

Introduction to Functions & Arrays

Ken Currier, Norfolk Southern Corporation, Roanoke, VA

Introduction.

When first exploring the world of SAS functions, you probably came across the following verbiage:

"A SAS function returns a value from a computation or system manipulation that requires zero or more arguments... The syntax of a function consists of a function name and function arguments; the arguments must be enclosed in parentheses. For example, use the following form for a function with two arguments:

function-name(argument-1,argument-2)

Parentheses must always be supplied even if the function does not require an argument."

Great definition!! But what does it mean? What I found, once I got past definition shock, was that SAS functions provide quick and easy ways to extract, analyze, compartmentalize, and manipulate data in any number of situations. Your imagination is the only boundary that you will encounter in applying SAS functions. This paper will present a few of the ways that you can use SAS functions in your day-to-day programming and processing requirements.

We're Not In Kansas Anymore, Toto

These words uttered by Dorothy in The Wizard of OZ, typify the current state of information requirements in the business world. Information requirements are constantly changing, but information systems typically are not. The corporate information platform is often a patchwork of legacy and generic systems (i.e., general purpose systems such as general ledger, accounts payable, etc. that are marketed by a company to a large number of business types). This means that we often have to develop alternative ways to obtain the needed information.

SAS functions are a great tool to assist you in extracting, creating and presenting this new information. Before we get too deep into the syntax and possible uses of SAS functions, let's take a brief tour of a company named "Humongus, Inc."

Humongus Inc.

Your name - Ida Genius. The company - Humongus, Inc. You've just arrived at your new office, fully armed with a complete knowledge of SAS functions and a large "S" emblazoned across your chest. It's a quarter to five and in rushes Clu Less (the top production manager in the world - 1st runner up in the universe) with a perplexed look across his face. "I assumed they knew how I wanted them to key the machine numbers into the database", he blurts out. You quickly assess the situation to determine what has transpired.

Clu had set up a database to enter critical information about all the company's production machinery. The machine number was set up as a 10 byte alpha-numeric field. The first 2 bytes was the acquisition year of the machine, and the next 5-7 bytes were the machine number. The keypunch work had been divided among four people for input. Clu had wanted the machine number to be entered as a continuous alpha-numeric field. Unfortunately, three of the keypunch types thought it would be a good idea to set off the acquisition year bytes from the machine number. One keypunch operator had inserted a dash after the year, another a period, and another an underline mark.

"How am I going to fix this mess", moans Clu. "There are over 250,000 machines in this database. Can you help me Ida?" Quickly you remember the COMPRESS function. "Gee, I don't know Clu, that's a tough one", you mutter

with a feigned look of horror on your face. "I'll probably have to work through the night to undo that big of a mess. I needed to mow my lawn and trim the hedges at my house tonight. Do you think you could do that for me while I work on this problem for you?" Clu ponders for a moment about how long it's going to take him to correct the database, and then reluctantly agrees to accomplish the small chores you had scheduled for yourself that evening. You quickly find out from Clu where the database is located. After Clu leaves your office to attend to the chores at your house, you bring the database into SAS and you type in the COMPRESS function program step:

```
MACHNO = COMPRESS(MACHNO,'-._');
```

Voila! Another hard night's work accomplished. A quick scan of the newspaper to find out where that great new blockbuster movie is playing and a call to make reservations at your favorite restaurant and you're on your way to a great evening of food and entertainment.

The next morning as you work at your desk, you notice a shadow tentatively creeps across your computer screen. You look up to see Hes Nervous standing in your doorway. "Is there something I can do for you Hes", you intuitively ask. "Well I'm ... sorta in charge ... of ... the employee database for ... uh ... H.H. (Head Honcho)", stammers Hes, "and ... uh ... she's ... not happy ... with the ... uh ... way ... my company picnic ... uh ... mailing labels look". Hes hesitantly places the labels in front of you. Sharp as a tack, you surmise Hes' problem with a quick glance of the mailing labels.

John Smith
3124 Somewhere Drive
Elsewhere VA 24044

EbenezeriteScrooge
1435 Overthere Road
Onto VA 24000

"Can ... uh ... you ... uh ... maybe ... help me", squeaks Hes. "I'm afraid I'll have to have

authorization from S.L. (Supreme Leader) to work on this one", you reply, "but I'll see what I can do". Two weeks later you have your authorization from S. L. to work on the project. With great subtlety and cunning you use the TRIM function to create new datafields called TOTNAME and CITYST as follows:

```
TOTNAME = TRIM(FNAME) || ' ' || LNAME;  
CITYST = TRIM(CITY) || ' ' || STATE;
```

Your task complete, you hand the new labels to H.H., who is absolutely thrilled with the new look.

John Smith
3124 Somewhere Drive
Elsewhere, VA 24044

Ebenezerite Scrooge
1435 Overthere Road
Onto, VA 24000

The next morning your phone rings. It's H.H. She would like to see you in her office. As you enter the palacious surroundings, H.H. announces that she's been thinking. An uncontrollable shiver runs down your spine. "I would like the state to be spelled out, since that looks so much more formal -- could you work some more of your magic", she asks? "Well, I'll see what I can do, but it will have to be cleared through S.L.", you reply. H.H. looks up from the 3 foot stack of papers on her desk and replies, "I would appreciate it. Poor Hes had a complete nervous breakdown when I told him what I wanted. As you can see the paper's getting pretty deep in here."

Two weeks later you have your approval from S.L. to work on the project. You quickly redefine the CITYST field to be:

```
CITYST = TRIM(CITY) || ' ' ||  
         ZIPNAMEL(ZIPCODE);
```

Mission accomplished you deliver the new labels to H.H. who is absolutely ecstatic.

John Smith
3124 Somewhere Drive
Elsewhere, Virginia 24044

Ebenezerite Scrooge
1435 Overthere Road
Onto, Virginia 24000

You no sooner get back to your office when in walks Clu Less. "Ida, can you help me", he begs? "S.D. (Slave Driver) wants a list of all our pile drivers and I just don't know how to get that out of the production machinery database". A worried look comes over your face. "Is there anyway to use the machine number to identify the pile drivers", you ask. "Well, not exactly", replies Clu, "the first three bytes of the machine number after the acquisition year identify the machines belonging to the pile drivers, tampers, and compactor machine group. But the machine description field would show that it was a pile driver." You wrinkle your brow and stare into space. "What is the three byte code that is associated with that machine group", you ask? Clu stammers, "It ... It's group code 164 ... but S.D. only wants a list of the pile drivers". Deep in thought you ask, "is the machine description entered in lower and uppercase"? "Well, yes...", Clu replies, "but I don't see what that has to do with the problem". You look at Clu and instinctively know he's panicking. He wanted this encounter to be a simple "no problem" experience. You poise your head and confidently reply, "it will take some time, but I think I can do the report for you". Clu gratefully tells you he will always be in your debt.

Once Clu leaves, you access the database and craftily write the following selection statement:

```
DATA work.file;SET machfile;
IF SUBSTR(MACHNO,3,3) = '164'
AND
INDEX(UPCASE(MACHDESC),'PILE DRIV')
> 0;
```

You arrive at work early the next morning, carefully splash some water on your face, ruffle your hair, and stagger into Clu's office. You

hand the report to Clu and exclaim, "you just wouldn't believe what I had to do to get that report out of your production machinery database". Clu is in complete awe of your dedication and hard work.

Back to Reality.

Hopefully this tongue-in-cheek tour of Humongus, Inc. has been both entertaining and informative. A key theme you should have obtained from this tour is that SAS functions provide a variety of shortcuts to assist you in writing programs and meeting day-to-day information requests that you receive.

Please note, Humongus is a fictional company and that any similarities that you may have noted between Humongus and other companies is purely coincidental.

SAS Functions.

Some of the specific items you need to remember about SAS functions are:

- The syntax consists of a function name and function argument(s).
- The arguments always must be enclosed by parentheses. Parentheses must be used even if an argument is not required.
- The function must be used as part of a SAS statement.
- Functions can be embedded within other functions if the resulting values are compatible (i.e., numeric or character).
- Typically when using a function that requires more than one argument, the arguments must be separated by commas. However, if the arguments of a function are all variables and are in a sequence (for example, Y1 through Y25, or X Y Z), the function can be written in one of two abbreviated argument list forms preceded by OF as shown in the syntax below:

FUNCTION(OF *variable1-variablen*)
FUNCTION(OF *variable variable*
variable ...)

Arithmetic & Truncation Functions.

Arithmetic and truncation functions provide a wide variety of tasks when dealing with numeric data. They can be useful in performing calculations (e.g., ROUND, SUM) or testing parameters of data (e.g., ABS).

ABS: returns the absolute value

Syntax: ABS(*argument*)

This function will return a positive number of the same magnitude as the original value of the argument. Example:

```
DATA;  
X=ABS(4.1);  
Y=ABS(-4.1);  
PUT X Y;
```

The value returned for both X and Y is 4.1. This function is useful when you are testing a number or calculated variance for its magnitude only, such as + or - \$100.

MAX: returns the largest value

Syntax: MAX(*argument,argument,...*)

This function returns the largest value among the nonmissing values of the arguments. At least two nonmissing values are required. Examples:

```
X=MAX(1,7,.);  
X=MAX(25,-8,22,-7);  
X=MAX(4,.,3);
```

the values returned are 7, 25, and 4 respectively. This function permits the use of the abbreviated argument list preceded by OF.

MOD: calculates the remainder

Syntax: MOD(*argument1,argument2*)

This function returns the remainder when the quotient of argument1 is divided by argument2 is calculated. Example:

```
Y=MOD(12,7);
```

gives a value of 7.

ROUND: rounds a value to nearest roundoff unit

Syntax: ROUND(*argument,roundoffunit*)

This function rounds a value to the nearest roundoff unit. Example:

```
X=ROUND(1017.45687,.1);
```

gives X a value of 1017.5.

SUM: calculates the sum of the arguments

Syntax: SUM(*argument,argument, ...*)

This function returns the sum of the arguments. Two or more arguments are required. Example:

```
X=SUM(4,5,6,7,8,9);  
returns the value 39.
```

You can also use an abbreviated argument list preceded by OF for this function. Example:

```
Y=SUM(OF MONTH1-MONTH12);
```

Character Functions.

This set of functions is helpful in those situations where (1) you are trying to normalize character fields within existing databases (e.g., COMPRESS, LEFT, RIGHT, TRANSLATE), or (2) perform specific data extractions on data fields that were not intended to be primary indexes (e.g., INDEX, SUBSTR).

Compress: removes specific characters from a character expression

Syntax: COMPRESS(*argument*)
COMPRESS(*argument1, argument2*)

If only one argument is specified this function returns the character value after removing all the blanks. If two arguments are specified this function returns the character value after removing all the characters specified in the second argument. Examples:

```
X = 'INTO THE SUN';
Y=COMPRESS(X);
  yields Y as 'INTOTHESUN';
```

```
X='#(555)846-1322';
Y=COMPRESS(X,'#-');
  yields Y as '5558461322';
```

INDEX: searches the first argument for the character specified by the second
Syntax: INDEX(argument1,argument2)

This function searches argument1 from left to right, for the first occurrence of the string specified in the second argument, and then returns the position of the string's first character. If the string is not found then INDEX returns a value of 0. Example:

```
A='ABC.DEF (X=Y)';
B='X=Y';
X=INDEX(A,B)
```

The value of X is 10, the starting position in A of the string specified by B, X=Y.

LEFT: left aligns a SAS character expression
Syntax: LEFT(argument)

The left function returns the argument with leading blanks moved to the end of the value.
Example:

```
X=' CELEBRITY';
Y=LEFT(X);
```

gives X the value 'CELEBRITY '.

LENGTH: returns the length of an argument
Syntax: LENGTH(argument)

This function returns the length of the character expression specified by the argument. The result is numeric and gives the position of the right-most nonblank character in the argument.
Example:

```
LNGLTH=LENGTH('FOUR SCORE');
```

results in LNGLTH having a value of 10.

REPEAT: repeats a character expression
Syntax: REPEAT(argument1,n)

This function returns a character value consisting of the first argument repeated *n* times. Thus the result will be the first argument appearing *n+1* times. Example:

```
Z=REPEAT('SAS',3);
```

gives the value SASSASSASSAS.

SCAN: returns a given word from a character expression
Syntax: SCAN(argument1,n,delimiters)
SCAN(argument1,n)

This function separates the first argument, a character expression, into "words" and returns the *n*th word. The delimiters are characters defined as separators between words. If the delimiters argument is not specified, the following characters are default delimiters:

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

The statements

```
ARG='ABC.DEF(X=Y)';
WORD=SCAN(ARG,3);
```

give WORD the value X=Y, the third "word" in the character expression, ARG.
Leading delimiters before the first word have no effect. If there are two or more contiguous delimiters, they are treated as one. If there are fewer than *n* words in the first argument, SCAN returns a blank value.

SUBSTR: extracts a substring from an argument or replaces character value contents.

Syntax for Substring Function:

SUBSTR(argument1,position,n)

When the SUBSTR function is used on the right side of an equal sign, it extracts from *argument1* a substring that is *n* characters long, beginning with the character specified by *position*. If *n* is omitted, the substring consists of the remainder of argument, beginning with the character specified by *position*. Examples:

```
DATE='06OCT1996';
MONTH=SUBSTR(DATE,3,3);
DAY=SUBSTR(DATE,1,2);
YEAR=SUBSTR(DATE,6,4);
MONYR=SUBSTR(DATE,3);
```

Syntax for Character Value Insertion:

SUBSTR(argument1,position,n) = X

When the SUBSTR is used on the left side of an equal sign, it serves as a pseudo-variable function, replacing the contents of a character value. The value of the variable or constant on the right side of the equal sign is placed into the character variable specified by the first argument, starting with the character specified by position, and replacing the number of characters specified by *n*. Notice that the first argument must be a character variable. This restriction is an exception and applies only when SUBSTR is used as a pseudo-variable function. Example:

```
Y='UNDO';
SUBSTR(Y,1,2)='RE';
```

gives Y the value 'REDO'.

If the *n* argument is omitted, characters in the first argument are replaced from *position* to the end of the first argument.

TRIM: removes trailing blanks from character expression

Syntax: **TRIM(argument)**

This function returns the argument with trailing blanks removed. This function is especially useful when concatenating, since concatenation does not remove trailing blanks. Example:

```
FIRST='JOHN';
LAST='SMITH';
NAME=TRIM(FIRST) || ' ' || TRIM(LAST);
```

will give NAME the value 'JOHN SMITH'.

UPCASE: converts all characters in the argument to uppercase

Syntax: **UPCASE(argument)**

State & Zip Code Functions.

This group of functions provides a wide variety of useful calculations involving state and zip code information. Some of these calculations include: converting zip codes to postal (e.g., AL) or FIPS (e.g., 01) code or state name (e.g., Alabama); converting FIPS codes to postal codes or state names; and converting postal codes to FIPS codes or state names.

FIPNAME: converts FIPS codes to state names in uppercase

Syntax: **FIPNAME(fips)**

This function takes a numeric FIPS code and returns the 20 character state name in uppercase. Example:

```
X=FIPNAME(01);
```

gives X the value 'ALABAMA'.

STNAME: converts state postal codes to state names in uppercase

Syntax: **STNAME(postalcode)**

This function takes a two character state postal code (enclosed in quotes) and returns the corresponding 20 character state name in uppercase. Example:

X=STNAME('AL');

gives X the value 'ALABAMA'.

ZIPNAME: converts ZIP codes to state names in uppercase

Syntax: **ZIPNAME(zipcode)**

This function takes a five character ZIP code (enclosed in quotes) and returns the corresponding 20 character state name in uppercase. Example:

X=ZIPNAME('35222');

gives X the value 'ALABAMA'.

Date & Time Functions.

This set of functions is helpful in any number of programming calculations that involve dates.

DATE: returns today's date as a SAS date value

Syntax: **DATE()**
TODAY()

DATEJUL: converts a Julian date to a SAS date value

Syntax: **DATEJUL(juliandate)**

DAY: returns the day of the month from a SAS date value

Syntax: **DAY(date)**

MDY: returns a SAS date value from month, day, and year

Syntax: **MDY(month,day,year)**

QTR: returns the quarter from a SAS date value or literal

Syntax: **QTR(date)**

Some examples of programming uses for the above functions are:

- You need to summarize material issue transactions for the past year on a quarterly basis. The date (variable name *TRANSDT*) is entered into the material system in Julian date form (*yyddd*). Your program could use a couple of date functions.

```
NEWDATE=DATEJUL(TRANSDT);
QUARTER=QTR(NEWDATE);
```

- You need to identify receivables that are 30 or more days overdue, but the due date in the system is in 3 separate data fields of *DAY*, *MONTH*, and *YEAR*.

```
DUEDATE=MDY(MONTH,DAY,YEAR);
TDAY=DATE();
DAYS DUE=TDAY - DUEDATE;
IF DAYS DUE > 29 THEN PUT .....
```

Financial Functions.

This set of functions is relatively new on the scene and can provide you with a number of financial analysis shortcuts. Some of the calculations accomplished by these functions include depreciation, net present values, and internal rates of return.

Depreciation Group.

The basic form for depreciation calculations can be broken into two groups. These are (1) declining balance depreciation, and (2) all other depreciation (e.g., straight-line and sum-of-the-years). Both groups will calculate both the current period's depreciation expense and the life-to-date depreciation reserve. All of the depreciation functions, except for the declining balance converting to straight-line group, permit the use of fractional periods (e.g., 1.25, 1.75, etc.).

Declining Balance Group

DACCDB: returns accumulated declining balance depreciation.

Syntax: **DACCDB(period,value,years,rate)**

DACCDBSL: returns the accumulated declining balance converting to straight-line depreciation.

Syntax: DACCDBSL(*period,value,years,rate*)

DEPDB: returns declining balance depreciation.

Syntax: DEPDB(*period,value,years,rate*)

DEPDBSL: returns declining balance converting to straight-line depreciation.

Syntax: DEPDBSL(*period,value,years,rate*)

Let's say that I want to calculate the current year's depreciation (*CYDEPR*) as well as the total depreciation taken to date (*LTDPR*) on a car that I bought 3 years ago (Jan. 1st for simplicity's sake) for \$18,000 with an expected life of 7 years, and that I chose to depreciate this car using a 200% declining balance rate (also known as double declining balance). The programming steps to accomplish this would include:

```
CYDEPR= ROUND(DEPDB(3,18000,7,2),.01);  
LTDPR=ROUND(DACCDB(3,18000,7,2),.01);
```

gives a value of \$2,623.91 for *CYDEPR* and \$11,440.23 for *LTDPR*.

Other Depreciation Group.

This group includes straight-line and sum-of-the-years digits depreciation method. The key difference between this group and the declining balance group is that you no longer need to provide the rate information (e.g., 200%).

DACCSL: returns accumulated straight-line depreciation.

Syntax: DACCSL(*period,value,years*)

DACCSYD: returns accumulated sum-of-years' digits depreciation.

Syntax: DACCSYD(*period,value,years*)

DEPSL: returns straight-line depreciation

Syntax: DEPSL(*period,value,years*)

DEPSYD: returns sum-of-years' digits depreciation.

Syntax: DEPSYD(*period,value,years*)

Using the information previously provided, if we used the sum-of-the-years digit method the programming steps would be:

```
CYDEPR=ROUND(DEPSYD(3,18000,7),.01);  
LTDPR=ROUND(DACCSYD(3,18000,7),.01);
```

gives a value of \$3,214.29 for *CYDEPR* and \$11,571.43 for *LTDPR*.

Internal Rate of Return Group.

This group calculates an internal rate of return for an anticipated stream of cashflows.

INTRR: returns internal rate of return as a fraction.

Syntax: INTRR(*period cash0,cash1,...*)

IRR: returns internal rate of return as a percentage.

Syntax: IRR(*period,cash0,cash1,...*)

For example, suppose someone presents you with the following proposal - loan me \$100 today and I will pay you \$45 a year for the next three years. You would like to know what your rate of return would be on your \$100 loan. Note that your *cash0* number will be negative, since you have a cash outflow.

```
ROR1=ROUND  
      (INTRR(1,-100,45,45,45).0001);  
ROR2=ROUND(IRR(1,-100,45,45,45).01);
```

ROR1 will return a value of 0.1665 (fraction) and ROR2 will return a value of 16.65 (percentage). Therefore, this proposal would yield a rate of return of 16.65%.

Net Present Value Group.

This group calculates the present value for a discounted stream of cash flows. Both of the

functions that are available return the same value, the only change between the two is the form that the rate is entered into the formula. In the NPV function the rate is entered as a percentage (e.g., 10.0) while in the NETPV the rate is entered as a fraction (e.g., .1).

NETPV: returns net present value with rate expressed as a fraction.

Syntax: `NETPV(rate,period,cash0,cash1,...)`

NPV: returns net present value with rate expressed as a percentage.

Syntax: `NPV(rate,period,cash0,cash1,...)`

For example, suppose you were presented with the option of (1) receiving \$260 cash now, or (2) receiving three payments of \$100 over the next three years. You know you could invest the money today in a 3-year note and get a 10% per annum return. To calculate which option you should take you would use the net present value function(s):

`NPV1=ROUND(NETPV(.1,1,100,100,100),.01);`

`NPV2=ROUND(NPV(10,1,100,100,100),.01);`

both give the value \$273.55. In this case, you would take the three payments of \$100.

SAS Arrays.

Often in your DATA step programming you are required to perform the same action on more than one variable. Array processing allows you to group these variables together and then perform the repetitive action using a DO loop. Basically, there are two steps within the array process.

- Arrange the variables in a temporary grouping called an array.
- Perform the desired actions for all elements of the array.

For example, you have created a SAS data set (stores95) with the following variables:

EXP Expense Code \$ 5.

STORE1	Store 1	8.2
STORE2	Store 2	8.2
STORE3	Store 3	8.2
STORE4	Store 4	8.2
STORE5	Store 5	8.2

To create an array you would type the following:

```
DATA work.file1;SET stores95;
ARRAY EXPEN{5} STORE1-STORE5;
```

You have created a 1 X 5 array. Let's say you want to see what would happen if your expenses in the stores increased 10%. If you did not use the array you could accomplish this task by the following program:

```
DATA work.file1;SET stores95;
STORE1 = ROUND(STORE1 * 1.1,.01);
STORE2 = ROUND(STORE2 * 1.1,.01);
STORE3 = ROUND(STORE3 * 1.1,.01);
STORE4 = ROUND(STORE4 * 1.1,.01);
STORE5 = ROUND(STORE5 * 1.1,.01);
...
```

or by using an array you could write:

```
DATA work.file1;SET stores95;
ARRAY EXPEN{5} STORE1-STORE5;
DO I=1 TO 5;
EXPEN{I} = ROUND(EXPEN{I}*1.1,.01);
END;
...
```

In this example, there is not much difference in programming steps to accomplish the task, but imagine the difference if you had 50 or 100 stores. Using an array, your programming steps would be the same, you would only change the dimension of the array and the parameters of the DO loop.

Within array processing you are provided additional functions that will help you in setting up your arrays and repetitive processing loops. These are:

DIM: automatically changes the upper bound of the index variable.

Syntax: DIM(*indexvariable*)

LBOUND: automatically changes the lower bound of the index variable.

Syntax: LBOUND(*indexvariable*)

HBOUND: automatically changes the upper bound of the index variable.

Syntax: HBOUND(*indexvariable*)

For example let's say you wanted to automatically change the lower and upper bounds in your DO loop in the above example:

```
DATA work.file1;SET stores95;
ARRAY EXPEN{5} STORE1-STORE5;
DO I=LBOUND(EXPEN) TO
    HBOUND(EXPEN);
EXPEN{I} = ROUND(EXPEN{I} * 1.1,.01);
END;
```

...

You could use the DIM function to change the upper bound in the DO loop as follows:

```
DATA work.file1;SET stores95;
ARRAY EXPEN{5} STORE1-STORE5;
DO I=1 TO DIM(EXPEN);
EXPEN{I} = ROUND(EXPEN{I} * 1.1,.01);
END;
```

...

Summary.

In this paper we have presented several of the SAS functions that can assist you in your programming and daily processing requirements. In addition, we have briefly covered the concept of an array, as well as those SAS functions that specifically deal with arrays. As you begin to incorporate SAS functions and arrays in your programming technique, you will find they are great tools in helping you meet a variety of programming and processing requirements.

References

SAS® User's Guide: Basics, Version 5 Edition, SAS Institute, Inc., Cary, NC

SAS® Language: Reference, Version 6, First Edition, SAS Institute, Inc., Cary, NC

SAS® Language and Procedures, Usage 2, Version 6, First Edition, SAS Institute, Inc., Cary, NC

Author

Ken Currier, CPA, MTX
Norfolk Southern Corporation
110 Franklin Road SE
Roanoke, VA 24042-0028
(540)981-5795

SAS® is a registered trademark of SAS Institute Inc., in the USA and other countries. ® indicates USA registration.