PharmaSUG 2022 - Paper AP-111

Functions (and More!) on CALL!

Richann Watson, DataRich Consulting;
Louise Hadden, Abt Associates Inc.

ABSTRACT

SAS® Functions have deservedly been the focus of many excellent SAS papers. SAS CALL subroutines, which rely on and collaborate with SAS functions, are less well known, although many SAS programmers use these subroutines frequently. This paper will look at numerous SAS functions and CALL subroutines, as well as explaining how both SAS functions and CALL subroutines work in practice.

There are many areas that SAS CALL subroutines cover including CAS (Cloud Analytic Services) specific functions, character functions, character string matching, combinatorial functions, date and time functions, external subroutines, macro functions, mathematical functions, sort functions, random number functions, special functions, variable control functions, and variable information functions.

While there are myriad SAS CALL subroutines and SAS functions, we plan to drill down on character function SAS CALL subroutines including string matching; macro, external and special subroutines; sort subroutines; random number generation subroutines; and variable control and information subroutines. We could go on and on about SAS CALL subroutines, but we are going to limit the SAS CALL subroutines discussed in this paper, excluding any environment specific SAS CALL subroutines such as those designated for use with CAS and TSO, as well as other redundant examples. We hope to demystify SAS CALL subroutines by demonstrating real world applications of specific SAS CALL subroutines, bringing some amazing capabilities to light.

INTRODUCTION

SAS functions act as subroutines, combining arguments, operations, and optional parameters, and return a value as a result. These values can be either numeric or character, and can be used to create variables in expressions, via the data step, where statements, the SAS macro language, within PROC REPORT, and in PROC SQL. Users can write their own functions using PROC FCMP or take advantage of the myriad SAS provided functions and CALL subroutines.

CALL subroutines are very similar to SAS functions, sometimes sharing the same name and providing the same or similar functionality. However, you cannot use CALL subroutines in expressions or assignment statements. CALL subroutines, as the name implies, begin with CALL statements, with the routine name appearing after “CALL”, followed by parenthetical argument(s). As with SAS functions, the number of required and optional arguments vary.

A number of SAS functions and their homonymous SAS CALL subroutines will be demonstrated throughout this paper to demonstrate the similarities and differences. Some CALL subroutines are designed to be used in conjunction with SAS functions, including two that we are demonstrating. SAS functions and CALL subroutines are used in many different platforms, but not all are universally available across systems. Users will note that the result of a SAS function (often an assigned variable) usually precedes an equal sign followed by the SAS function and arguments, while the result of a CALL routine is typically the first argument (with a few exceptions) following the opening parenthesis after the invocation of the CALL routine. This paper is appropriate for all users at all levels and in all industries. Examples were run in Base SAS version 9.4 Maintenance Release 7. Users of SAS on CAS platforms should consult the SAS documentation before using specific SAS functions and CALL subroutines, as the availability is variable. Data used for provided examples are discussed below.

SAMPLE DATA SET(S) FOR PAPER

Data Display 1 is a sample of the custom PETS data set that will be used throughout the paper to illustrate the various functions and CALL subroutines. Some modifications are made to enhance the demonstrated examples for specific CALL subroutines and will be detailed in the paper.

Row	NAME	NICKNAME	BREED	TYPE
1	Chewbaca Watson	Chewy	Maltese/Poodle	Dog
2	Loki Jack Watson	Loki or LoJack or Bunny	Bichon/Shih-Tzu	Dog
3	Pepper Watson	Pepper	Unknown	Rabbit
4	Peter Rabbit Watson	Petey	Netherland Dwarf	Rabbit
5	Easter Watson	Easter or EC	Netherland Dwarf	Rabbit
6	Bailey Watson	Bales	Maltese-Poodle	Dog
7	Digger Kier	Diggy	Maltese/Poodle	Dog
8	Kandi Kier	Kandi	Poodle/Yorkshire Terrier	Dog
9	TC Brucken	TC	Tortoishell	Cat
10	Bandit Brucken	Bandit	Domestic Shorthair	Cat

Row	WEIGHT (LB)	LENGTH (IN)	STATUS	DATE OF BIRTH	DEATH DATE	AGE
1	14	10	Deceased	2003-12-23	2018-01-19	14.08
2	12	14	Alive	2018-05-07		3.00
3	8	8	Deceased	1996	2008-11-28	12.91
4	2.5	5	Deceased	2000-02-27	2010-03-25	10.07
5	2	5	Deceased	2000-04-23	2013-01-16	12.73
6	10	14	Deceased	2002-08-08	2018-01-31	15.48
7	8	9	Deceased	2002-05-25	2011-12-14	9.54
8	10	12	Alive	2012-04-18		10.00
9	10		Deceased	1976	1992	16.00
10	10		Deceased	1988	2005	17.00

Row	MEDICAL CONDITIONS	FAVORITE TOY	FAVORITE THING TO DO
1	Auto immune, diabetic, bladder cancer	Red, green, blue cloth ball	Car shows, boating, ATVing, kayaking, being with his humans
2		Any toy he can destroy	Playing nerf ball in backyard with his daddy or curling up with mommy
3	Malocclusion (overgrown teeth)	Wood blocks	Running around the basement
4		Wood blocks	Getting into things he shouldn't
5		Wood blocks	Sitting and watching everything
6	Skin condition, congestive heart failure	Stuffed football	Liked to go "long" after his stuffed football and curl up in mommy's lap and chew on rawhides. His first toy was a blue zebra.
7	Skin allergies, heart attack		Loved chewing on chew bones
8		Chicken in barn	Running around and loves to wear clothes
9	Old age		Biting and scratching people
10	Kidney failure		Hiding

Data Display 1: Sample of Custom PETS Data Set

CHARACTER AND STRING-MATCHING SAS CALL SUBROUTINES

CALL CATS AND CATX SUBROUTINES

Most of us are familiar with the (fabulous feline) CAT functions. However, for this paper we are going to discuss CAT CALLs. Although there are five CAT functions (CAT, CATS, CATT, CATX and CATQ), there are only three CAT CALLs (CALL CATS, CALL CATT and CALL CATX). These CALL subroutines behave similarly to the CAT functions with the main difference being that in the CAT functions it returns a value that is used to populate another variable, while the CAT CALLs return the value in the first variable in the list of arguments.

As mentioned previously, the CAT functions return a value to populate another variable. Refer to SAS Program 1 for an example of how CATS function is implemented with the corresponding output found in Data Display 2. Notice that in the output, that a new variable STT is created, and this variable houses the concatenated values of STATUS, DEATH_DATE enclosed in parenthesis, and MEDICAL_CONDITIONS.

Even though some of the arguments had either a leading or trailing blank, those were removed because the CATS function strips both leading and trailing blanks.

```
data pet_stat_func;
   set PETS (keep = NAME STATUS DEATH_DATE MEDICAL_CONDITIONS);
   STT = cats(STATUS, " (", DEATH_DATE, "): ", MEDICAL_CONDITIONS);
run;
```

SAS Program 1: Use of CATS Function

Row	NAME	STATUS	DEATH_DATE	MEDICAL_CONDITIONS
1	Chewbaca Watson	Deceased	2018-01-19	Auto immune, diabetic, bladder cancer
2	Loki Jack Watson	Alive	N/A	
3	Pepper Watson	Deceased	2008-11-28	Malocclusion (overgrown teeth)
4	Peter Rabbit Watson	Deceased	2010-03-25	
5	Easter Watson	Deceased	2013-01-16	
6	Bailey Watson	Deceased	2018-01-31	Skin condition, congestive heart failure
7	Digger Kier	Deceased	2011-12-14	Skin allergies, heart attack
8	Kandi Kier	Alive	N/A	
9	TC Brucken	Deceased	1992	Old age
10	Bandit Brucken	Deceased	2005	Kidney failure

Row	STT
1	***Deceased: (2018-01-19):Auto immune, diabetic, bladder cancer***
2	***Alive: (N/A):***
3	***Deceased: (2008-11-28):Malocclusion (overgrown teeth)***
4	***Deceased: (2010-03-25):***
5	***Deceased: (2013-01-16):***
6	***Deceased: (2018-01-31):Skin condition, congestive heart failure***
7	***Deceased: (2011-12-14):Skin allergies, heart attack***
8	***Alive: (N/A):***
9	***Deceased: (1992):Old age***
10	***Deceased: (2005):Kidney failure***

Data Display 2: Output Produced from SAS Program 1

Unlike the functions the CALL subroutines will write the result to the first argument in the call. Something to note is that if the buffer for the first argument is not sufficient to capture the full concatenated value, it will produce a log message and truncate based on the length of the variable.

The syntax for CATS CALL is

CALL CATS(*result* <, *item-1*, …, *item-n*>);

Where result is the first argument that will be in the concatenated value. Result must be a character value. Item-1 through Item-n are the individual values that will be appended to the result. If Item-1 through Item-n are numeric they are converted to a character string using the BESTw. format.

Refer to SAS Program 2 for an illustration of CALL CATs. SAS Log 1 is a portion of the corresponding log indicating that the first argument was not long enough to contain the full value and so it is truncated. The log message also indicates which argument is causing the truncation issue. In this example it is the third argument. Because of this truncation the value for STATUS only contains the first 7 characters as seen in Data Display 3.

```
data pet_stat_call;
   set PETS (keep = NAME STATUS DEATH_DATE MEDICAL_CONDITIONS);
   call cats(STATUS, ": (", DEATH_DATE, "): ", MEDICAL_CONDITIONS);
run;
```

SAS Program 2: Use CALL CATS Causing Truncated Value

```
WARNING: In a call to the CATS routine, the first argument was not long enough to contain
         the concatenation of all the arguments.
         The correct result would contain 58 characters, but the actual result was
         truncated to 8 character(s). The following note indicates the left-most argument
         that caused truncation.
NOTE: Argument 2 to function CATS('Deceased',' (','2018-01-19','): ','Auto immune,'[12 of
      62 characters shown]) at line 31 column 9
      is invalid.
```

SAS Log 1: Log Message Produced from SAS Program 2

NAME	STATUS	DEATH_DATE	MEDICAL_CONDITIONS
Chewbaca Watson	***Deceased***	2018-01-19	Auto immune, diabetic, bladder cancer
Loki Jack Watson	***Alive: (***	N/A	
Pepper Watson	***Deceased***	2008-11-28	Malocclusion (overgrown teeth)
Peter Rabbit Watson	***Deceased***	2010-03-25	
Easter Watson	***Deceased***	2013-01-16	
Bailey Watson	***Deceased***	2018-01-31	Skin condition, congestive heart failure
Digger Kier	***Deceased***	2011-12-14	Skin allergies, heart attack
Kandi Kier	***Alive: (***	N/A	
TC Brucken	***Deceased***	1992	Old age
Bandit Brucken	***Deceased***	2005	Kidney failure

Data Display 3: Output Produced from SAS Program 2

To circumvent this issue and keep the variables in the same order, you can use a RETAIN and LENGTH statement prior to the set statement to put the variables in the program data vector. SAS Program 3 is the same as SAS Program 2 with the only difference being the RETAIN and LENGTH statements prior to the SET statement. This allows the variable order to be maintained as well as setting a length that will capture the entire concatenated value as seen in Data Display 4. Something to keep in mind is that when using CALL subroutines, if the first argument has a format applied to it then this may not produce the expected results. For example, if STATUS had a format of $8, CALL CATS still produces the concatenated value but because of the format it would still only show the first 8 characters.

```
data pet_stat_call2;
   retain NAME STATUS;
   length STATUS $200;
   set PETS (keep = NAME STATUS DEATH_DATE MEDICAL_CONDITIONS);
   call cats(STATUS, ": (", DEATH_DATE, "): ", MEDICAL_CONDITIONS);
run;
```

SAS Program 3: Use CALL CATS with No Truncation and Maintaining Variable Order

Row	NAME	STATUS
1	Chewbaca Watson	***Deceased: (2018-01-19):Auto immune, diabetic, bladder cancer***
2	Loki Jack Watson	***Alive: (N/A):***
3	Pepper Watson	***Deceased: (2008-11-28):Malocclusion (overgrown teeth)***
4	Peter Rabbit Watson	***Deceased: (2010-03-25):***
5	Easter Watson	***Deceased: (2013-01-16):***
6	Bailey Watson	***Deceased: (2018-01-31):Skin condition, congestive heart failure***
7	Digger Kier	***Deceased: (2011-12-14):Skin allergies, heart attack***
8	Kandi Kier	***Alive: (N/A):***
9	TC Brucken	***Deceased: (1992):Old age***
10	Bandit Brucken	***Deceased: (2005):Kidney failure***

Row	DEATH_DATE	MEDICAL_CONDITIONS
1	2018-01-19	Auto immune, diabetic, bladder cancer
2	N/A	
3	2008-11-28	Malocclusion (overgrown teeth)
4	2010-03-25	
5	2013-01-16	
6	2018-01-31	Skin condition, congestive heart failure
7	2011-12-14	Skin allergies, heart attack
8	N/A	
9	1992	Old age
10	2005	Kidney failure

Data Display 4: Output Produced from SAS Program 3

CATT and CATX functions have similar functionality to the CATS function, with the difference being that that CATT strips trailing blanks from the arguments while CATS strips leading and trailing blanks. CATX strips leading and trailing arguments and also allows you to specify what kind of delimiter to use between each argument instead of not having any delimiter as is done with CATS. The corresponding CALL subroutines also behave in a similar manner with some exceptions.

In the CATS example, note that the string arguments had spaces but those were stripped. In addition, when there was no MEDICAL_CONDITIONS, a semicolon was still added after the (DEATH_DATE). If you need to keep the spaces and want only the semicolon to be added if there is a value, then you can use CATX function as shown in SAS Program 4. With the CATX function if an argument is missing then the delimiter is not added to the concatenated value. The delimiter is only added when there is a non-missing value for the argument.

```
data pet_stat_func_x;
   set PETS (keep = NAME STATUS DEATH_DATE MEDICAL_CONDITIONS);
   STT = catx(": ", STATUS, cats("(", DEATH_DATE, ")"), MEDICAL_CONDITIONS);
run;
```

SAS Program 4: Use of CATX Function

While the CATX CALL routine behaves similarly to CATX function, there are some key differences. With the CAT functions, they can be nested as shown in SAS Program 4. However, nesting is not allowed in the CALL routine. In order to produce something similar to what was produced with the CATX function but have the value stored in the STATUS variable, you can nest a CATS function within the CATX CALL (SAS Program 5).

```
data pet_stat_call_x;
   retain NAME STATUS;
   length STATUS $200;
   set PETS (keep = NAME STATUS DEATH_DATE MEDICAL_CONDITIONS);
   call catx(": ", STATUS, cats("(", DEATH_DATE, ")"), MEDICAL_CONDITIONS);
run;
```

SAS Program 5: Use of CATX CALL routine with Nested CATS Function

In addition, with CATX function, the second argument can be either numeric or character, but with CALL CATX routine, the second argument must be a character. If AGE (numeric variable) is concatenated with DATE_OF_BIRTH (character variable), using CALL CATX as seen in SAS Program 6, it produces a warning in the log message (SAS Log 2).

```
data pet_call_call_x_inv;
   set PETS (keep = NAME AGE DATE_OF_BIRTH);
   call catx(": ", AGE, DATE_OF_BIRTH);
run;
```

SAS Program 6: Use of CATX CALL routine with Invalid Second Argument

```
WARNING 716-185: Argument #2 is a numeric variable, while a character variable must be
                 passed to the CATX subroutine call in order for the variable to be
                 updated.
```

SAS Log 2: Log Message Produced from SAS Program 5

CALL MISSING ROUTINE

When cleaning data, you undoubtedly have to deal with missing values. Sometimes you may know the data type for a particular variable and can use the proper syntax for identifying records with missing values. For example, character variables with missing values are denoted as a blank space, such as ‘ ’ while numeric variables with missing values use a period. You have probably used the logic illustrated in SAS Program 7.

```
data pets_calc_age;
   set pets (keep = NAME BIRTHDT DEATH_DATE);
   format DEATHDT date9.;
   length AGEATDTH $10;
   if BIRTHDT ^= . and DEATH_DATE ^= '' then do;
      if length(DEATH_DATE) = 10 then DEATHDT = input(DEATH_DATE, yymmdd10.);
      else DEATHDT = mdy(12, 31, input(DEATH_DATE, best.));
      AGE_CALC = round(yrdif(BIRTHDT, DEATHDT, 'age'), 1);
      AGEATDTH = catx(' ', round(yrdif(BIRTHDT, DEATHDT, 'age'), 1), 'YEARS');
   end;
   else do;
      DEATHDT = .;
      AGE_CALC = .;
      AGEATDTH = ' ';
   end;
run;
```

SAS Program 7: Using VAR = . or VAR = ‘’ Syntax

Notice that SAS Program 7 checks for non-missing values for birth date and death date before any further processing is done and if one of them is missing, then the variables are initialized to missing. While this approach is fine for a few variables, it can get a bit cumbersome for a lot of variables or if you don’t know the data types.

There are three missing functions: MISSING, NMISS and CMISS. The following is the syntax for each.

MISSING(*numeric-expression* | *character-expression*)

NMISS(*argument-1* <, *argument-2*,…>)

CMISS(*argument-1* <, *argument-2*,…>)

For MISSING, either a numeric or character expression must be provided, where the expression is a numeric/character constant, variable or expression. For NMISS and CMISS, at least one argument-n is required, where argument-n is a numeric/character constant, variable or expression. All three functions evaluate the arguments to see if they have missing values. MISSING only allows for one argument so it returns a 0 or 1. 0 if the expression is missing and 1 if it is non-missing. NMISS and CMISS allow for multiple arguments, so it returns a 0 or non-zero. If all the arguments are non-missing then it returns a zero, otherwise it returns the number of missing arguments. The difference between NMISS and CMISS is that NMISS requires all arguments to be numeric while CMISS can have a mix of character and numeric values. Thus, if you need to check more than one variable for missing values or if you are unsure of the data type, CMISS can be used since it allows for both data types. Note that the functions do not change the value of the argument(s).

In addition, to the missing functions, there is a CALL MISSING routine. The CALL routine does change the values of the arguments specified. It assigns a missing value to each argument. It assigns a numeric missing value (.) to the numeric variables in the list and a character missing value (‘ ’) to the character variables in the list. The syntax for CALL MISSING is

CALL MISSING(*variable-name-1* <, *variable-name-2*, ...>);

Using these functions and CALL routing, SAS Program 7 can be rewritten as shown in SAS Program 8. SAS Program 8 utilizes the missing function, MISSING, and the CALL MISSING routine. It also shows an alternative approach using CMISS to determine the number of missing arguments. Keep in mind that in this example BIRTHDT is numeric and DEATH_DATE is character, so the CMISS is used because it allows for a mix of data types. The values of BIRTHDT and DEATH_DATE are not overwritten in either code. However, DEATHDT, AGE_CALC and AGEATDTH are all initialized to missing using the CALL MISSING routine.

```
data pets_calc_age_2a;
   set pets (keep = NAME BIRTHDT DEATH_DATE);
   format DEATHDT date9.;
   length AGEATDTH $10;
   if ^missing(BIRTHDT) and ^missing(DEATH_DATE) then do;
      if length(DEATH_DATE) = 10 then DEATHDT = input(DEATH_DATE, yymmdd10.);
      else DEATHDT = mdy(12, 31, input(DEATH_DATE, best.));
      AGE_CALC = round(yrdif(BIRTHDT, DEATHDT, 'age'), 1);
      AGEATDTH = catx(' ', AGE_CALC, 'YEARS');
   end;
   else call missing(DEATHDT, AGE_CALC, AGEATDTH);
run;

OR

data pets_calc_age_2b;
   set pets (keep = NAME BIRTHDT DEATH_DATE);
   format DEATHDT date9.;
   length AGEATDTH $10;
   if cmiss(BIRTHDT, DEATH_DATE) = 0 then do;
      if length(DEATH_DATE) = 10 then DEATHDT = input(DEATH_DATE, yymmdd10.);
      else DEATHDT = mdy(12, 31, input(DEATH_DATE, best.));
      AGE_CALC = round(yrdif(BIRTHDT, DEATHDT, 'age'), 1);
      AGEATDTH = catx(' ', AGE_CALC, 'YEARS');
   end;
   else call missing(DEATHDT, AGE_CALC, AGEATDTH);
run;
```

SAS Program 8: Using Missing Functions and CALL MISSING Routine

Another feature of CALL MISSING is that you can use the 'of' with the ':' operator, if you know that there are number of variables with the same prefix. The 'else call missing' statement SAS Program 8 could be written as

```
call missing(DEATHDT, of AGE:);
```

All these approaches produce the same data set.

NAME	DEATH_DATE	BIRTHDT	DEATHDT	AGEATDTH	AGE_CALC
Chewbaca Watson	2018-01-19	23DEC2003	19JAN2018	14 YEARS	14
Loki Jack Watson		07MAY2018	-		-
Pepper Watson	2008-11-28	01JAN1996	28NOV2008	13 YEARS	13
Peter Rabbit Watson	2010-03-25	27FEB2000	25MAR2010	10 YEARS	10
Easter Watson	2013-01-16	23APR2000	16JAN2013	13 YEARS	13
Bailey Watson	2018-01-31	08AUG2002	31JAN2018	15 YEARS	15
Digger Kier	2011-12-14	25MAY2002	14DEC2011	10 YEARS	10
Kandi Kier		18APR2012	-		-
TC Brucken	1992	01JAN1976	31DEC1992	17 YEARS	17
Bandit Brucken	2005	01JAN1988	31DEC2005	18 YEARS	18

Data Display 5: Output Produced from SAS Program 7 and SAS Program 8

CALL PRXCHANGE ROUTINE

Within SAS you can Perl-regular expression (PRX) functions and CALL subroutines to help with pattern-matching. The PRX functions and CALL subroutines allow you to search for a pattern, extract substrings, and search and replace within a text. There are five PRX functions and five PRX CALL subroutines. However, for the purpose of this paper only PRXCHANGE and PRXPARSE are discussed.

Using our PETS data, each pet is assigned a unique three-digit ID and mini narrative is created that consists of the date of birth and medical conditions (Data Display 6).

PETID	NARRATIVE
001	Pet ID number 001: date of birth 2003-12-23. Had the following conditions: Auto immune, diabetic, bladder cancer
002	Pet ID number 002: date of birth 2018-05-07. Had the following conditions: N/A
003	Pet ID number 003: date of birth 1996. Had the following conditions: Malocclusion (overgrown teeth)
004	Pet ID number 004: date of birth 2000-02-27. Had the following conditions: N/A
005	Pet ID number 005: date of birth 2000-04-23. Had the following conditions: N/A
006	Pet ID number 006: date of birth 2002-08-08. Had the following conditions: Skin condition, congestive heart failure
007	Pet ID number 007: date of birth 2002-05-25. Had the following conditions: Skin allergies, heart attack
008	Pet ID number 008: date of birth 2012-04-18. Had the following conditions: N/A
009	Pet ID number 009: date of birth 1976. Had the following conditions: Old age
010	Pet ID number 010: date of birth 1988. Had the following conditions: Kidney failure

Data Display 6: Mini Narratives for PETS

For illustration purposes, the PETID and date of birth need to be masked so that the pet cannot be identified. In other words, we want to change all the PETID to a generic value such as 'XXX' and the date of births to something like 'DDMONYYYY'. This can be achieved using PRXCHANGE.

The syntax for PRXPARSE and PRXCHANGE are:

regular-expression-id = PRXPARSE(*perl-regular-expression*)

PRXCHANGE(*regular-expression-id* | *perl-regular-expression*, *times*, *source*)

The regular-expression-id is a numeric variable that has a pattern identifier that will be used in PRXCHANGE. With perl-regular-expression there are a several characters that are used to indicate a specific behavior. These are metacharacters and they can help to build search patterns. In PRXCHANGE function you need either regular-expression-id or perl-regular-expression. If using a regular-expression-id, then PRXPARSE needs to be used.

With PRXCHANGE function it is not required to store the pattern in a regular-expression-id, the pattern can be used directly within the function. In addition, to specifying the pattern, the number of times the pattern should be searched and replaced needs to be specified. If TIMES = -1, then the search and replace will repeat to the end of the string. The last argument in PRXCHANGE is the source where the search and replace occurs.

SAS Program 9 illustrates two different approaches on how the data in Data Display 6 can be masked using PRXCHANGE. The first approach there are three separate statements for PRXCHANGE with the perl-regular expression used as the first argument. The second approach utilizes the regular-expression-id which is generated using the PRXPARSE. The perl-regular expression is used in the PRXPARSE function and is assigned a numeric value to identify the pattern. This numeric value is then used in PRXCHANGE as the regular-expression-id.

```
data pet2;
   set PETS;
   NARRATIVE = prxchange('s/\d{3}:/XXX:/', -1, NARRATIVE);
   NARRATIVE = prxchange('s/\d{4}-\d{2}-\d{2}/DDMONYYYY/', -1, NARRATIVE);
   NARRATIVE = prxchange('s/\d{4}/DDMONYYYY/', -1, NARRATIVE);
run;

OR

data pet2b (drop = regid:);
   set PETS;
   regid1 = prxparse('s/\d{3}:/XXX:/');
   regid2 = prxparse('s/\d{4}-\d{2}-\d{2}/DDMONYYYY/');
   regid3 = prxparse('s/\d{4}/DDMONYYYY/');
   NARRATIVE = prxchange(regid1, -1, NARRATIVE);
   NARRATIVE = prxchange(regid2, -1, NARRATIVE);
   NARRATIVE = prxchange(regid3, -1, NARRATIVE);
run;
```

SAS Program 9: Using PRXCHANGE Function

These functions could be nested but as you can tell they can become difficult to read. When nesting you would work from the inside out, with the pattern that needs to be matched first being the innermost function call.

```
NARRATIVE = prxchange('s/\d{4}/DDMONYYYY/', -1, prxchange('s/\d{4}-\d{2}-
\d{2}/DDMONYYYY/', -1, prxchange('s/\d{3}:/XXX:/', -1, NARRATIVE)));
```

There is also a CALL routine for PRXCHANGE. It works similar to PRXCHANGE function with the differences being that the function outputs to another variable (i.e., VAR = PRXCHANGE(…)). In the example the variable it is writing to happens to be the same variable that is being processed. If the original variable is to be overwritten, then the CALL PRXCHANGE routine could be used instead. CALL PRXCHANGE returns the value as one of its arguments instead of writing to another variable. Another difference between the function and the CALL routine is that the CALL routine ***must*** use PRXPARSE.

The following is the syntax for CALL PRXCHANGE.

CALL PRXCHANGE(*regular-expression-id*, *times*, *old-string* <, *new-string* <, *result-length* <, *truncation-value* <, *number-of-changes*> > > >);

CALL PRXCHANGE has more arguments than the function however only the first three are required which is what is used to illustrate in SAS Program 10.

```
data pet3 (drop = regid:);
   set PETS;
   regid1 = prxparse('s/\d{3}:/XXX:/');
   regid2 = prxparse('s/\d{4}-\d{2}-\d{2}/DDMONYYYY/');
   regid3 = prxparse('s/\d{4}/DDMONYYYY/');
   call prxchange(regid1, -1, NARRATIVE);
   call prxchange(regid2, -1, NARRATIVE);
   call prxchange(regid3, -1, NARRATIVE);
run;
```

SAS Program 10: Using CALL PRXCHANGE

Both SAS Program 9 and SAS Program 10 produce the same output, Data Display 7.

PETID	NARRATIVE
001	Pet ID number XXX: date of birth DDMONYYYY. Had the following conditions: Auto immune, diabetic, bladder cancer
002	Pet ID number XXX: date of birth DDMONYYYY. Had the following conditions: N/A
003	Pet ID number XXX: date of birth DDMONYYYY. Had the following conditions: Malocclusion (overgrown teeth)
004	Pet ID number XXX: date of birth DDMONYYYY. Had the following conditions: N/A
005	Pet ID number XXX: date of birth DDMONYYYY. Had the following conditions: N/A
006	Pet ID number XXX: date of birth DDMONYYYY. Had the following conditions: Skin condition, congestive heart failure
007	Pet ID number XXX: date of birth DDMONYYYY. Had the following conditions: Skin allergies, heart attack
008	Pet ID number XXX: date of birth DDMONYYYY. Had the following conditions: N/A
009	Pet ID number XXX: date of birth DDMONYYYY. Had the following conditions: Old age
010	Pet ID number XXX: date of birth DDMONYYYY. Had the following conditions: Kidney failure

Data Display 7: Output Produced from SAS Program 9 and SAS Program 10

For details on the metacharacters, visit SAS Tables of Perl Regular Expression (PRX) Metacharacters.

Random Number Generation and Other Specialized SAS CALL subroutines

SAS has provided a number of functions and CALL subroutines to perform sampling on SAS data sets. In the 70s, different functions were available (RANUNI, RANNOR, RANBIN, etc.) to provide different types of sampling subroutines. By the late 1990s, SAS had replaced this familiar functionality with the RAND function and accompanying options and subroutines, largely due to the rapidly evolving (and growing) definition of “big data”. The earlier RANXXX subroutines were able to provide accurate random selection up to approximately two billion records, while the RAND function, which uses the Mersenne-Twister algorithm, provides nearly unrepeatable sequences**.** (SAS Institute Inc., 2022) Our purpose in including the RAND function in a paper on SAS CALL subroutines is to demonstrate how SAS functions and CALL subroutines work hand in hand to deliver specific functionalities. As of SAS 9.4 M5, SAS has deprecated the old RANNOR and RANUNI functions.

The RAND function takes a number of arguments, including the specification of a sampling distribution or type, and a “seed” for random number generation. CALL STREAMINIT provides a seed that is reproducible and is used prior to the invocation of the RAND function. If you do not use CALL STREAMINIT or specify a negative seed value in the RAND function, a different sample is produced every time the sampling routine is rerun. CALL STREAMINIT generates the random number stream, which is then fed into the RAND function.

The syntax for CALL STREAMINIT and the RAND function is as follows.

CALL STREAMINIT(*RNG*);

CALL STREAMINIT(*seed*);

RNG refers to random number generator, which include supported algorithms such as MT64 (64-bit Mersenne twister) and PCG | PCG64i (64-bit permuted congruential generator) among others described in the SAS documentation (SAS Institute Inc., 2022). You can specify a RNG alone, a seed alone, or a combination of a RNG and seed. For our example, we chose to use the PCG RNG and a seed of 0. The RNG argument is not case sensitive and negative seeds are permitted.

RAND(*distribution*, *parameter-1*, …, *parameter-k*);

The RAND function is extremely complex, with many different distributions and sampling parameters possible. It is recommended that users consult the RAND Function SAS documentation for a full listing of all the options (SAS Institute Inc., 2022). For our example, we chose a uniform distribution and no parameters. CALL STREAMINIT provides the seed, and the RAND function creates a random number based on a uniform distribution.

CALL STREAMINIT ROUTINE AND THE RAND FUNCTION

We will demonstrate the use of CALL STREAMINIT and the RAND Function in SAS Program 11.

```
data pet_sample;
   set dd.PETS;
      call streaminit('PCG', 0); /* auto-generate seed */
      ranseq = rand('uniform');
run;

proc sort data = pet_sample;
   by ranseq;
run;

data pet_sample_2;
   set pet_sample;
   by ranseq;
   if _n_ le 5;
run;
```

SAS Program 11: Using the RAND Function with CALL STREAMINIT

NAME	BREED	NICKNAME	RANSEQ
Lancelot		Lancey	0.05560
Kisuke Gregory	Domestic Mediumhair	Cheesecake, Keesters	0.08198
Yoruichi Gregory	Domestic Mediumhair	Yichi, Princess	0.08618
Lily Mendez	Miniture Doberman	Lily	0.10548
Kandi Kier	Poodle/Yorkshire Terrier	Kandi	0.12089

Data Display 8: Output Produced from SAS Program 11

CALL COMPCOST ROUTINE

The COMPGED function is one of many SAS functions that "score" differences between text strings. The GED refers to the 'generalized edit distance'; the 'cost' of recreating the first string from a second string. SAS assigns default costs to certain edits, for example, to APPEND a single character to a string costs 50 points. The CALL COMPCOST Routine allows users to change up the costs for one or more edits.

The syntax for CALL COMPCOST and the COMPGED function is as follows.

COMPGED(*string-1*, *string-2* <, *cutoff*> <, *modifier(s)*>)

String-1 and *string-2* refer to the two character / varchar variables being compared. *Cutoff* allows the user to stop counting the cost of a comparison with long strings. There are a number of modifiers available for use in the COMPGED function, to deal with ignoring case, removing leading blanks, normalizing n literals, etc. It is recommended that users consult the documentation regarding the available modifiers and to see

a detailed description of operations. The default cost of operations with operations in alphabetic order is: Append, 50; Blank, 10; Delete, 100; Double, 20; Fdelete, 200; Finsert, 200, Freplace, 200; Insert, 100; Match, 0; Punctuation, 30; Replace, 100; Single, 20; Swap, 20; and Truncate, 10. Note that Fdelete, Finsert and Freplace works similarly to their counterparts with the exception that when the string is empty there is an extra 100 unit cost.

The purpose of CALL COMPCOST is to allow users to change the default cost of operations, by listing the operation (case agnostic) and the desired cost separated by commas, for as many operation / cost pairs as is desired, as shown in the syntax below.

CALL COMPCOST(*operation-1*, *value-1* <, *operation-2*, *value-2* …>);

We will demonstrate the use of CALL COMPCOST and the COMPGED Function in SAS Program 12 Part 1 and Part 2. In order to perform the example, we have modified the PETS data set slightly by adding a first name variable and an altered first name variable (refer to Data Display 9 and Data Display 10).

```
proc format;
   value $ nm2nm
   'Chew' = 'Chewy' /* 1 insert */
   'Loki' = 'Loky' /* 1 replace */
   'Pepper' = 'Peper' /* 1 delete */
   'Peter' = 'Peter' /* 1 match */;
   value $ ops
   'Chew' = 'insert' /* 1 insert */
   'Loki' = 'replace' /* 1 replace */
   'Pepper' = 'delete' /* 1 delete */
   'Peter' = 'match' /* 1 match */;
run;

data PETS_compcost1;
   length fname revfname $ 20 operation $ 8;
   set dd.PETS;
   if _n_ le 4;
   fname=scan(name,1);
   if fname='Chewbaca' then fname='Chew';
   revfname=put(fname,$nm2nm.);
   operation=put(fname,$ops.);
   fncomp=compged(revfname,fname);
run;
```

SAS Program 12 Part 1: Using the COMPGED Function

FNAME	REVFNAME	OPERATION	FNCOMP
Chew	Chewy	insert	50
Loki	Loky	replace	100
Pepper	Peper	delete	20
Peter	Peter	match	0

Data Display 9: Output Produced from SAS Program 12 Part 1

```
data PETS_compcost2;
   length fname revfname $ 20 operation $ 8;
   set dd.PETS;
   if _n_ le 4;
   fname=scan(name,1);
   if fname='Chewbaca' then fname='Chew';
   revfname=put(fname,$nm2nm.);
   operation=put(fname,$ops.);
   call compcost('match',0,'insert',10,'delete',15,'replace',5);
   fncomp2=compged(revfname,fname);
run;
```

SAS Program 12 Part 2: Using the COMPGED Function with CALL COMPCOST

FNAME	REVFNAME	OPERATION	FNCOMP
Chew	Chewy	insert	10
Loki	Loky	replace	5
Pepper	Peper	delete	15
Peter	Peter	match	0

Data Display 10: Output Produced from SAS Program 12 Part 2

SORT SAS CALL SUBROUTINES

CALL SORT, CALL SORTC, and CALL SORTN all perform the function of reassigning values of specified variables of the same types so that they are in ascending order. For example, if a=3, b=1, and c=2, variables and values are reordered so that a=1, b=2, and c=3. CALL SORT can accommodate input variables of numeric, character and varchar types, and is specific to CAS. As we do not operate in a CAS environment, we will not demonstrate CALL SORT, but focus on CALL SORTC and CALL SORTN for this paper. CALL SORTC reorders character variables and values, while CALL SORTN reorders numeric variables and values. CALL SORTC and CALL SORTN are available for both CAS and SAS 9.4.

CALL SORTC ROUTINE AND CALL SORTN ROUTINE

We will demonstrate the use of CALL SORTC in SAS Program 14 and CALL SORTN in SAS Program 13 Part 1 and Part 2. In order to demonstrate the functionality of CALL SORTC and CALL SORTN, we have added 6 variables to our PETS data set: 3 variables for veterinary visit types, and 3 variables for veterinary visit dates. The resulting data set is called PETS_VISITS.

NAME	VISIT1_TYPE	VISIT2_TYPE	VISIT3_TYPE
Chewbaca Watson	Chemo	Checkup	Chemo

NAME	VISIT1_DATE	VISIT2_DATE	VISIT3_DATE
Chewbaca Watson	10/02/2016	10/04/2016	09/30/2016

NAME	VISIT_COMBO1	VISIT_COMBO2	VISIT_COMBO3
Chewbaca Watson	Chemo|10/02/2016	Checkup|10/04/2016	Chemo|09/30/2016

Data Display 11: Additional data set PETS_VISITS

The syntax for CALL SORTC and the CALL SORTN subroutines is as follows.

CALL SORTC(*variable-1* <, …, *variable-n*>);

CALL SORTN(*variable-1* <, …, *variable-n*>);

Variable-1 – Variable-N refer to character variables in the case of CALL SORTC, and to numeric variables in the case of CALL SORTN. A warning or error will be generated in the log should you mix and match variable types with CALL subroutines (refer to SAS Program 13 Part 1 and SAS Log 3).

```
data PETS_SORTMIXED;
   set PETS_VISITS;
   if _n_=1;
   format visit_date: mmddyy10.;
   call sortn(visit1_date,visit2_date,visit3_type);
run;
```

SAS Program 13 Part 1: CALL SORTN with mixed type arguments

```
ERROR: In a call to the SORT function or routine, argument 1 and argument 3 have
different data types.
ERROR: Internal error detected in function SORTN.  The DATA step is terminating during
the EXECUTION phase.
NOTE: The SAS System stopped processing this step because of errors.
NOTE: Due to ERROR(s) above, SAS set option OBS=0, enabling syntax check mode. This
prevents execution of subsequent data modification statements.
```

SAS Log 3: Log Message Produced from SAS Program 13 Part 1

We correct the error in the CALL SORTN program in SAS Program Part 2 and rerun. Note that users may get warnings or errors depending on how their system is set up.

```
data PETS_SORTN;
   set PETS_VISITS;
   if _n_=1;
   format visit_date: mmddyy10.;
   call sortn(visit1_date,visit2_date,visit3_date);
run;
```

SAS Program 13 Part 2: CALL SORTN

Name	Visit1_Type	Visit2_Type	Visit3_Type
Chewbaca Watson	Chemo	Checkup	Chemo

Name	Visit1_Date	Visit2_Date	Visit3_Date
Chewbaca Watson	09/30/2016	10/02/2016	10/04/2016

Name	Visit_Combo1	Visit_Combo2	Visit_Combo3
Chewbaca Watson	Chemo|10/02/2016	Checkup|10/04/2016	Chemo|09/30/2016

Data Display 12: Using CALL SORTN

Note that the output has not sorted the character variables visit1_type, visit2_type and visit3_type and visit_combo1-3 horizontally but has reordered the values in the number variables visit1_date, visit2_date, and visit3_date.

```
data PETS_SORTC;
   length visit1_type visit2_type visit3_type $ 8;
   length visit_combo1 visit_combo2 visit_combo3 $ 25;
   set PETS_VISITS;
   if _n_=1;
   visit_combo1=catx('|',visit1_type,put(visit1_date,mmddyy10.));
   visit_combo2=catx('|',visit2_type,put(visit2_date,mmddyy10.));
   visit_combo3=catx('|',visit3_type,put(visit3_date,mmddyy10.));
   call sortc(visit1_type,visit2_type,visit3_type);
   call sortc(visit_combo1,visit_combo2,visit_combo3);
   new_visit_type1=scan(visit_combo1,1,'|');
   new_date1=input(scan(visit_combo1,2,'|'),mmddyy10.);
   new_visit_type2=scan(visit_combo2,1,'|');
   new_date2=input(scan(visit_combo2,2,'|'),mmddyy10.);
   new_visit_type3=scan(visit_combo3,1,'|');
   new_date3=input(scan(visit_combo3,2,'|'),mmddyy10.);
   format new_date: mmddyy10.;
run;
```

SAS Program 14: CALL SORTC

NAME	VISTA1_TYPE	VISIT2_TYPE	VISIT3_TYPE
Chewbaca Watson	Checkup	Chemo	Chemo

NAME	VISIT1_DATE	VISIT2_DATE	VISIT3_DATE
Chewbaca Watson	10/02/2016	10/04/2016	09/30/2016

NAME	VISIT_COMBO1	VISIT_COMBO2	VISIT_COMBO3
Chewbaca Watson	Checkup|10/04/2016	Chemo|09/30/2016	Chemo|10/02/2016

NAME	NEW_DATE1	NEW_DATE2	NEW_DATE3
Chewbaca Watson	10/04/2016	09/30/2016	10/02/2016

NAME	NEW_VISIT_TYPE1	NEW_VISIT_TYPE2	NEW_VISIT_TYPE3
Chewbaca Watson	Checkup	Chemo	Chemo

Data Display 13: Using CALL SORTC

In this output, we can see that the character variables visit1_type, visit2_type and visit3_type and visit_combo1-3 have been sorted horizontally, while the numeric variables visit1_date, visit2_date, and visit3_date have not. The dangers of horizontal sorts of pairs of variables discontinuously are obvious, as the dates for each visit now do not match the visit types. Thus, we scan the visit_combo variables using the delimiter to provide correctly paired visits and dates, in this case horizontally ordered by visit type.

VARIABLE CONTROL AND INFORMATION CALL SUBROUTINES

"V" functions are incredibly utile in SAS processing, allowing users to access variable level information contained in SAS Dictionary Tables, the metadata which underlies procedures such as PROC CONTENTS and PROC DATASETS, and access methods such as PROC SQL. "V" functions include VLABEL and VNAME: we will describe the CALL routine versions of these two functions, and two other information CALL subroutines, CALL VNEXT and CALL SET. The "V" functions and these CALL subroutines are most often used in do loops, array processing, and macros.

CALL LABEL ROUTINE AND CALL VNAME ROUTINE

The CALL LABEL Routine obtains the label for a specified variable from SAS metadata and creates a second variable containing the obtained label, much like the VLABEL function. If the specified variable's label is blank, the second (label) variable contains the name of the originating variable. The CALL VNAME Routine performs as the VNAME function does; that is, CALL VNAME retrieves the name of a specified variable from metadata and creates a second variable containing the first variable's name.

The syntax for CALL LABEL and the CALL VNAME subroutines is as follows.

CALL LABEL(*variable-1*, *variable-2*);

CALL VNAME(*variable-1*, *variable-2*);

Refer to SAS Program 15 for how to implement CALL VNAME and CALL LABEL. The resulting data can be seen in Data Display 14.

```
data pets_call_label (keep=lab name);
   length name $ 32 lab $ 256;
   set dd.pets (rename=(name=pet_name));

   array abc{*} _character_;  /* all character variables in old */
   array def{*} _numeric_;  /* all numeric variables in old */

   do i=1 to dim(abc);
      call vname(abc{i}, name); /* get name of character variable */
      call label(abc{i},lab); /* get label of character variable */
      output; /* write label to an observation */
   end;

   do j=1 to dim(def);
      call vname(def{j}, name); /* get name of numeric variable */
      call label(def{j},lab);  /* get label of numeric variable */
      output; /* write label to an observation */
   end;
   stop;
run;
```

SAS Program 15: Using the CALL LABEL and CALL VNAME subroutines

NAME	LAB
name	name
lab	lab
pet_name	Name
Nickname	Nickname
Breed	Breed
Type	Type
Status	Status
Date_of_Birth	Date of Birth
Death_Date	Death Date
Medical_Conditions	Medical Conditions
Favorite_Toy	Favorite Toy
Favorite_Thing_to_Do	Favorite Thing to Do
Weight__lb_	Weight (lb)
Length__in_	Length (in)
Age	Age
Data_Entry	Data Entry

Data Display 14: Output Produced from SAS Program 15

CALL VNEXT Routine

The CALL VNEXT Routine returns the variable name and optionally, the variable type and length, of all variables, including automatic variables such as first. and last, in a data set. Note that the order in which variables are processed might not be as expected, as it follows the order of variables in the DDV (data set data vector).

The syntax for CALL VNEXT is as follows.

CALL VNEXT(*variable-name* <,*variable-type* <,*variable-length*> >);

Variable-name is the name of a character variable that CALL VNEXT populates with each variable in the DDV in turn. Once all the variables have been accounted for, the last record contains a blank in the field. Variable-type is an optional argument, and returns “C” (for character) and “N” (for numeric) as values for each variable in turn. Similarly, variable-length is optional and returns the variable length for each variable.

An illustration of CALL VNEXT can be found in SAS Program 16 with the corresponding output found in SAS Log 4.

```
proc sort data=dd.pets (rename=(name=pet_name)) out=pets_call_vnext;
    by pet_name;
run;

data pets_attributes;
   length name $ 32 type $ 3;
   set pets_call_vnext;
   by pet_name;
   if _n_=1;
   name=' ';
   length=666;
   do i=1 to 99 until(name=' ');
      call vnext(name, type, length);
      put i= name @40 type= length=;
   end;
run;
```

SAS Program 16: Using the CALL VNEXT Routine

```
i=1 name                                type=C length=32
i=2 type                                type=C length=3
i=3 pet_name                            type=C length=28
i=4 Nickname                            type=C length=23
i=5 Breed                               type=C length=24
i=6 Weight__lb_                         type=N length=8
i=7 Length__in_                         type=N length=8
i=8 Status                              type=C length=8
i=9 Date_of_Birth                       type=C length=10
i=10 Death_Date                         type=C length=10
i=11 Age                                type=N length=8
i=12 Medical_Conditions                 type=C length=62
i=13 Favorite_Toy                       type=C length=45
i=14 Favorite_Thing_to_Do               type=C length=126
i=15 Data_Entry                         type=N length=8
i=16 FIRST.pet_name                     type=N length=8
i=17 LAST.pet_name                      type=N length=8
i=18 length                             type=N length=8
i=19 i                                  type=N length=8
i=20 _ERROR_                            type=N length=8
i=21 _N_                                type=N length=8
i=22                                    type=  length=0
NOTE: There were 68 observations read from the data set WORK.PETS_CALL_VNEXT.run;
```

SAS Log 4: Produced by SAS Program 16

CALL SET ROUTINE

The CALL SET routine is analogous to a SET statement when used in a data step process, and therefore not terribly useful in the data step, as it requires you to use the OPEN and CLOSE functions. It is used to link SAS data set variables to DATA step or macro variables that have the same name and data type, and can be a convenient way to mass produce macro variables. The CALL SYMPUT or CALL SYMPUTX subroutines and PROC SQL INTO are other ways to accomplish this, among others, but CALL SET (or %SYSCALL SET) has the advantage of automatically doing some cleanup such as trimming and setting the macro variables to the correct length.

The syntax for CALL SET is as follows.

CALL SET(*dsid*);

Note that CALL SET is used in conjunction with a number of functions and macro functions, such as %SYSFUNC, FETCHOBS, GETVARN, GETVARC, and SYSMSG.

In SAS Program 17 Part 1 and Part 2, we demonstrate using CALL SET with the macro facility first (%SYSCALL SET), and then with the data step (CALL SET).

```
%let name=;
%let breed=;
%let rc=%sysfunc(open(DD.PETS));
%syscall set(rc);
%let rc1=%sysfunc(fetchobs(&rc,1));
%let rc2=%sysfunc(sysmsg());
%let rc3=%sysfunc(curobs(&rc));
%let Weight=%sysfunc(getvarn(&rc,5));
%let rc4=%sysfunc(close(&rc));
%put Returned Variables : name=&name breed=&breed weight=&weight;
%put Return Codes : RC2=&Rc2 Rc3=&Rc3;
run;
```

SAS Program 17 Part 1: Using the CALL SET Routine – Macro version

```
265         %let name=;
266         %let breed=;
267         %let rc=%sysfunc(open(DD.PETS));
NOTE: Data file DD.PETS.DATA is in a format that is native to another host, or the file
encoding does not match the session encoding. Cross Environment Data Access will be
      used, which might require additional CPU resources and might reduce performance.
268         %syscall set(rc);
269         %let rc1=%sysfunc(fetchobs(&rc,1));
270         %let rc2=%sysfunc(sysmsg());
271         %let rc3=%sysfunc(curobs(&rc));
272         %let Weight=%sysfunc(getvarn(&rc,5));
273         %let rc4=%sysfunc(close(&rc));
274         %put Returned Variables : name=&name breed=&breed weight=&weight;
Returned Variables : name=Chewbaca Watson              breed=Maltese/Poodle
weight=14
275         %put Return Codes : RC2=&Rc2 Rc3=&Rc3;
Return Codes : RC2= Rc3=1
276         Run;
```

SAS Log 5: Produced by Program 17 Part 1

```
data dd.pets_call_set;
    set dd.pets (keep=name breed Weight__lb_
                 rename=(name=name weight__lb_=weight));
    if _n_=1;
run;

DATA _null_;
 retain name breed;
 Rc=open('dd.pets_call_set');
 Call set(rc);
 Rc1=fetchobs(rc,1);
 Rc2=sysmsg();
 Rc3=curobs(rc);
 name=getvarc(rc,1);
 breed=getvarc(rc,2);
 weight=getvarn(rc,3);
 Rc4=close(rc);
 Put 'Returned Variables : ' name= / breed= / Weight=;
 Put 'Return Codes : ' Rc2= Rc3=;
Run;
```

SAS Program 17 Part 2: Using the CALL SET Routine – Data step version

```
282         data dd.pets_call_set;
283             set dd.pets (keep=name breed Weight__lb_
284                          rename=(name=name weight__lb_=weight));
NOTE: Data file DD.PETS.DATA is in a format that is native to another host, or the file
encoding does not match the session encoding. Cross Environment Data Access will be
      used, which might require additional CPU resources and might reduce performance.
285             if _n_=1;
286         run;
NOTE: There were 68 observations read from the data set DD.PETS.
NOTE: The data set DD.PETS_CALL_SET has 1 observations and 3 variables.
NOTE: DATA statement used (Total process time):
      real time           0.04 seconds
      cpu time            0.03 seconds


287
288         DATA _null_;
289          retain name breed;
290          Rc=open('dd.pets_call_set');
291          Call set(rc);
292          Rc1=fetchobs(rc,1);
293          Rc2=sysmsg();
294          Rc3=curobs(rc);
295          name=getvarc(rc,1);
```

```
296         breed=getvarc(rc,2);
297         weight=getvarn(rc,3);
298         Rc4=close(rc);
299         Put 'Returned Variables : ' name= / breed= / Weight=;
300         Put 'Return Codes : ' Rc2= Rc3=;
301        Run;

Returned Variables : name=Chewbaca Watson
breed=Maltese/Poodle
weight=14
Return Codes : Rc2=  Rc3=1276
```

SAS Log 6: Produced by Program 17 Part 2

EXTERNAL AND SPECIAL SAS CALL SUBROUTINES

There are a number of SAS CALL subroutines that operate on a number of different levels, with different advantages / disadvantages, parameters, and timing in operations in the SAS system. We will discuss two that may be most familiar to you, CALL EXECUTE and CALL SYMPUT/SYMPUTX. These two sets of CALL subroutines, as with many functions and CALL subroutines, have the capability of going outside of typical data set operations, interrupting the flow through the PDV. For this reason, they are most commonly used in conjunction with the SAS macro facility. CALL EXECUTE is very powerful, and its use can be very complex because of the disconnect between how macro processing works versus how the data step process works. CALL SYMPUT/SYMPUTX is also powerful, and can be combined with %macro loops, data set do loops, arrays and functions to provide solutions to many quandaries involving the creation of macro variables. We will provide examples that will demonstrate both the power and complexity of these CALL subroutines.

CALL EXECUTE ROUTINE

Whether you are just an end-user or a developer, you have probably had some exposure to macros. As you may know, to execute a macro you need to actually invoke it and while that is simple enough to do if you have a few scenarios, this can become tedious if you have to write a call to a macro for a large number of scenarios. Or worse you have all your macro call statements written just to find out later that either a specific macro parameter value is either no longer valid (i.e., not in the data) or you have new values that warrant its own macro call. This situation can be avoided with a data-driven technique that utilizes the CALL EXECUTE routine.

The CALL EXECUTE routine takes one argument and resolves the argument. The resolved argument is then executed at the next step boundary (e.g., run; quit;). With CALL EXECUTE you can dynamically write code that is based on the data, and this includes writing macro call statements.

The syntax is

CALL EXECUTE(*argument*);

Argument can be a character string enclosed in quotation marks. The argument can also be the name of a variable within a DATA step. The variable name is ***not*** enclosed in quotes. It can also be a character expression that resolves to macro expression or a SAS statement. Note that if you are trying to mask certain macro facility characters (e.g., %, &) you need to use single quotes. If you use double quotes in this situation, CALL EXECUTE will resolve to a macro invocation and execute the macro immediately.

SAS Program 18 demonstrates how macro calls can be generated based on values in the data set using CALL EXECUTE.

```
%macro pettype(type = );  ❶
   %let dsn = %scan(&type, 1);
   proc means data = PETS noprint n mean std;
      var AGE;
      where upcase(TYPE) =: %upcase("&type");
      output out = &dsn._age n = n mean = mean std = std;
   run;
   /* … ADDITIONAL SAS CODE … */
%mend pettype;

proc sort data = PETS.PETS
          out = PETS (keep = NAME TYPE AGE);
   by TYPE;
run;

data _null_;   ❷
   set PETS;
   by TYPE;
   if first.TYPE;
   call execute(cats('%pettype(type = ', TYPE, ")"));
run;
```

SAS Program 18: Illustration of CALL EXECUTE to Generate Macro Calls

❶ Macro that needs to be executed for each pet TYPE.

❷ CALL EXECUTE is called from within a data step. Because there is no need to save what is being generated to another SAS data set, DATA _NULL_ is used. Since we only want to execute the macro once for each pet TYPE, the data is subsetted for the first record of each pet TYPE. Thus, for each unique pet TYPE, CALL EXECUTE is run and resolves to a SAS statement. In this case it resolves to a macro call for %pettype. It is populating the macro parameter (TYPE =) based on what is in the PETS data set.

The CALL EXECUTE statement within the DATA _NULL DATA step produces the following 6 macro calls based on the data:

%pettype(type = ‘CAT’)

%pettype(type = ‘DOG’)

%pettype(type = ‘FERRET’)

%pettype(type = ‘GUINEA PIG’)

%pettype(type = ‘IGUANA’)

%pettype(type = ‘RABBIT’)

SAS Log 7 shows a snippet of what is generated from CALL EXECUTE. You can see the full log in the Appendix.

```
NOTE: CALL EXECUTE generated line.
1          + proc means data = PETS noprint n mean std;       var AGE;       where
upcase(TYPE) =: "CAT";       output out = Cat_age
n = n mean = mean std = std;     run;

NOTE: There were 40 observations read from the data set WORK.PETS.
      WHERE UPCASE(TYPE)=:'CAT';
NOTE: The data set WORK.CAT_AGE has 1 observations and 5 variables.
NOTE: PROCEDURE MEANS used (Total process time):
      real time           0.01 seconds
      cpu time            0.01 seconds
```

SAS Log 7: Log Snippet for SAS Program 18

By having the macro calls generated based on the data, removes the user need to manually write code to call the %pettype macro. Thus, if more types are pets are added to the PETS data set, the macro call will

be generated with CALL EXECUTE.

CALL EXECUTE can be used for more than just generating macro call statements. You can also use it for conditional execution so that only certain statements are executed based on the data, application development, and macro development.

CALL SYMPUT AND CALL SYMPUTX SUBROUTINES

At one point in your programming career, you have had to create macro variables. There are several different ways to create a macro variable with one way being CALL SYMPUT or CALL SYMPUTX. Both of these CALL subroutines assign a value in a data step to a macro variable and have two required arguments, the name of the macro variable and the value to be assigned.

The syntax is

CALL SYMPUT(*macro-variable*, *value*);

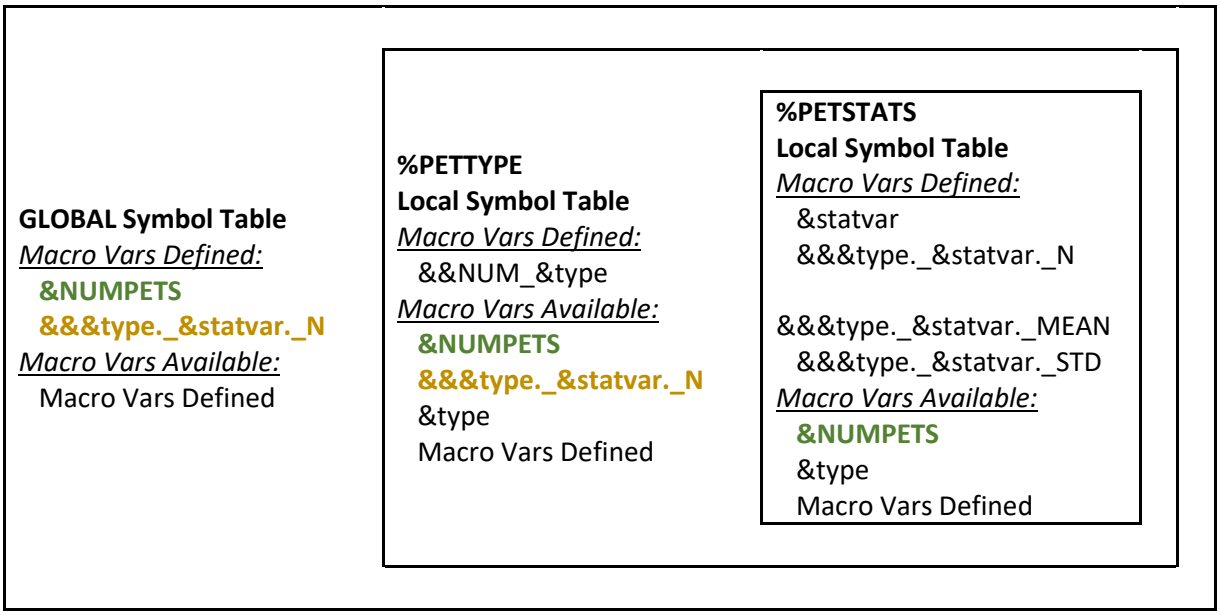
CALL SYMPUTX(*macro-variable*, *value*<, *symbol-table*>);

To understand the difference between these two CALL subroutines, you need to understand the symbol tables. There are local symbol tables and a global symbol table. The local symbol tables only exist within the scope of the macro in which it is created and can only be accessed within that macro. Therefore, local macro variables have no meaning when referenced outside the macro. In fact, you would get the following WARNING message.

```
WARNING: Apparent symbolic reference <<macro-variable>> not resolved.
```

However, the global symbol table exists for the duration of the SAS session and the macro variable can be used anywhere within the SAS program. (SAS Institute Inc., n.d.)

CALL SYMPUT stores the macro variables within the most local symbol table if there is one available otherwise it stores it in the global symbol table. So, what does 'most local' mean? Because a local macro variable only exists within the scope of the macro there can be multiple local symbol tables. The most local is the one in which it is created. Refer to Display 1 for an illustration of the scope.



Display 1: Illustration of Scope of Macro Variables

SAS Program 19 illustrates macro scope and the use of CALL SYMPUT versus CALL SYMPUTX. The global and local symbol tables are written to the log throughout the program to show what is stored in

them at various times during the program execution. Snippets of the log from the program can be found in the Appendix.

```
data pets;
   set PETS.PETS (keep = NAME TYPE WEIGHT__LB_ LENGTH__IN_ AGE);
   format _all_;
   informat _all_;
   if type not in: ('D' 'C') then __TYPE = 'OTHER';
   else __TYPE = type;
run;

data _null_;    ❶
   set pets end = eof;
   if eof then call symput("NUMPETS", _n_);
run;

%macro petstats(statvar = );
   proc means data = &type noprint n mean std;
      var &statvar;
      output out = &type._&statvar n = n mean = mean std = std;
   run;

   data _null_;    ❷
      set &type._&statvar;
      call symputx("&type._&statvar._N", n, 'g');
      call symputx("&type._&statvar._MEAN", mean, 'l');
      call symputx("&type._&statvar._STD", std, 'f');
   run;

   %put **************PETSTATS MACRO***************;
   %put _global_;
   %put _local_;

%mend petstats;

%macro pettype(type = );
   data &type;
      set PETS;
      where upcase(__TYPE) =: %upcase("&type");
   run;

   data _null_;    ❸
      set &type end = eof;
      if eof then call symput("num&type", _n_);
   run;

   %put **************PETTYPE MACRO***************;
   %put _global_;
   %put _local_;

   %petstats(statvar = AGE)
   %petstats(statvar = WEIGHT__LB_)
   %petstats(statvar = LENGTH__IN_)
%mend pettype;

%put **************MAIN PROGRAM***************;
%put _global_ _local_;
%pettype(type = Dog)
%pettype(type = Cat)
%pettype(type = Other)

%put _global_;
%put _local_;
```

SAS Program 19: Program to Illustrate Macro Scope and CALL SYMPUT and CALL SYMPUTX

❶ This DATA _NULL_ step is at the beginning of the program and not within a macro definition. It also uses CALL SYMPUT to create a macro variable NUMPETS. Recall that CALL SYMPUT stores the macro variable within the most local symbol table if one is available otherwise it is stored in the global symbol table. Since one does not exist then NUMPETS is saved in the global symbol table as seen in Appendix SAS Log 2.

❷ The next DATA _NULL_ is found within the inner macro, %PETSTATS. In this paper it is referred to as the inner macro since it is called from within another macro, %PETTYPE, which is referred to as the outer macro. In this DATA _NULL_, CALL SYMPUTX is used so that where the macro variable is stored can be specified.

- All the macro variables that house the count ("&&&type._&statvar._N) are stored in the global symbol table.
- All the macro variables that contain the mean ("&&&type._&statvar._MEAN) are stored in the local symbol table.
- All the macro variables that hold the standard deviation ("&&&type._&statvar._STD) are stored in symbol table where it exists if the macro variable already exists. If it does not exist, then it is stored in the most local symbol table.

❸ The outer macro also has a DATA _NULL_ and it uses CALL SYMPUT. In the DATA _NULL_ found in ❶ the macro variable was stored in the global symbol table. However, for this one it is stored in the local symbol table since a local symbol table does exist.

As illustrated CALL SYMPUT and CALL SYMPUTX differ based on which symbol table the macro variable resides. However, that is not the only difference. CALL SYMPUTX has some other key differences.

- CALL SYMPUTX trims leading and trailing blanks from the macro variable, but CALL SYMPUT does not. This can be seen in the log snippets found in the Appendix.
- If the value argument is numeric, CALL SYMPUT writes a note to the SAS log indicating the value was converted from numeric to character. However, CALL SYMPUTX does not write a note to the log.
- CALL SYMPUTX allows for up to 32 characters when a numeric value is converted to a character value, but CALL SYMPUT only has up to 12 characters.

CREATE A CUSTOM SAS FUNCTION AND SAS CALL ROUTINE WITH PROC FCMP

Although SAS has a lot of great functions and CALL subroutines, sometimes you just cannot find one that meets your needs. The great thing is that SAS allows you to write your own functions and subroutines with the FCMP Procedure. With PROC FCMP, you can create, test and store your own SAS functions and CALL subroutines. It allows you to store and reuse complex code. Most of the SAS features are available within PROC FCMP. However, there are a few that do not work. For example, CALL SORTC cannot be used with PROC FCMP. There are a number of statements that can be used in PROC FCMP. However, we are only going to focus on FUNCTION and SUBROUTINE.

The general syntax for PROC FCMP is

```
PROC FCMP outlib = libname.dataset.package  <options>;
        SUBROUTINE subroutine-name (argument-1 <, argument-2, …>);
                OUTARGS out- argument-1 <, out-argument-2, …>;
                <<< SAS STATEMENTS >>>
        ENDSUB;
        FUNCTION function-name (argument-1 <, argument-2, …>) <$>;
                OUTARGS out- argument-1 <, out-argument-2, …>;
                <<< SAS STATEMENTS >>>
                RETURN(expression)
        ENDFUNC;
QUIT;
```

When using PROC FCMP you need to specify the three-level name. This is the output data set where the compiled subroutines and functions are written.

With a SUBROUTINE you can create independent code that can be used like any other CALL statements in a DATA step. To initiate the subroutine definition, you start with a SUBROUTINE statement, and you close the definition with ENDSUB statement. When defining a subroutine, you need to provide it a name and specify at least one input argument and at least one output argument. The output arguments are specified on the OUTARGS statement which is within the subroutine or function definition. OUTARGS indicates which input arguments are to be updated. When an argument is character you place a dollar sign ($) after the argument name.

The FUNCTION statement allows create a custom function and works similar to other functions in that it returns a value. Like the SUBROUTINE, a FUNCTION definition is initiated with a FUNCTION statement and ends with SUBFUNC statement. It requires at least one argument and if an argument is character then it must have a dollar sign after it. The FUNCTION statement also requires an expression, which is the value that is returned. If the return value is a character value, then a dollar sign is required on the FUNCTION statement after the arguments.

SAS Program 20 demonstrates how to define a subroutine and function.

```
libname pets "C:\Users\gonza\Desktop\Conferences\Pets";

options cmplib = ();

proc fcmp outlib = pets.funcs.mycalc;

   /* impute partial dates */
   subroutine DATVAR(_dat $);    ❶
      outargs _dat;
      if not missing(_dat) and not (notdigit(strip(_dat)) ) then do;
         if length(_dat) = 8 then _dat = strip(_dat);
         else if length(_dat) = 6 then _dat = cats(_dat, '15');
         if length(_dat) = 4 then _dat = cats(_dat, '0615');
      end;
      else _dat = '-';
   endsub;

   function CALCAGE(DOB $, REFDT $);  ❷
      length _dob _ref $10;
      _dob = compress(DOB, '-');
      if not(missing(REFDT)) then _ref = compress(REFDT, '-');
      else _ref = compress(put(today(), yymmdd10.), '-');

      call DATVAR(_dob);
      call DATVAR(_ref);

      if strip(_dob) ne '-' then do;
         _AGE = yrdif(input(_dob, yymmdd8.), input(_ref, yymmdd8.), 'age');
      end;
      else call missing(_AGE);
      return(_AGE);
   endfunc;
quit;
```

SAS Program 20: Using PROC FCMP to Create User-Defined CALL routine and Function

❶ A subroutine, DATVAR, is defined that will impute partial dates. If only the year is provided, then the month and day are imputed to June 15. If year and month are provided, then the day is imputed to the 15th of the month. If the entire date is missing, then a single dash is returned.

❷ The function CALCAGE is created in order to calculate the age based on two character dates that are in the format of YYYY-MM-DD (or YYYYMMDD). If the dates have the dashes, then those are stripped and the subroutine DATVAR is called so that any partial dates can be imputed. Prior to CALL DATVAR, if the REFDT (end date) is missing, then it is set to the current date. Once both dates have

been updated using CALL DATVAR, the function then attempts to calculate age using the SAS function YRDIF.

Once the subroutines and/or functions are compiled and stored, you can reference them in a data step by specifying the SAS data set which contains the compiled subroutines and functions. This can be accomplished using the system option CMPLIB (SAS Program 21 and SAS Program 22). The results produced from SAS Program 21 are seen in Data Display 15.

```
options cmplib = pets.funcs;

data PETS;
   set PETS.PETS (keep = NAME DATE_OF_BIRTH DEATH_DATE);
   NEWAGE = calcage(DATE_OF_BIRTH, DEATH_DATE);
run;
```

SAS Program 21: Using the User-Defined Function

NAME	DATE_OF_BIRTH	DEATH_DATE	NEWAGE
Chewbaca Watson	2003-12-23	2018-01-19	14.0739726
Loki Jack Watson	2018-05-07		3.904109589
Pepper Watson	1996	2008-11-28	12.45479452
Peter Rabbit Watson	2000-02-27	2010-03-25	10.07123288
Easter Watson	2000-04-23	2013-01-16	12.73424658
Bailey Watson	2002-08-08	2018-01-31	15.48219178
Digger Kier	2002-05-25	2011-12-14	9.556164384
Kandi Kier	2012-04-18		9.956164384
TC Brucken	1976	1992	16
Bandit Brucken	1988	2005	17

Data Display 15: Output Produced from SAS Program 21

Although the CALCAGE function calls the DATVAR subroutine, CALL DATVAR can be used as a regular CALL routine as seen in SAS Program 22 with the results in Data Display 16.

```
options cmplib = pets.funcs;

data PETS2;
   set PETS;
   DOB = compress(DATE_OF_BIRTH, '-');
   DDT = compress(DEATH_DATE, '-');
   call datvar(DOB);
   call datvar(DDT);
run;
```

SAS Program 22: Using the User-Defined CALL routine

NAME	DATE_OF_BIRTH	DEATH_DATE	DOB	DDT
Chewbaca Watson	2003-12-23	2018-01-19	20031223	20180119
Loki Jack Watson	2018-05-07		20180507	-
Pepper Watson	1996	2008-11-28	19960615	20081128
Peter Rabbit Watson	2000-02-27	2010-03-25	20000227	20100325
Easter Watson	2000-04-23	2013-01-16	20000423	20130116
Bailey Watson	2002-08-08	2018-01-31	20020808	20180131
Digger Kier	2002-05-25	2011-12-14	20020525	20111214
Kandi Kier	2012-04-18		20120418	-
TC Brucken	1976	1992	19760615	19920615
Bandit Brucken	1988	2005	19880615	20050615

Data Display 16: Output Produced from SAS Program 22

CONCLUSION

SAS has provided some incredible utility in SAS CALL subroutines, which often work hand in glove or resemble SAS functions (sometimes even in names)! We have discussed a wide array of CALL subroutines including those handling string matching; macro, external and special subroutines; sort subroutines; random number generation subroutines; and variable control and information subroutines. We hope our explorations into SAS CALL subroutines enlightens and encourages you to use these valuable tools in your work.

REFERENCES

Campbell, J. (2012). Perl Regular Expressions in SAS® 9.1+ - Practical Applications. *PharmaSUG.* San Francisco: PharmaSUG. Retrieved from https://www.pharmasug.org/proceedings/2012/TA/PharmaSUG-2012-TA08.pdf

Carpenter, A. L. (2018). Using PROC FCMP to the Fullest: Getting Started and Doing More. *SAS Global Forum 2018.* Denver: SAS Institute Inc. Retrieved from https://www.sas.com/content/dam/SAS/support/en/sas-global-forum-proceedings/2018/2403-2018.pdf

Carpenter, A. L. (n.d.). Before You Get Started: A Macro Language Preview in Three Parts. *SAS Global Forum.* SAS Institute Inc. Retrieved from https://support.sas.com/resources/papers/proceedings14/1444-2014.pdf

Hadden, L. (2018). Purrfectly Fabulous Feline Functions. *PharmaSUG 2018.* Seattle: PharmaSUG. Retrieved from https://www.lexjansen.com/pharmasug/2018/EP/PharmaSUG-2018-EP13.pdf

Kunwar, P. (2019). Quick Tips and Tricks: Perl Regular Expressions in SAS®. *SAS Global Forum.* Dallas: SAS Institute Inc. Retrieved from https://www.sas.com/content/dam/SAS/support/en/sas-global-forum-proceedings/2019/4005-2019.pdf

Mullins, L. (2019). Using the PRXCHANGE Function to Remove Dictionary Code Values from the Coded Text Terms. *PharmaSUG.* Philadelphia: PharmaSUG. Retrieved from https://www.pharmasug.org/proceedings/2019/BP/PharmaSUG-2019-BP-315.pdf

SAS Institute Inc. (2022, Feb 16). *CALL STREAMINIT Routine*. Retrieved 2022, from SAS® 9.4 and SAS® Viya® 3.2 Programming Documentation: https://documentation.sas.com/doc/en/pgmsascdc/9.4_3.2/lefunctionsref/p0gw58qo85qp56n1kbpiz50ww8lv.htm

SAS Institute Inc. (2022, Feb 16). *RAND Function*. Retrieved 2022, from SAS® 9.4 and SAS® Viya® 3.2 Programming Documentation: https://documentation.sas.com/doc/en/pgmsascdc/9.4_3.2/lefunctionsref/p0fpeei0opypg8n1b06qe4r040lv.htm

SAS Institute Inc. (2022, Feb 16). *Using Random-Number Functions and CALL subroutines in the DATA Step*. Retrieved 2022, from SAS® 9.4 and SAS® Viya® 3.2 Programming Documentation: https://documentation.sas.com/doc/en/pgmsascdc/9.4_3.2/lefunctionsref/p026ygl6toz3tgn14lt4iu6cl5bb.htm

SAS Institute Inc. (n.d.). *Dictionary of Functions and CALL subroutines*. Retrieved from SAS® 9.4 and SAS® Viya® 3.2 Programming Documentation: https://documentation.sas.com/doc/en/pgmsascdc/9.4_3.2/lefunctionsref/p1q8bq2v0o11n6n1gpij335fqpph.htm

SAS Institute Inc. (n.d.). *FCMP Procedure*. Retrieved from SAS® 9.4 and SAS® Viya® 3.2 Programming Documentation: https://documentation.sas.com/doc/en/pgmsascdc/9.4_3.5/proc/p10b4qouzgi6sqn154ipglazix2q.htm

SAS Institute Inc. (n.d.). *Perl Regular Expressions Tip Sheet.* Retrieved from https://support.sas.com/rnd/base/datastep/perl_regexp/regexp-tip-sheet.pdf

SAS Institute Inc. (n.d.). *Scopes of Macro Variables*. Retrieved from SAS® 9.4 and SAS® Viya® 3.2 Programming Documentation: https://documentation.sas.com/doc/en/pgmsascdc/9.4_3.2/mcrolref/p1b76sxg9dbcyrn1l5age5j5nvgw.htm

SAS Institute Inc. (n.d.). *Tables of Perl Regular Expression (PRX) Metacharacters*. Retrieved from SAS® 9.4 and SAS® Viya® 3.2 Programming Documentation: https://documentation.sas.com/doc/en/pgmsascdc/9.4_3.2/lefunctionsref/p0s9ilagexmjl8n1u7e1t1jfnzlk.htm

Tyndall, R. (2015, 02 13). *SAS macro variables: how to determine scope*. Retrieved from SAS Blogs: https://blogs.sas.com/content/sgf/2015/02/13/sas-macro-variables-how-to-determine-scope/

Watson, R., & Hadden, L. (2019). Quick, Call the 'FUZZ': Using Fuzzy Logic. *MWSUG 2019.* Chicago: MWSUG. Retrieved from https://www.mwsug.org/proceedings/2019/SP/MWSUG-2019-SP-065.pdf

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the authors at:

Richann Watson
DataRich Consulting
richann.watson@datarichconsulting.com

Louise Hadden
Abt Associates, Inc.
louise_hadden@abtassoc.com

APPENDIX

LOG SNIPPET FOR SAS PROGRAM 18

When using CALL EXECUTE, SAS generates a statement (or a macro call) that gets executed at the step boundary and in this case, the six macro call statements are produced and only executed once it gets to the run statement.

With MPRINT option turned, if you examine Appendix SAS Log 1, you will notice that the CALL EXECUTE resolved the macro call (see turquoise highlight) but it none of the PROC MEANS were executed until it got to the step boundary. It wasn't until the DATA _NULL_ reached the end of the file, did resolved arguments from CALL EXECUTE actually execute (see yellow highlight). It was at this point that each of the PROC MEANS that were generated from the resolution of CALL EXECUTE in the DATA _NULL_ DATA step actually ran.

```
42          %macro pettype(type = );
43             %let dsn = %scan(&type, 1);
44             proc means data = PETS noprint n mean std;
45                var AGE;
46                where upcase(TYPE) =: %upcase("&type");
47                output out = &dsn._age n = n mean = mean std = std;
48             run;
49          %mend pettype;
50
51          data _null_;
52             set PETS;
53             by TYPE;
54             if first.TYPE;
55             call execute(cats('%pettype(type = ', TYPE, ")"));
56          run;

MPRINT(PETTYPE):   proc means data = PETS noprint n mean std;
MPRINT(PETTYPE):   var AGE;
MPRINT(PETTYPE):   where upcase(TYPE) =: "CAT";
MPRINT(PETTYPE):   output out = Cat_age n = n mean = mean std = std;
MPRINT(PETTYPE):   run;
MPRINT(PETTYPE):   proc means data = PETS noprint n mean std;
MPRINT(PETTYPE):   var AGE;
MPRINT(PETTYPE):   where upcase(TYPE) =: "DOG";
MPRINT(PETTYPE):   output out = Dog_age n = n mean = mean std = std;
MPRINT(PETTYPE):   run;
MPRINT(PETTYPE):   proc means data = PETS noprint n mean std;
MPRINT(PETTYPE):   var AGE;
MPRINT(PETTYPE):   where upcase(TYPE) =: "FERRET";
MPRINT(PETTYPE):   output out = Ferret_age n = n mean = mean std = std;
MPRINT(PETTYPE):   run;
MPRINT(PETTYPE):   proc means data = PETS noprint n mean std;
MPRINT(PETTYPE):   var AGE;
MPRINT(PETTYPE):   where upcase(TYPE) =: "GUINEA PIG";
MPRINT(PETTYPE):   output out = Guinea_age n = n mean = mean std = std;
MPRINT(PETTYPE):   run;
MPRINT(PETTYPE):   proc means data = PETS noprint n mean std;
MPRINT(PETTYPE):   var AGE;
MPRINT(PETTYPE):   where upcase(TYPE) =: "IGUANA";
MPRINT(PETTYPE):   output out = Iguana_age n = n mean = mean std = std;
MPRINT(PETTYPE):   run;
MPRINT(PETTYPE):   proc means data = PETS noprint n mean std;
MPRINT(PETTYPE):   var AGE;
MPRINT(PETTYPE):   where upcase(TYPE) =: "RABBIT";
MPRINT(PETTYPE):   output out = Rabbit_age n = n mean = mean std = std;
MPRINT(PETTYPE):   run;
NOTE: There were 68 observations read from the data set WORK.PETS.
NOTE: DATA statement used (Total process time):
      real time           0.00 seconds
      cpu time            0.00 seconds
```

```
3                                                          The SAS System
23:51 Friday, April 1, 2022

NOTE: CALL EXECUTE generated line.
1          + proc means data = PETS noprint n mean std;       var AGE;       where
upcase(TYPE) =: "CAT";       output out = Cat_age
n = n mean = mean std = std;    run;

NOTE: There were 40 observations read from the data set WORK.PETS.
      WHERE UPCASE(TYPE)=:'CAT';
NOTE: The data set WORK.CAT_AGE has 1 observations and 5 variables.
NOTE: PROCEDURE MEANS used (Total process time):
      real time           0.01 seconds
      cpu time            0.01 seconds


2          + proc means data = PETS noprint n mean std;       var AGE;       where
upcase(TYPE) =: "DOG";       output out = Dog_age
n = n mean = mean std = std;    run;

NOTE: There were 19 observations read from the data set WORK.PETS.
      WHERE UPCASE(TYPE)=:'DOG';
NOTE: The data set WORK.DOG_AGE has 1 observations and 5 variables.
NOTE: PROCEDURE MEANS used (Total process time):
      real time           0.01 seconds
      cpu time            0.01 seconds


3          + proc means data = PETS noprint n mean std;       var AGE;       where
upcase(TYPE) =: "FERRET";       output out =
Ferret_age n = n mean = mean std = std;    run;

NOTE: There were 2 observations read from the data set WORK.PETS.
      WHERE UPCASE(TYPE)=:'FERRET';
NOTE: The data set WORK.FERRET_AGE has 1 observations and 5 variables.
NOTE: PROCEDURE MEANS used (Total process time):
      real time           0.01 seconds
      cpu time            0.01 seconds


4          + proc means data = PETS noprint n mean std;       var AGE;       where
upcase(TYPE) =: "GUINEA PIG";       output out =
Guinea_age n = n mean = mean std = std;    run;

NOTE: There were 2 observations read from the data set WORK.PETS.
      WHERE UPCASE(TYPE)=:'GUINEA PIG';
NOTE: The data set WORK.GUINEA_AGE has 1 observations and 5 variables.
NOTE: PROCEDURE MEANS used (Total process time):
      real time           0.01 seconds
      cpu time            0.01 seconds


5          + proc means data = PETS noprint n mean std;       var AGE;       where
upcase(TYPE) =: "IGUANA";       output out =
Iguana_age n = n mean = mean std = std;    run;

NOTE: There were 1 observations read from the data set WORK.PETS.
      WHERE UPCASE(TYPE)=:'IGUANA';
NOTE: The data set WORK.IGUANA_AGE has 1 observations and 5 variables.
NOTE: PROCEDURE MEANS used (Total process time):
      real time           0.01 seconds
      cpu time            0.03 seconds
```

```
6          + proc means data = PETS noprint n mean std;       var AGE;       where
upcase(TYPE) =: "RABBIT";       output out =
Rabbit_age n = n mean = mean std = std;    run;
4                                                    The SAS System
23:51 Friday, April 1, 2022


NOTE: There were 4 observations read from the data set WORK.PETS.
      WHERE UPCASE(TYPE)=:'RABBIT';
NOTE: The data set WORK.RABBIT_AGE has 1 observations and 5 variables.
NOTE: PROCEDURE MEANS used (Total process time):
      real time           0.01 seconds
      cpu time            0.00 seconds
```

Appendix SAS Log 1: Log Illustrating Code Produced from CALL EXECUTE

LOG SNIPPET FOR SAS PROGRAM 19

Below are snippets of the log from SAS Program 19. The portions in yellow represent the macro variable that is being created and stored in the global symbol table. The ones in bold italics were created using CALL SYMPUTX so the leading and trailing blanks are removed. The portions in turquoise are the macro variables that are being stored in the most local symbol table.

Keep in mind that once the macro execution has completed the local macro variables for that macro iteration are deleted. In other words, they are not carried to the next macro iteration as seen in the Appendix SAS Log 4 and Appendix SAS Log 5. Notice how the global macro table is accruing more macro variables after each invocation of the macros. That is because in the program CALL SYMPUTX utilized the optional argument that specified the symbol table. By specifying 'g', as the symbol table, indicated that the macro variable being created showed stored in the global symbol table.

```
GLOBAL NUMPETS           68
GLOBAL SASWORKLOCATION "C:\Users\gonza\AppData\Roaming\SAS\EnterpriseGuide\EGTEMP\SEG-
26732-a7f5e0b1\contents\SAS Temporary
Files\_TD19824_RICHANN_\Prc2/"
GLOBAL SYSSTREAMINGLOG true
GLOBAL _CLIENTAPP 'SAS Enterprise Guide'
GLOBAL _CLIENTAPPABBREV EG
GLOBAL _CLIENTMACHINE 'RICHANN'
GLOBAL _CLIENTPROCESSFLOWNAME 'Standalone Not In Project'
GLOBAL _CLIENTPROJECTNAME ''
GLOBAL _CLIENTPROJECTPATH ''
GLOBAL _CLIENTPROJECTPATHHOST ''
GLOBAL _CLIENTTASKLABEL 'CALL SYMPUT and SYMPUTX - Pets.sas'
GLOBAL _CLIENTUSERID 'gonza'
GLOBAL _CLIENTUSERNAME 'Richann Watson'
GLOBAL _CLIENTVERSION '8.3.3.181'
GLOBAL _EG_WORKSPACEINIT 1
GLOBAL _SASHOSTNAME 'Richann'
GLOBAL _SASPROGRAMFILE 'C:\Users\gonza\Desktop\Conferences\Draft Papers\CALL
subroutines\Development\CALL SYMPUT and SYMPUTX - Pets.sas'
GLOBAL _SASPROGRAMFILEHOST 'RICHANN'
GLOBAL _SASSERVERNAME 'Local'
```

Appendix SAS Log 2: List of Global and Local Macro Variables From Main Part of Program Prior to the Macro Calls

```
GLOBAL NUMPETS           68
GLOBAL SASWORKLOCATION "C:\Users\gonza\AppData\Roaming\SAS\EnterpriseGuide\EGTEMP\SEG-
26732-a7f5e0b1\contents\SAS Temporary
Files\_TD19824_RICHANN_\Prc2/"
GLOBAL SYSSTREAMINGLOG true
GLOBAL _CLIENTAPP 'SAS Enterprise Guide'
GLOBAL _CLIENTAPPABBREV EG
GLOBAL _CLIENTMACHINE 'RICHANN'
GLOBAL _CLIENTPROCESSFLOWNAME 'Standalone Not In Project'
GLOBAL _CLIENTPROJECTNAME ''
```

```
GLOBAL _CLIENTPROJECTPATH ''
GLOBAL _CLIENTPROJECTPATHHOST ''
GLOBAL _CLIENTTASKLABEL 'CALL SYMPUT and SYMPUTX - Pets.sas'
GLOBAL _CLIENTUSERID 'gonza'
GLOBAL _CLIENTUSERNAME 'Richann Watson'
GLOBAL _CLIENTVERSION '8.3.3.181'
GLOBAL _EG_WORKSPACEINIT 1
GLOBAL _SASHOSTNAME 'Richann'
GLOBAL _SASPROGRAMFILE 'C:\Users\gonza\Desktop\Conferences\Draft Papers\CALL
subroutines\Development\CALL SYMPUT and SYMPUTX -
Pets.sas'
GLOBAL _SASPROGRAMFILEHOST 'RICHANN'
GLOBAL _SASSERVERNAME 'Local'
PETTYPE NUMDOG           19
PETTYPE TYPE Dog
```

Appendix SAS Log 3: List of Global and Local Macro Variables From First Iteration of the Outer Macro %PETTYPE

```
GLOBAL DOG_AGE_N 19
GLOBAL NUMPETS           68
GLOBAL SASWORKLOCATION "C:\Users\gonza\AppData\Roaming\SAS\EnterpriseGuide\EGTEMP\SEG-
26732-a7f5e0b1\contents\SAS Temporary
Files\_TD19824_RICHANN_\Prc2/"
GLOBAL SYSSTREAMINGLOG true
GLOBAL _CLIENTAPP 'SAS Enterprise Guide'
GLOBAL _CLIENTAPPABBREV EG
GLOBAL _CLIENTMACHINE 'RICHANN'
GLOBAL _CLIENTPROCESSFLOWNAME 'Standalone Not In Project'
GLOBAL _CLIENTPROJECTNAME ''
GLOBAL _CLIENTPROJECTPATH ''
GLOBAL _CLIENTPROJECTPATHHOST ''
GLOBAL _CLIENTTASKLABEL 'CALL SYMPUT and SYMPUTX - Pets.sas'
GLOBAL _CLIENTUSERID 'gonza'
GLOBAL _CLIENTUSERNAME 'Richann Watson'
GLOBAL _CLIENTVERSION '8.3.3.181'
GLOBAL _EG_WORKSPACEINIT 1
GLOBAL _SASHOSTNAME 'Richann'
GLOBAL _SASPROGRAMFILE 'C:\Users\gonza\Desktop\Conferences\Draft Papers\CALL
subroutines\Development\CALL SYMPUT and SYMPUTX -
Pets.sas'
GLOBAL _SASPROGRAMFILEHOST 'RICHANN'
GLOBAL _SASSERVERNAME 'Local'
PETSTATS DOG_AGE_MEAN 9.6559609496
PETSTATS DOG_AGE_STD 4.748402252
PETSTATS STATVAR AGE
```

Appendix SAS Log 4: List of Global and Local Macro Variables From First Iteration of the Inner Macro %PETSTATS

```
GLOBAL DOG_AGE_N 19
GLOBAL DOG_WEIGHT__LB__N 19
GLOBAL NUMPETS           68
GLOBAL SASWORKLOCATION "C:\Users\gonza\AppData\Roaming\SAS\EnterpriseGuide\EGTEMP\SEG-
26732-a7f5e0b1\contents\SAS Temporary
Files\_TD19824_RICHANN_\Prc2/"
GLOBAL SYSSTREAMINGLOG true
GLOBAL _CLIENTAPP 'SAS Enterprise Guide'
GLOBAL _CLIENTAPPABBREV EG
GLOBAL _CLIENTMACHINE 'RICHANN'
GLOBAL _CLIENTPROCESSFLOWNAME 'Standalone Not In Project'
GLOBAL _CLIENTPROJECTNAME ''
GLOBAL _CLIENTPROJECTPATH ''
GLOBAL _CLIENTPROJECTPATHHOST ''
GLOBAL _CLIENTTASKLABEL 'CALL SYMPUT and SYMPUTX - Pets.sas'
GLOBAL _CLIENTUSERID 'gonza'
GLOBAL _CLIENTUSERNAME 'Richann Watson'
```

```
GLOBAL _CLIENTVERSION '8.3.3.181'
GLOBAL _EG_WORKSPACEINIT 1
GLOBAL _SASHOSTNAME 'Richann'
GLOBAL _SASPROGRAMFILE 'C:\Users\gonza\Desktop\Conferences\Draft Papers\CALL
subroutines\Development\CALL SYMPUT and SYMPUTX -
Pets.sas'
GLOBAL _SASPROGRAMFILEHOST 'RICHANN'
GLOBAL _SASSERVERNAME 'Local'
PETSTATS DOG_WEIGHT__LB__MEAN 25.368421053
PETSTATS DOG_WEIGHT__LB__STD 24.649116564
PETSTATS STATVAR WEIGHT__LB_
```

Appendix SAS Log 5: List of Global and Local Macro Variables From Second of the Inner Macro %PETSTATS

```
GLOBAL DOG_AGE_N 19
GLOBAL DOG_LENGTH__IN__N 16
GLOBAL DOG_WEIGHT__LB__N 19
GLOBAL NUMPETS           68
GLOBAL SASWORKLOCATION "C:\Users\gonza\AppData\Roaming\SAS\EnterpriseGuide\EGTEMP\SEG-
26732-a7f5e0b1\contents\SAS Temporary
Files\_TD19824_RICHANN_\Prc2/"
GLOBAL SYSSTREAMINGLOG true
GLOBAL _CLIENTAPP 'SAS Enterprise Guide'
GLOBAL _CLIENTAPPABBREV EG
GLOBAL _CLIENTMACHINE 'RICHANN'
GLOBAL _CLIENTPROCESSFLOWNAME 'Standalone Not In Project'
GLOBAL _CLIENTPROJECTNAME ''
GLOBAL _CLIENTPROJECTPATH ''
GLOBAL _CLIENTPROJECTPATHHOST ''
GLOBAL _CLIENTTASKLABEL 'CALL SYMPUT and SYMPUTX - Pets.sas'
GLOBAL _CLIENTUSERID 'gonza'
GLOBAL _CLIENTUSERNAME 'Richann Watson'
GLOBAL _CLIENTVERSION '8.3.3.181'
GLOBAL _EG_WORKSPACEINIT 1
GLOBAL _SASHOSTNAME 'Richann'
GLOBAL _SASPROGRAMFILE 'C:\Users\gonza\Desktop\Conferences\Draft Papers\CALL
subroutines\Development\CALL SYMPUT and SYMPUTX -
Pets.sas'
GLOBAL _SASPROGRAMFILEHOST 'RICHANN'
GLOBAL _SASSERVERNAME 'Local'
PETSTATS DOG_LENGTH__IN__MEAN 21.0625
PETSTATS DOG_LENGTH__IN__STD 12.566987175
PETSTATS STATVAR LENGTH__IN_
```

Appendix SAS Log 6: List of Global and Local Macro Variables From Third Iteration of the Inner Macro %PETSTATS

```
GLOBAL CAT_AGE_N 39
GLOBAL CAT_LENGTH__IN__N 30
GLOBAL CAT_WEIGHT__LB__N 39
GLOBAL DOG_AGE_N 19
GLOBAL DOG_LENGTH__IN__N 16
GLOBAL DOG_WEIGHT__LB__N 19
GLOBAL NUMPETS           68
GLOBAL OTHER_AGE_N 8
GLOBAL OTHER_LENGTH__IN__N 7
GLOBAL OTHER_WEIGHT__LB__N 8
GLOBAL SASWORKLOCATION "C:\Users\gonza\AppData\Roaming\SAS\EnterpriseGuide\EGTEMP\SEG-
26732-a7f5e0b1\contents\SAS Temporary
Files\_TD19824_RICHANN_\Prc2/"
GLOBAL SYSSTREAMINGLOG true
GLOBAL _CLIENTAPP 'SAS Enterprise Guide'
GLOBAL _CLIENTAPPABBREV EG
GLOBAL _CLIENTMACHINE 'RICHANN'
GLOBAL _CLIENTPROCESSFLOWNAME 'Standalone Not In Project'
GLOBAL _CLIENTPROJECTNAME ''
```

```
GLOBAL _CLIENTPROJECTPATH ''
GLOBAL _CLIENTPROJECTPATHHOST ''
GLOBAL _CLIENTTASKLABEL 'CALL SYMPUT and SYMPUTX - Pets.sas'
GLOBAL _CLIENTUSERID 'gonza'
GLOBAL _CLIENTUSERNAME 'Richann Watson'
GLOBAL _CLIENTVERSION '8.3.3.181'
GLOBAL _EG_WORKSPACEINIT 1
GLOBAL _SASHOSTNAME 'Richann'
GLOBAL _SASPROGRAMFILE 'C:\Users\gonza\Desktop\Conferences\Draft Papers\CALL
subroutines\Development\CALL SYMPUT and SYMPUTX -
Pets.sas'
GLOBAL _SASPROGRAMFILEHOST 'RICHANN'
GLOBAL _SASSERVERNAME 'Local'
```

Appendix SAS Log 7: List of Global and Local Macro Variables After the Final Macro Execution

MACRO ITERATION	SCOPE	MACVAR	CALL ROUTINE
	GLOBAL	NUMPETS	SYMPUT
1	PETTYPE	NUMDOG	SYMPUT
1.1	GLOBAL	DOG_AGE_N	SYMPUTX
1.1	PETSTATS	DOG_AGE_MEAN	SYMPUTX
1.1	PETSTATS	DOG_AGE_STD	SYMPUTX
1.2	GLOBAL	DOG_WEIGHT__LB__N	SYMPUTX
1.2	PETSTATS	DOG_WEIGHT__LB__MEAN	SYMPUTX
1.2	PETSTATS	DOG_WEIGHT__LB__STD	SYMPUTX
1.3	GLOBAL	DOG_WEIGHT__LB__N	SYMPUTX
1.3	PETSTATS	DOG_WEIGHT__LB__MEAN	SYMPUTX
1.3	PETSTATS	DOG_WEIGHT__LB__STD	SYMPUTX
2	PETTYPE	NUMCAT	SYMPUT
2.1	GLOBAL	CAT_AGE_N	SYMPUTX
2.1	PETSTATS	CAT_AGE_MEAN	SYMPUTX
2.1	PETSTATS	CAT_AGE_STD	SYMPUTX
2.2	GLOBAL	CAT_WEIGHT__LB__N	SYMPUTX
2.2	PETSTATS	CAT_WEIGHT__LB__MEAN	SYMPUTX
2.2	PETSTATS	CAT_WEIGHT__LB__STD	SYMPUTX
2.3	GLOBAL	CAT_WEIGHT__LB__N	SYMPUTX
2.3	PETSTATS	CAT_WEIGHT__LB__MEAN	SYMPUTX
2.3	PETSTATS	CAT_WEIGHT__LB__STD	SYMPUTX
3	PETTYPE	NUMOTHER	SYMPUT
3.1	GLOBAL	OTHER_AGE_N	SYMPUTX
3.1	PETSTATS	OTHER_AGE_MEAN	SYMPUTX
3.1	PETSTATS	OTHER_AGE_STD	SYMPUTX
3.2	GLOBAL	OTHER_WEIGHT__LB__N	SYMPUTX
3.2	PETSTATS	OTHER_WEIGHT__LB__MEAN	SYMPUTX
3.2	PETSTATS	OTHER_WEIGHT__LB__STD	SYMPUTX
3.3	GLOBAL	OTHER_WEIGHT__LB__N	SYMPUTX
3.3	PETSTATS	OTHER_WEIGHT__LB__MEAN	SYMPUTX
3.3	PETSTATS	OTHER_WEIGHT__LB__STD	SYMPUTX

Appendix Table 1: Summary of Macro Variables Created in SAS Program 19