND_SS
 ____  ___________  ____  ___________
 1A    123-45-6789  1B    123-45-6789
 2A    111-45-7654  2B    111-44-7654
 3A    999-99-9999  3B    899-99-9999
 3A    999-99-9999  4B    989-99-9999
```

Program 1.39: Fuzzy matching on names using the spelling distance (SPEDIS) function

```
***Primary function: SPEDIS
***Other function: UPCASE;

***See Program 1-37 for the creation of data set BOTH and the
   explanation of the PROC SQL code.;

DATA POSSIBLE;
   SET BOTH;
   DISTANCE = SPEDIS(UPCASE(NAME1),UPCASE(NAME2));
RUN;

PROC PRINT DATA=POSSIBLE NOOBS;
   WHERE DISTANCE LE 15;
   TITLE "Possible Matches between two files";
   VAR NAME1 NAME2 DISTANCE X Y;
RUN;
```

Explanation

This program starts at the point following PROC SQL in Program 1.37.

The next DATA step (data set POSSIBLE) uses the SPEDIS function to compute the spelling distance between every pair of names in the data set BOTH. Notice that all the names are converted to uppercase, using the UPCASE function, to avoid missing possible matches because of case differences. (Note: the COMPGED and COMPLEV functions allow a modifier to ignore case in the comparison.) You might want to add a subsetting IF statement to this DATA step to limit pairs of names with a given spelling distance. For the purpose of demonstration, this was not done here, and the WHERE statement in the following PROC PRINT is used to select possible matches. The smaller the value of DISTANCE, the closer the match. If DISTANCE equals 0, the two names are an exact match. To see the results of all names within a spelling distance of 15, take a look at the output from the PROC PRINT. As you can see from the listing below, this method of fuzzy matching produces a different list of matches from the one produced using the SOUNDEX function.

```
                Possible Matches between two files

NAME1        NAME2       DISTANCE    X    Y

Friedman     Freedman        12       4    5
Friedman     Freidman         6       4    9
Shields      Schields         7       1    2
MacArthur    McArthur        11       7    7
ADAMS        Adams            0       9    3
```

Functions That Divide Strings into "Words"

These extremely useful functions and call routines can divide a string into words. Words can be characters separated by blanks or other delimiters that you specify.

SCAN and SCANQ

The two functions, SCAN and SCANQ are similar. They both extract "words" from a string, **words** being defined as characters separated by a set of specified delimiters. Pay particular attention to the fact that the SCAN and SCANQ functions use different sets of default delimiters. The SCANQ function also has some additional useful features. Programs demonstrating both of these functions follow the definitions.

Function: SCAN

Purpose: Extracts a specified word from a character expression, where **word** is defined as the characters separated by a set of specified delimiters. The length of the returned variable is 200, unless previously defined.

Syntax: **SCAN(*character-value, n-word <,'delimiter-list'>*)**

character-value is any SAS character expression.

n-word is the *n*th "word" in the string. If *n* is greater than the number of words, the SCAN function returns a value that contains no characters. If *n* is negative, the character value is scanned from right to left. A value of zero is invalid.

delimiter-list is an optional argument. If it is omitted, the default set of delimiters are (for ASCII environments):

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

For EBCDIC environments, the default delimiters are:

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

If you specify any delimiters, only those delimiters will be active. Delimiters before the first word have no effect. Two or more contiguous delimiters are treated as one.

Examples

For these examples `STRING1 = "ABC DEF"` and `STRING2 = "ONE?TWO THREE+FOUR|FIVE"` This is an ASCII example.

Function	Returns
`SCAN(STRING1,2)`	`"DEF"`
`SCAN(STRING1,-1)`	`"DEF"`
`SCAN(STRING1,3)`	no characters
`SCAN(STRING2,4)`	`"FIVE"`
`SCAN(STRING2,2," ")`	`"THREE+FOUR|FIVE"`
`SCAN(STRING1,0)`	An error in the SAS log

SAS9.1 Function: SCANQ

Purpose: To extract a specified word from a character expression, **word** being defined as characters separated by a set of specified delimiters. The basic differences between this function and the SCAN function are the default set of delimiters (see syntax below) and the fact that a value of 0 for the word count does not result in an error message. SCANQ also ignores delimiters enclosed in quotation marks (SCAN recognizes them).

Syntax: **SCANQ(*character-value, n-word* <,'*delimiter-list*'>)**

character-value is any SAS character expression.

n-word is the *n*th "word" in the string (**word** being defined as one or more characters, separated by a set of specified delimiters. If *n* is negative, the scan proceeds from right to left. If *n* is greater than the number of words or 0, the SCANQ function will return a blank value. Delimiters located before the first word or after the last word are ignored. If two or more delimiters are located between two words, they are treated as one. If the character value contains sets of quotation marks, any delimiters within these marks are ignored.

delimiter-list is an optional argument. If it is omitted, the default set of delimiters are white space characters (blank, horizontal and vertical tab, carriage return, line feed, and form feed). **Note: this is a different set of default delimiters from the SCAN function.** If you specify any delimiters, only those delimiters will be active. You cannot specify single or double quotation marks as delimiters.

Examples

For these examples `STRING1 = "ABC DEF", STRING2 = "ONE TWO THREE FOUR FIVE", STRING3 = "'AB CD' 'X Y'",` and `STRING4 = "ONE# ::TWO"`

Function	Returns
`SCANQ(STRING1,2)`	`"DEF"`
`SCANQ(STRING1,-1)`	`"DEF"`
`SCANQ(STRING1,3)`	no characters
`SCANQ(STRING2,4," ")`	`"FOUR"`
`SCANQ(STRING3,2)`	`"'X Y'"`
`SCANQ(STRING1,0)`	no characters
`SCANQ(STRING4,2," #:")`	`"TWO"`

Program 1.40: A novel use of the SCAN function to convert mixed numbers to decimal values

```
***Primary function: SCAN
***Other function: INPUT;

DATA PRICES;
   INPUT @1 STOCK $3.
         @5 MIXED $6.;
   INTEGER = SCAN(MIXED,1,'/ ');
   NUMERATOR = SCAN(MIXED,2,'/ ');
   DENOMINATOR = SCAN(MIXED,3,'/ ');
   IF NUMERATOR = ' ' THEN VALUE = INPUT(INTEGER,8.);
   ELSE VALUE = INPUT(INTEGER,8.) +
                (INPUT(NUMERATOR,8.) / INPUT(DENOMINATOR,8.));
   KEEP STOCK VALUE;
DATALINES;
ABC 14 3/8
XYZ 8
TWW 5 1/8
;
PROC PRINT DATA=PRICES NOOBS;
   TITLE "Listing of Data Set PRICES";
RUN;
```

Explanation

The SCAN function has many uses besides merely extracting selected words from text expressions. In this program, you want to convert numbers such as 23 5/8 into a decimal value (23.675). An elegant way to accomplish this is to use the SCAN function to separate the mixed number into three parts: the integer, the numerator of the fraction, and the denominator. Once this is done, all you need to do is to convert each piece to a numerical value (using the INPUT function) and add the integer portion to the fractional portion. If the number being processed does not have a fractional part, the SCAN function returns a blank value for the two variables NUMERATOR and DENOMINATOR. The listing is shown below:

```
                        Listing of Data Set PRICES

                            STOCK     VALUE

                             ABC     14.375
                             XYZ      8.000
                             TWW      5.125
```

Program 1.41: Program to read a tab-delimited file

```
***Primary function: SCANQ;

DATA READ_TABS;
   INFILE 'C:\BOOKS\FUNCTIONS\TAB_FILE.TXT' PAD;
   INPUT @1 STRING $30.;
   LENGTH FIRST MIDDLE LAST $ 12;
   FIRST = SCANQ(STRING,1);
   MIDDLE = SCANQ(STRING,2);
   LAST = SCANQ(STRING,3);
   DROP STRING;
RUN;
PROC PRINT DATA=READ_TABS NOOBS;
   TITLE "Listing of Data Set READS_TABS";
RUN;
```

Explanation

This program reads values separated by tab characters. Although you can use the INFILE option DLM='09'X (the ASCII Hex value for a tab character, or '05'X for EBCDIC) to read this file, the SCANQ function provides an easy, alternate method. Here you take advantage of the fact that one of the default delimiters for the SCANQ function is a tab character. This method could be especially useful if you imported a file from another system, and individual character values contained tabs or other non-printing white space characters.

A listing of the resulting data set is shown next:

```
                        Listing of Data Set READS_TABS

                           FIRST     MIDDLE    LAST

                           Ron       P.        Cody
                           Ralph     Waldo     Emerson
                           Alfred    E.        Newman
```

Program 1.42: Alphabetical listing by last name when the name field contains first name, possibly middle initial, and last name

```
***Primary function: SCAN;

***Making the problem a little harder.  Extracting the last name
   when there may or may not be a middle initial;

DATA FIRST_LAST;
   INPUT @1  NAME  $20.
         @21 PHONE $13.;
   ***Extract the last name from NAME;
   LAST_NAME = SCAN(NAME,-1,' '); /* Scans from the right */
DATALINES;
Jeff W. Snoker      (908)782-4382
Raymond Albert      (732)235-4444
Steven J. Foster    (201)567-9876
Jose Romerez        (516)593-2377
;
PROC REPORT DATA=FIRST_LAST NOWD;
   TITLE "Names and Phone Numbers in Alphabetical Order (by Last Name)";
   COLUMNS NAME PHONE LAST_NAME;
   DEFINE LAST_NAME / ORDER NOPRINT WIDTH=20;
   DEFINE NAME      / DISPLAY 'Name' LEFT WIDTH=20;
   DEFINE PHONE     / DISPLAY 'Phone Number' WIDTH=13 FORMAT=$13.;
RUN;
```

Explanation

It is easy to extract the last name by using a –1 as the second argument of the SCAN function. A negative value for this argument results in a scan from right to left. Output from the REPORT procedure is shown below:

```
        Names and Phone Numbers in Alphabetical Order (by Last Name)

                  Name                 Phone Number
                  Raymond Albert       (732)235-4444
                  Steven J. Foster     (201)567-9876
                  Jose Romerez         (516)593-2377
                  Jeff W. Snoker       (908)782-4382
```

CALL SCAN and CALL SCANQ

The SCAN and SCANQ call routines are similar to the SCAN and SCANQ functions. But both call routines return a position and length of the *n*th word (to be used, perhaps, in a subsequent SUBSTR function) rather than the actual word itself.

Differences between CALL SCAN and CALL SCANQ are the same differences between the two functions, SCAN and SCANQ.

SAS9.1 Function: CALL SCAN

Purpose: To break up a string into words, where **words** are defined as the characters separated by a set of specified delimiters, and to return the starting position and the length of the *n*th word.

Syntax: **CALL SCAN(*character-value*, *n-word*, *position*, *length* <,*'delimiter-list'*>)**

character-value is any SAS character expression.

n-word is the nth "word" in the string. If *n* is greater than the number of words, the SCAN call routine returns a value of 0 for *position* and *length*. If *n* is negative, the scan proceeds from right to left.

position is the name of the numeric variable to which the starting position in the *character-value* of the *n*th word is returned.

length is the name of a numeric variable to which the length of the *n*th word is returned.

delimiter-list is an optional argument. If it is omitted, the default set of delimiters are (for ASCII environments):

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

For EBCDIC environments, the default delimiters are:

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

If you specify any delimiters, only those delimiters will be active. Delimiters are slightly different in ASCII and EBCDIC systems.

Examples

For these examples `STRING1 = "ABC DEF"` and `STRING2 = "ONE?TWO THREE+FOUR|FIVE"`

Function	Position	Length
`CALL SCAN(STRING1,2,POSITION,LENGTH)`	5	3
`CALL SCAN(STRING1,-1,POSITION,LENGTH)`	5	3
`CALL SCAN(STRING1,3,POSITION,LENGTH)`	0	0
`CALL SCAN(STRING2,1,POSITION,LENGTH)`	1	7
`CALL SCAN(STRING2,4,POSITION,LENGTH)`	20	4
`CALL SCAN(STRING2,2,POSITION,LENGTH," ")`	9	15
`CALL SCAN(STRING1,0,POSITION,LENGTH)`	missing	missing

Program 1.43: Demonstrating the SCAN call routine

```
***Primary function: CALL SCAN;

DATA WORDS;
   INPUT STRING $40.;
   DELIM = 'Default';
   N = 2;
   CALL SCAN(STRING,N,POSITION,LENGTH);
   OUTPUT;
   N = -1;
   CALL SCAN(STRING,N,POSITION,LENGTH);
   OUTPUT;
   DELIM = '#';
   N = 2;
   CALL SCAN(STRING,N,POSITION,LENGTH,'#');
   OUTPUT;
DATALINES;
ONE TWO THREE
One*#Two Three*Four
;
PROC PRINT DATA=WORDS NOOBS;
   TITLE "Listing of Data Set WORDS";
RUN;
```

Explanation

The SCAN routine is called three times in this program, twice with default delimiters and once with the pound sign (#) as the delimiter. Notice that using a negative argument results in a scan from right to left. The output from this program is shown next:

```
                    Listing of Data Set WORDS

      STRING              DELIM       N    POSITION    LENGTH

ONE TWO THREE             Default      2        5          3
ONE TWO THREE             Default     -1        9          5
ONE TWO THREE             #            2        0          0
One*#Two Three*Four       Default      2        5          4
One*#Two Three*Four       Default     -1       16          4
One*#Two Three*Four       #            2        6         35
```

SAS9.1 Function: CALL SCANQ

Purpose: To break up a string into words, where *words* are defined to be the characters separated by a set of specified delimiters, and to return the starting position and the length of the *n*th word. The basic differences between this call routine and CALL SCAN is that CALL SCANQ uses white space characters as default delimiters and it can accept a value of 0 for the *n*-word argument. In addition, the SCANQ call routine ignores delimiters within quotation marks.

Syntax: **CALL SCAN(*character-value*, *n-word*, *position*, *length* <,'*delimiter-list*'>)**

character-value is any SAS character expression.

n-word is the *n*th "word" in the string. If *n* is zero, or if the absolute value of *n* is greater than the number of words, the SCANQ call routine returns values of 0 for *position* and *length*.

position is the name of the numeric variable to which the starting position in the *character-value* of the *n*th word is returned.

length is the name of a numeric variable to which the length of the *n*th word is returned.

delimiter-list is an optional argument. If it is omitted, the default set of delimiters are white space characters (blank, horizontal and vertical tab, carriage return, line feed, and form feed). Also, the beginning or end of a line also delimits a word. **Note: this is a different set of default delimiters from the SCAN call routine.** If you specify any delimiters, only those delimiters will be active. Also note that in the last example (STRING3), *position* and *length* include single quotation marks.

If you specify any delimiters, only those delimiters will be active.

Examples

For these examples `STRING1 = "ABC DEF"` and `STRING2 = "ONE TWO THREE FOUR FIVE",` and `STRING3 = "'AB CD' 'X Y'"`

Function	Position	Length
CALL SCANQ(STRING1,2,POSITION,LENGTH)	5	3
CALL SCANQ(STRING1,-1,POSITION,LENGTH)	5	3
CALL SCANQ(STRING1,3,POSITION,LENGTH)	0	0
CALL SCANQ(STRING2,4,POSITION,LENGTH)	5	3
CALL SCANQ(STRING2,2,POSITION,LENGTH," ")	9	15
CALL SCANQ(STRING1,0,POSITION,LENGTH)	0	0
CALL SCANQ(STRING3,2,POSITION,LENGTH)	9	5

Program 1.44: Using CALL SCANQ to count the words in a string

```
***Primary function: CALL SCANQ;

DATA COUNT;
   INPUT STRING $40.;
   DO I = 1 TO 99 UNTIL (LENGTH EQ 0);
      CALL SCANQ(STRING,I,POSITION,LENGTH);
   END;
   NUM_WORDS = I-1;
   DROP POSITION LENGTH I;
DATALINES;
ONE TWO THREE
ONE TWO
ONE
;
PROC PRINT DATA=COUNT NOOBS;
   TITLE "Listing of Data Set COUNT";
RUN;
```

Explanation

When the value of the second argument in the CALL SCANQ routine is greater than the number of words in a string, both the position and length values are set to 0. Here, the call routine is placed in a DO loop, which iterates until the routine returns a value of 0 for

the length. The number of words is, therefore, one fewer than this number. The listing is shown below:

```
                    Listing of Data Set COUNT

                                      NUM_
STRING                               WORDS

ONE TWO THREE                          3
ONE TWO                                2
ONE                                    1
```

Functions That Substitute Letters or Words in Strings

TRANSLATE can substitute one character for another in a string. TRANWRD is more flexible—it can substitute a word or several words for one or more words.

Function: TRANSLATE

Purpose To exchange one character value for another. For example, you might want to change values 1–5 to the values A–E.

Syntax: **TRANSLATE(*character-value, to-1, from-1 <,… to-n, from-n*>)**

character-value is any SAS character expression.

to-n is a single character or a list of character values.

from-n is a single character or a list of characters.

Each character listed in *from-n* is changed to the corresponding value in *to-n*. If a character value is not listed in from-n, it will be unaffected.

Examples

In these examples, `CHAR = "12X45", ANS = "Y"`

Function	Returns
`TRANSLATE(CHAR,"ABCDE","12345")`	`"ABXDE"`
`TRANSLATE(CHAR,'A','1','B','2','C','3','D','4','E','5')`	`"ABXDE"`
`TRANSLATE(ANS,"10","YN")`	`"1"`

Program 1.45: Converting values of '1','2','3','4', and '5' to 'A','B','C','D', and 'E' respectively

```
***Primary function: TRANSLATE;

DATA MULTIPLE;
   INPUT QUES : $1. @@;
   QUES = TRANSLATE(QUES,'ABCDE','12345');
DATALINES;
1 4 3 2 5
5 3 4 2 1
;
PROC PRINT DATA=MULTIPLE NOOBS;
   TITLE "Listing of Data Set MULTIPLE";
RUN;
```

Explanation

In this example, you want to convert the character values of 1–5 to the letters A–E. The two arguments in this function seem backwards to this author. You would expect the order to be "from–to" rather than the other way around. I suppose others at SAS felt the same way, since a more recent function, TRANWRD (next example), uses the "from – to" order for its arguments. While you could use a format, along with a PUT function to do this translation,

the TRANSLATE function is more compact and easier to use in cases like this. A listing of data set MULTIPLE follows:

```
Listing of Data Set MULTIPLE

QUES

 A
 D
 C
 B
 E
 E
 C
 D
 B
 A
```

Program 1.46: Converting the values "Y" and "N" to 1's and 0's

```
***Primary functions: TRANSLATE, UPCASE
***Other functions: INPUT;

DATA YES_NO;
   LENGTH CHAR $ 1;
   INPUT CHAR @@;
   X = INPUT(
       TRANSLATE(
       UPCASE(CHAR),'01','NY'),1.);
DATALINES;
N Y n y A B 0 1
;
PROC PRINT DATA=YES_NO NOOBS;
   TITLE "Listing of Data Set YES_NO";
RUN;
```

Explanation

This rather silly program was written mainly to demonstrate the TRANSLATE and UPCASE functions. A couple of IF statements, combined with the UPCASE function in the DATA step would probably be more straightforward. In this program, the UPCASE function converts lowercase values of "n" and "y" to their uppercase equivalents. The

TRANSLATE function then converts the Ns and Ys to the characters "0" and "1," respectively. Finally, the INPUT function does the character to numeric conversion. Note that the data values of "1" and "0" do not get translated, but do get converted to numeric values. As you can see in the listing below, the program does get the job done.

```
                    Listing of Data Set YES_NO

                         CHAR    X

                         N       0
                         Y       1
                         n       0
                         y       1
                         A       .
                         B       .
                         0       0
                         1
```

Function: TRANWRD

Purpose: To substitute one or more words in a string with a replacement word or words. It works like the find and replace feature of most word processors.

Syntax: **TRANWRD(*character-value*, *from-string*, *to-string*)**

character-value is any SAS character expression.

from-string is one or more characters that you want to replace with the character or characters in the to-string.

to-string is one or more characters that replace the entire *from-string*.

Making the analogy to the find and replace feature of most word processors here, *from-string* represents the string to find and *to-string* represents the string to replace. Notice that the order of *from-* and *to-string* in this function is opposite (and more logical to this author) from the order in the TRANSLATE function.

Examples

For these examples `STRING = "123 Elm Road"`, `FROM = "Road"` and `TO = "Rd."`

Function	Returns
`TRANWRD(STRING,FROM,TO)`	`"123 Elm Rd."`
`TRANWRD("Now is the time","is","is not")`	`"Now is not the time"`
`TRANWRD("one two three","four","4")`	`"one two three"`
`TRANWRD("Mr. Rogers","Mr."," ")`	`"  Rogers"`
`TRANWRD("ONE TWO THREE","ONE TWO","A B")`	`"A B THREE"`

Program 1.47: Converting words such as Street to their abbreviations such as St. in an address

```
***Primary function: TRANWRD;

DATA CONVERT;
   INPUT @1 ADDRESS $20. ;
   *** Convert Street, Avenue and Road to their abbreviations;
   ADDRESS = TRANWRD(ADDRESS,'Street','St.');
   ADDRESS = TRANWRD (ADDRESS,'Avenue','Ave.');
   ADDRESS = TRANWRD (ADDRESS,'Road','Rd.');
DATALINES;
89 Lazy Brook Road
123 River Rd.
12 Main Street
;
PROC PRINT DATA=CONVERT;
   TITLE 'Listing of Data Set CONVERT';
RUN;
```

Explanation

TRANWRD is one of the relatively new SAS functions—and it is enormously useful. This example uses it to help standardize a mailing list, substituting abbreviations for full words. Another use for this function is to make *`to-string`* a blank, thus allowing you to remove

words such as Jr. or Mr. from an address. The converted addresses are shown in the listing below:

```
Listing of Data Set CONVERT

Obs    ADDRESS

 1     89 Lazy Brook Rd.
 2     123 River Rd.
 3     12 Main St.
```

Functions That Compute the Length of Strings

These four functions compute the length of character values. The LENGTH function (the oldest of the lot) does not count trailing blanks in its calculation. The LENGTHN function is identical to the LENGTH function with one exception: If there is a null string (technically speaking, a string consisting of all blanks), the LENGTH function returns a value of 1 while the LENGTHN function returns a value of 0. I would recommend using the LENGTHN function as your general purpose length function in place of the older LENGTH function (unless, of course, you used the fact that the length of a null string is 1 instead of 0 in your program). The LENGTHC function operates like the LENGTH function except it counts trailing blanks in its computation. Finally, LENGTHM computes the length used to store this variable in memory. In most applications, the LENGTHM and LENGTHC functions return the same value. You may see some differences when working with macro variables. The LENGTHM function is a useful way to determine the storage length of a character variable (instead of using PROC CONTENTS, for example).

Function: LENGTH

Purpose: To determine the length of a character value, not counting trailing blanks. A null argument returns a value of 1.

Syntax: **LENGTH(*character-value*)**

character-value is any SAS character expression.

Examples

For these examples `CHAR = "ABC  "`

Function	Returns
`LENGTH("ABC")`	3
`LENGTH(CHAR)`	3
`LENGTH(" ")`	1

SAS9.1 Function: LENGTHC

Purpose: To determine the length of a character value, including trailing blanks.

Syntax: **`LENGTHC(`*`character-value`*`)`**

`character-value` is any SAS character expression.

Examples

For these examples `CHAR = "ABC  "`

Function	Returns
`LENGTH("ABC")`	3
`LENGTH(CHAR)`	6
`LENGTH(" ")`	1

SAS9.1 Function: LENGTHM

Purpose: To determine the length of a character variable in memory.

Syntax: **`LENGTHM(`*`character-value`*`)`**

`character-value` is any SAS character expression.

Examples

For these examples `CHAR = "ABC    "`

Function	Returns
`LENGTHM("ABC")`	`3`
`LENGTHM(CHAR)`	`6`
`LENGTHM(" ")`	`1`

SAS9.1 Function: LENGTHN

Purpose: To determine the length of a character value, not counting trailing blanks. A null argument returns a value of 0.

Syntax: **`LENGTHN(`*`character-value`*`)`**

`character-value` is any SAS character expression.

Examples

For these examples `CHAR = "ABC    "`

Function	Returns
`LENGTH("ABC")`	`3`
`LENGTH(CHAR)`	`3`
`LENGTH(" ")`	`0`

Program 1.48: Demonstrating the LENGTH, LENGTHC, LENGTHM, and LENGTHN functions

```
***Primary functions: LENGTH, LENGTHC, LENGTHM, LENGTHN;

DATA LENGTH_FUNC;
   NOTRAIL = "ABC";
   TRAIL   = "DEF   "; * Three trailing blanks;
   NULL    = " ";      * Null string;
   LENGTH_NOTRAIL = LENGTH(NOTRAIL);
   LENGTH_TRAIL   = LENGTH(TRAIL);
```

```
   LENGTH_NULL    = LENGTH(NULL);
   LENGTHC_NOTRAIL = LENGTHC(NOTRAIL);
   LENGTHC_TRAIL   = LENGTHC(TRAIL);
   LENGTHC_NULL    = LENGTHC(NULL);
   LENGTHM_NOTRAIL = LENGTHM(NOTRAIL);
   LENGTHM_TRAIL   = LENGTHM(TRAIL);
   LENGTHM_NULL    = LENGTHM(NULL);
   LENGTHN_NOTRAIL = LENGTHN(NOTRAIL);
   LENGTHN_TRAIL   = LENGTHN(TRAIL);
   LENGTHN_NULL    = LENGTHN(NULL);
RUN;

PROC PRINT DATA=LENGTH_FUNC NOOBS HEADING=H;
   TITLE "Listing of Data Set LENGTH_FUNC";
RUN;
```

Explanation

The LENGTH and LENGTHN functions return the length of a character variable, **not counting trailing blanks**. The only difference between the LENGTH and LENGTHN functions is that the LENGTH function returns a value of 1 for a null string while the LENGTHN function returns a 0. The LENGTHC function **does count trailing blanks** in its calculations. Finally, the LENGTHM function returns the number of bytes of memory used to store the variable. Notice in this program that the LENGTHM and LENGTHC functions yield the same value. Look over the listing below to be sure you understand the differences among these functions:

```
                    Listing of Data Set LENGTH_FUNC

                     LENGTH_ LENGTH_ LENGTH_ LENGTHC_ LENGTHC_ LENGTHC_
    NOTRAIL TRAIL NULL NOTRAIL  TRAIL    NULL   NOTRAIL   TRAIL    NULL

      ABC    DEF          3       3       1        3        6        1

    LENGTHM_    LENGTHM_    LENGTHM_    LENGTHN_    LENGTHN_    LENGTHN_
     NOTRAIL      TRAIL       NULL       NOTRAIL      TRAIL       NULL

        3           6           1           3           3           0
```

Functions That Count the Number of Letters or Substrings in a String

The COUNT function counts the number of times a given substring appears in a string. The COUNTC function counts the number of times specific characters occur in a string.

SAS9.1 Function: COUNT

Purpose: To count the number of times a given substring appears in a string. With the use of a modifier, case can be ignored. If no occurrences of the substring are found, the function returns a 0.

Syntax: **COUNT(*character-value*, *find-string* <,'*modifiers*'>)**

character-value is any SAS character expression.

find-string is a character variable or SAS string literal to be counted.

The following modifiers, placed in single or double quotation marks, may be used with COUNT:

`i` or `I` ignore case.

`t` or `T` ignore trailing blanks in both the character value and the *find-string*.

Examples

For these examples, `STRING1 = "How Now Brown COW"` and `STRING2 = "ow"`

Function	Returns
`COUNT(STRING1, STRING2)`	`3`
`COUNT(STRING1,STRING2,'I')`	`4`
`COUNT(STRING1, "XX")`	`0`
`COUNT("ding and dong","g ")`	`1`
`COUNT("ding and dong","g ","T")`	`2`

Program 1.49: Using the COUNT function to count the number of times the word "the" appears in a string

```
***Primary Function: COUNT;

DATA DRACULA;
   INPUT STRING $CHAR60.;
   NUM = COUNT(STRING,"the");
   NUM_NO_CASE = COUNT(STRING,"the",'I');
DATALINES;
The number of times "the" appears is the question
THE the
None on this line!
There is the map
;
PROC PRINT DATA=DRACULA NOOB;
   TITLE "Listing of Data Set Dracula";
RUN;
```

Explanation

In this program, the COUNT function is used with and without the I (ignore case) modifier. In the first observation, the first "The" has an uppercase T, so it does not match the substring and is not counted for the variable NUM. But when the I modifier is used, it does count. The same holds for the second observation. When there are no occurrences of the substring, as in the third observation, the function returns a 0. The fourth line of data demonstrates that COUNT ignores word boundaries when searching for strings. A listing of data set DRACULA is displayed below:

Listing of Data Set DRACULA

STRING	NUM	NUM_NO_CASE
The number of times "the" appears is the question	2	3
THE the	1	2
None on this line!	0	0
There is the map	1	2

SAS9.1 Function: COUNTC

Purpose: To count the number of individual characters that appear or do not appear in a string. With the use of a modifier, case can be ignored. Another modifier allows you to count characters that do not appear in the string. If no specified characters are found, the function returns a 0.

Syntax: **COUNTC(*character-value*, *characters* <,'*modifiers*'>)**

character-value is any SAS character expression.

characters is one or more characters to be counted. It may be a string literal (letters in quotation marks) or a character variable.

The following modifiers, placed in quotation marks, may be used with COUNTC:

`i` or `I` ignore case.

`o` or `O` If this modifier is used, COUNTC processes the character or characters and modifiers only once. If the COUNTC function is used in the same DATA step, the previous character and modifier values are used and the current values are ignored.

`t` or `T` ignore trailing blanks in the *character-value* or the `characters`. Note, this modifier is especially important when looking for blanks or when you are using the `v` modifier (below).

`v` or `V` count only the characters that do **not** appear in the `character-value`. Remember that this count will include trailing blanks unless the `t` modifier is used.

Examples

For these examples, STRING1 = "How Now Brown COW" and STRING2 = "wo"

Function	Returns
`COUNTC("AaBbbCDE","CBA")`	`3`
`COUNTC("AaBbbCDE","CBA",'I')`	`7`
`COUNTC(STRING1, STRING2)`	`6`
`COUNTC(STRING1,STRING2,'I')`	`8`
`COUNTC(STRING1, "XX")`	`0`
`COUNTC("ding and dong","g ")`	`4` (2 g's and 2 blanks)
`COUNTC("ding and dong","g ","T")`	`2` (blanks trimmed)
`COUNTC("ABCDEabcde","BCD",'VI')`	`4` (A, E, a, and e)

Program 1.50: Demonstrating the COUNTC function to find one or more characters or to check if characters are not present in a string

```
***Primary Function: COUNTC;

DATA COUNT_CHAR;
   INPUT STRING $20.;
   NUM_A = COUNTC(STRING,'A');
   NUM_Aa = COUNTC(STRING,'a','i');
   NUM_A_OR_B = COUNTC(STRING,'AB');
   NOT_A = COUNTC(STRING,'A','v');
   NOT_A_TRIM = COUNTC(STRING,'A','vt');
   NOT_Aa = COUNTC(STRING,'A','iv');
DATALINES;
UPPER A AND LOWER a
abAB
BBBbbb
;
PROC PRINT DATA=COUNT_CHAR;
   TITLE "Listing of Data Set COUNT_CHAR";
RUN;
```

Explanation

This program demonstrates several features of the COUNTC function. The first use of the function simply looks for the number of times the uppercase letter A appears in the string. Next, by adding the i modifier, the number of upper- or lowercase A's is counted. Next, when you place more than one character in the list, the function returns the total number of the listed characters. The v modifier is interesting. The first time it is used, COUNTC is counting the number of characters in the string that are not uppercase A's. Notice in the listing below, that this count includes the trailing blanks. However, in the next statement of the program, when the v and t modifiers are used together, the trailing blanks are not counted.

Listing of Data Set COUNT_CHAR

Obs	STRING	NUM_A	NUM_Aa	NUM_A_ OR_B	NOT_A	NOT_A_ TRIM	NOT_Aa
1	UPPER A AND LOWER a	2	3	2	18	17	17
2	abAB	1	2	2	19	3	18
3	BBBbbb	0	0	3	20	6	20

Miscellaneous String Functions

Don't be put off by the "miscellaneous" in this heading. Many of these functions are extremely useful—they just didn't fit neatly into categories.

Function: MISSING

Purpose: To determine if the argument is a missing (character or numeric) value. This is a handy function to use since you don't have to know if the variable you are testing is character or numeric. The function returns a 1 (true) if the value is a missing value, a 0 (false) otherwise.

Syntax: **MISSING(*variable*)**

variable is a character or numeric variable or expression.

Examples:

For these examples, `NUM1 = 5`, `NUM2 = .`, `CHAR1 = "ABC"`, and `CHAR2 = " "`

Function	Returns
`MISSING(NUM1)`	`0`
`MISSING(NUM2)`	`1`
`MISSING(CHAR1)`	`0`
`MISSING(CHAR2)`	`1`

Program 1.51: Determining if there are any missing values for all variables in a data set

```
***Primary function: MISSING
***Other function: DIM;

***First, create a data set for testing;
DATA TEST_MISS;
   INPUT @1 (X Y Z)(1.)
         @4 (A B C D)($1.);
DATALINES;
123ABCD
..7 FFF
987RONC
;
DATA FIND_MISS;
   SET TEST_MISS END=LAST;
   ARRAY NUMS[*] _NUMERIC_;
   ARRAY CHARS[*] _CHARACTER_;
   DO I = 1 TO DIM(NUMS);
      IF MISSING(NUMS[I]) THEN NN + 1;
   END;
   DO I = 1 TO DIM(CHARS);
      IF MISSING(CHARS[I]) THEN NC + 1;
   END;
   FILE PRINT;
   TITLE "Count of Missing Values";
   IF LAST THEN PUT NN "Numeric and " NC "Character values missing";
RUN;
```

Explanation

Notice that the MISSING function can take either a numeric or a character argument. In this program, since you need to have separate arrays for the character and numeric variables, you could have just as easily used the standard period and blank to represent missing values. Because of the END= option in the SET statement, the program outputs the counts when the last observation is processed from the data set TEST_MISS. Output from this program is shown below:

```
                    Count of Missing Values

                    2 Numeric and 1 Character values missing
```

Function: RANK

Purpose: To obtain the relative position of the ASCII (or EBCDIC) characters. This can be useful if you want to associate each character with a number so that an ARRAY subscript can point to a specific character.

Syntax: **RANK(*letter*)**

letter can be a string literal or a SAS character variable. If the literal or variable contains more than one character, the RANK function returns the collating sequence of the first character in the string.

Examples

For these examples, `STRING1 = "A"` and `STRING2 = "XYZ"`

Function	Returns
RANK(STRING1)	65
RANK(STRING2)	88
RANK("X")	88
RANK("a")	97

Program 1.52: Using the collating sequence to convert plain text to Morse Code

```
***Primary function: RANK
***Other functions: LENGTH, UPCASE, SUBSTR;

DATA _NULL_;
  ARRAY DOT_DASH[26] $ 4 _TEMPORARY_ ('.-' '-...' '-.-.' '-..' '.'
                                      '..-.' '--.' '....' '..' '.---'
                                      '-.-' '.-..' '--' '-.' '---' '.--.'
                                      '--.-' '.-.' '...' '-' '..-'
                                      '...-' '.--' '-..-' '-.--' '--..');
   INPUT @1 STRING $80.;
   FILE PRINT;
      TITLE "Morse Code Conversion Using the RANK Function";
   DO I = 1 TO LENGTH(STRING);
      LETTER = UPCASE(SUBSTR(STRING,I,1));
      IF LETTER = ' ' THEN PUT LETTER @;
      ELSE  DO;
         NUM = RANK(LETTER) - 64;
         PUT DOT_DASH[NUM] ' ' @;
      END;
   END;
   PUT;
DATALINES;
This is a test SOS
Now is the time for all good men
;
```

Explanation

The RANK function returns a value of 65 for an uppercase A, a value of 66 for a B, and so forth (in the ASCII character set). If you subtract 64 from the RANK value of the letters A to Z, you will get the numbers 1 to 26. Each element in the temporary array is the Morse code equivalent of the 26 letters of the alphabet.

The DO loop starts from 1 to the LENGTH of STRING. Each letter is converted to uppercase and its order in the alphabet is returned by the expression `RANK(LETTER) - 64`. This value is then used as the subscript in the DOT_DASH array and the appropriate series of dots and dashes is written to the output screen. As an "exercise for the reader," this problem can also be solved in an elegant manner using a user-defined format mapping the letters of the alphabet to the Morse equivalents. The output from this program is shown below:

```
Morse Code Conversion Using the RANK Function
-  ....  ..  ...    ..  ...    .-    -  .  ...  -    ...  ---  ...
-.  ---  .--    ..  ...    -  ....  .    -  ..  --  .    ..-.  ---  .-.
   .-  .-..  .-..    -..  ---  ---  -..    --  .  -.
```

Function: REPEAT

Purpose: To make multiple copies of a string.

Syntax: **REPEAT(*character-value, n*)**

character-value is any SAS character expression.

n is the number of repetitions. The result of this function is the original string plus *n* repetitions. Thus if *n* equals 1, the result will be two copies of the original string in the result. If you do not declare the length of the character variable holding the result of the REPEAT function, it will default to 200.

Examples

For these examples, `STRING = "ABC"`

Function	Returns
`REPEAT(STRING,1)`	`"ABCABC"`
`REPEAT("HELLO ",3)`	`"HELLO HELLO HELLO HELLO"`
`REPEAT("*",5)`	`"******"`

Program 1.53: Using the REPEAT function to underline output values

```
***Featured Function: REPEAT;

DATA _NULL_;
   FILE PRINT;
   TITLE "Demonstrating the REPEAT Function";
   LENGTH DASH $ 50;
   INPUT STRING $50.;
   IF _N_ = 1 THEN PUT 50*"*";
   DASH = REPEAT("-",LENGTH(STRING) - 1);
   PUT STRING / DASH;
DATALINES;
Short line
This is a longer line
Bye
;
```

Explanation

I must admit, I had a hard time coming up with a reasonable program to demonstrate the REPEAT function. The program above underlines each string with the same number of dashes as there are characters in the string. Since you want the line of dashes to be the same length as the string, you subtract one from the length, remembering that the REPEAT function results in $n + 1$ copies of the original string (the original plus n repetitions).

The two important points to remember when using the REPEAT function are: always make sure you have defined a length for the resulting character variable, and the result of the REPEAT function is $n + 1$ repetitions of the original string. The output from the program above is shown below:

```
Demonstrating the REPEAT Function
************************************************
Short line
----------
This is a longer line
--------------------
Bye
---
```

Function: REVERSE

Purpose: To reverse the order of text of a character value.

Syntax: **REVERSE(*character-value*)**

character-value is any SAS character expression.

Examples

For these examples STRING1 = "ABCDE" and STRING2 = "XYZ "

Function	Returns
REVERSE(STRING1)	"EDCBA"
REVERSE(STRING2)	" ZYX"
REVERSE("1234")	"4321"

Program 1.54: Using the REVERSE function to create backwards writing

```
***Primary function: REVERSE;

DATA BACKWARDS;
   INPUT @1 STRING $CHAR10.;
   GNIRTS = REVERSE(STRING);
DATALINES;
Ron Cody
   XYZ
ABCDEFG
         x
1234567890
;
PROC PRINT DATA=BACKWARDS NOOBS;
   TITLE "Listing of Data Set BACKWARDS";
RUN;
```

Explanation

It is important to realize that if you don't specify the length of the result, it will be the same length as the argument of the REVERSE function. Also, if there were trailing blanks in the original string, there will be leading blanks in the reversed string. Look specifically at the last two observations in the listing below to see that this is the case.

```
                    Listing of Data Set BACKWARDS

                      STRING        GNIRTS

                    Ron Cody        ydoC noR
                       XYZ            ZYX
                    ABCDEFG          GFEDCBA
                             x    x
                    1234567890    0987654321
```