PhUSE 2011

Paper CC07

Seven sharp surgery tips for Clinical Programmers

Rohit Banga, BIOP AG, Basel, Switzerland
David Garbutt, BIOP AG, Basel, Switzerland

ABSTRACT

Clinical programmers face a variety of challenging tasks daily. For many of these tasks we know that there are numerous ways to achieve desired results using SAS. This paper highlights some of the useful techniques that we have learned while working with SAS and illustrates each snippet with real examples and code. We aim to make a collection useful to beginners and experts and hope everyone will learn something useful or at the very least be reminded of a technique they have forgotten.

The sharp tips we will reveal include using formats to replace left joins, using hash objects to perform table lookups and joins, displaying dates differently with Picture Formats, an unexpected behavior when using a merge statement, a simpler to set up and less error prone DO OVER loop, a FCMP Procedure example and importing XML files using SAS Unicode server. Of course all the new programming techniques will produce errors in the SAS log so we will also show an innovative way to facilitate log checking.

INTRODUCTION

This paper describes some useful SAS® Techniques for intermediate and advanced SAS users. We have presented some real life scenarios along with SAS code to illustrate our point.

USE FORMATS INSTEAD OF JOINS

The validation checks for some complex studies can easily be more than 400 and many checks require left joins with other datasets . Instead of making left joins to perform table lookups we use formats. This technique is routinely employed by us in programming validation checks for Data Management studies.

Imagine a scenario where we need to check whether Adverse Event Start date is before Study Completion Date or not. In essence we need to check whether `aev.aevstdt >= cmp.lasttrt` for any particular patients.

We will first make dummy `aev` & `cmp` datasets -

```
*** Data CMP with Last Treatment Date ;
data cmp ;
    Input Subject $ LastTRT $9.;
  DataLines;
SUB_1 01JAN2010
SUB_2 21MAR2010
SUB_3 14NOV2010
SUB_4 22DEC2010
 ;
 run;

 *** Data AEV with Adverse Event Date ;

data dar;
```

```
  Input Subject $ Visit DOSDT $9.;
  dataLines;
SUB_1 1 01FEB2009
SUB_1 2 21JAN2010
SUB_1 3 13NOV2009
SUB_2 1 01JAN2009
SUB_2 2 22MAR2010
SUB_2 3 14NOV2009
;
run;
```

The traditional method would involve making a Left join / Merge between `AEV` & `CMP` dataset to get the Last Treatment date for each subject. However, we can also use formats to perform this lookup.

To achieve that we will first make a Dataset with three columns – `Fmtname`, `Start` & `Label`. '`Fmtname`' is equal to the name of format which would be '`LastTRT`'. The column `Start` is the coded Value which is equal to `cmp.subject` and column `Label` is the Coded Text which is equal to `cmp.lastTRT`.

This format dataset can be read by PROC FORMAT and result in a Format `$LastTRT`.

```
*** Make Input dataset for Formats ;

Data format(Keep = start end label fmtname);
    Set CMP;
    Length start end label $20.;
    If _N_ eq 1 then do;
      Start = 'OTHER';
      End = 'OTHER';
      Label = '';
      Fmtname = '$LastTRT';
      output;
    end;
    Start = Subject;
    End = Subject;
    Label = LastTRT;
    Fmtname = '$LastTRT';
    If Not missing(Label) then
        output;
run;

*** Output the format ;

proc format cntlin=format;
run;

proc format cntlout = chk_fmt;
run;
```

Now this format can be used multiple times in a dataset to perform table lookup.

```
*** Use the formatted value instead of Join;
data output;
    Set DAR;
    Last_Treatment_Date = Put(subject,$LastTRT.);
    If Input(DOSDT,??Date9.) GT input(Put(subject,$LastTRT.),??Date9.);
run;
```

UNEXPECTED BEHAVIOR OF MERGE STATEMENT

Consider this unexpected behavior of after merging two datasets –

Imagine we have a Lab Dataset with one record per patient per visit per lab parameter. We have

a column with Visit Numbers and another column for Visit Names.

```
data Lab;
     Input Subject $ VisitNo  VisitName $10. LabParam $;
datalines;
SUB_1 1 OldVisit1 HGB
SUB_1 1 OldVisit1 HCT
SUB_1 1 OldVisit1 GLUC
SUB_1 1 OldVisit1 WBC
SUB_1 2 OldVisit2 HGB
SUB_1 2 OldVisit2 HCT
SUB_1 2 OldVisit2 WBC
SUB_1 2 OldVisit2 GLUC
SUB_1 3 OldVisit3 HGB
SUB_1 3 OldVisit3 HCT
;
run;
```

We have a dataset `vis` where we have again have one record per patient per visit. This dataset has a Visit Number column and Visit Name column.

```
data vis;
    input Subject $ VisitNo  VisitName $10.;
   datalines;
SUB_1 1 NewVisit1
SUB_1 2 NewVisit2
SUB_1 3 NewVisit3
SUB_2 1 NewVisit1
SUB_2 2 NewVisit2
SUB_2 3 NewVisit3
;
run;
```

Our Task is that we want to put New VisitNames from the `VIS` dataset in the `LAB` Dataset. For this purpose we normally merge the two datasets keeping the `VIS` dataset on the right side and we imagine that since the VisitName column is in the `VIS` dataset on the right side, it will over write the Visit Names in the `LAB` Dataset in the left hand side and we will achieve our purpose.

```
data result;
    Merge Lab(in =a)  Vis (in=b);
    By Subject VisitNo;
    If a;
  run;
```

Let's look at the result of the above Merge in a graphic way–

	Subject	VisitNo	VisitName	LabParam
1	SUB_1	1	OldVisit1	HGB
2	SUB_1	1	OldVisit1	HCT
3	SUB_1	1	OldVisit1	GLUC
4	SUB_1	1	OldVisit1	WBC
5	SUB_1	2	OldVisit2	HGB
6	SUB_1	2	OldVisit2	HCT
7	SUB_1	2	OldVisit2	WBC
8	SUB_1	2	OldVisit2	GLUC
9	SUB_1	3	OldVisit3	HGB
10	SUB_1	3	OldVisit3	HCT

+

	Subject	VisitNo	VisitName
1	SUB_1	1	NewVisit1
2	SUB_1	2	NewVisit2
3	SUB_1	3	NewVisit3
4	SUB_2	1	NewVisit1
5	SUB_2	2	NewVisit2
6	SUB_2	3	NewVisit3

Expected Result –

	Subject	VisitNo	LabParam	VisitName
1	SUB_1	1	HGB	NewVisit1
2	SUB_1	1	HCT	NewVisit1
3	SUB_1	1	GLUC	NewVisit1
4	SUB_1	1	WBC	NewVisit1
5	SUB_1	2	HGB	NewVisit2
6	SUB_1	2	HCT	NewVisit2
7	SUB_1	2	WBC	NewVisit2
8	SUB_1	2	GLUC	NewVisit2
9	SUB_1	3	HGB	NewVisit3
10	SUB_1	3	HCT	NewVisit3

Actual Result

	Subject	VisitNo	VisitName	LabParam
1	SUB_1	1	NewVisit1	HGB
2	SUB_1	1	OldVisit1	HCT
3	SUB_1	1	OldVisit1	GLUC
4	SUB_1	1	OldVisit1	WBC
5	SUB_1	2	NewVisit2	HGB
6	SUB_1	2	OldVisit2	HCT
7	SUB_1	2	OldVisit2	WBC
8	SUB_1	2	OldVisit2	GLUC
9	SUB_1	3	NewVisit3	HGB
10	SUB_1	3	OldVisit3	HCT

If you look closely then you will find that the expected result is not same as the actual result. In reality the Visit Name column has the first value correct but the rest of the values are not correct.

This happens because during a match-merge SAS first merges the first BY- groups in the two datasets and `vis.visitname` successfully overwrites `lab.visitname` in the first observation. At the end of first iteration of data step SAS reinitializes variables to missing. SAS then determines if there are observations remaining for the current BY group to merge. No observations are left for the current BY – group in the VIS dataset and therefore in the next observation `lab.visitname` does not overwrite `lab.visitname` and the original value of `lab.visitname` is preserved.

To overcome this behaviour we should drop the variable `lab.visitname` so that when the merge occurs it is only `vis.visitname` which makes the `visitname` column.

Some useful options by which we can detect the overwriting of columns is

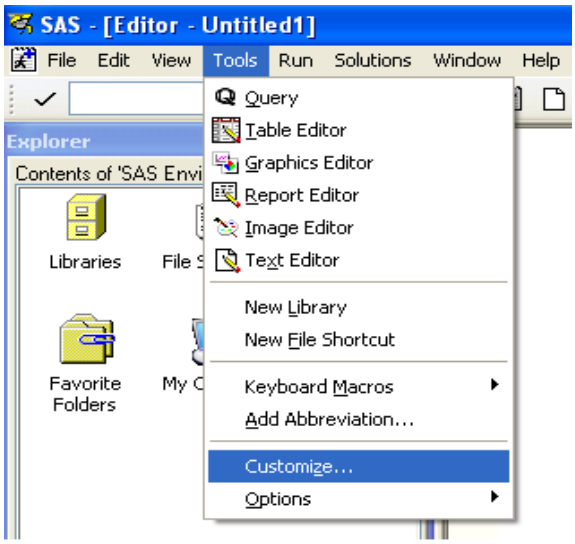
```
Options msglevel = i.
```

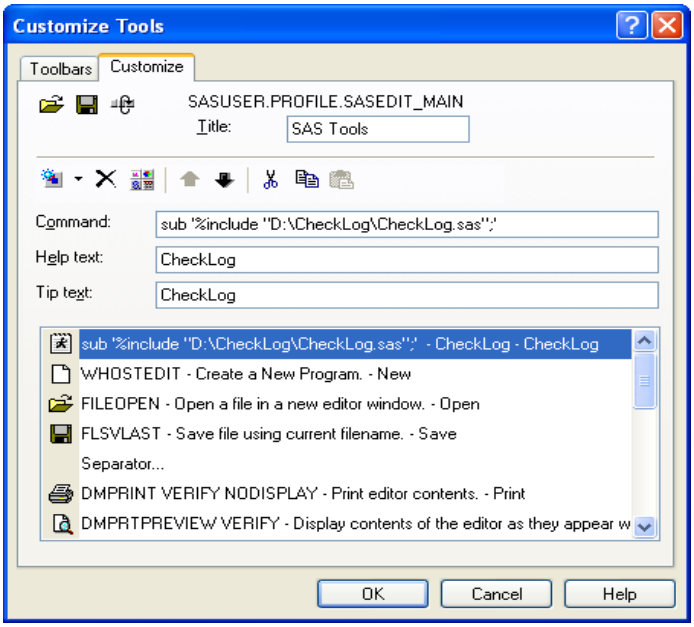
This option will result in a NOTE in the SAS Log when a variable from one dataset overwrites another variable while merging.

INNOVATIVE WAY OF CHECKING LOG

The purpose of this technique is more to demonstrate how to add custom functionality to SAS rather than checking the log itself.

It is easy to add buttons on the SAS Toolbar and enhance the functionality of currently available tools in SAS.



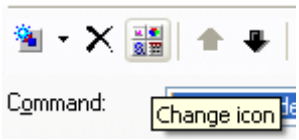


Click on Tools – Customize and click on 'Customize' tab of the 'Customize Tools' dialogue box that opens.

Suppose our SAS program to check logs is stored in `'D:\CheckLog\CheckLog.sas'`. Write the Statement –

```
sub '%include "D:\CheckLog\CheckLog.sas";'
```

to include the program.



Do not forget to assign an Icon by clicking the Change Icon button. Otherwise the new button will not be visible on the toolbar.

After we assign the button and click OK, a new button is visible on the SAS toolbar. This button can be pushed to run the program `CheckLog.sas`. There are numerous ways to write a code to check the log. One of the ways is displayed here –

```
filename cat catalog 'work.chklog.chklog.log' ;
dm 'log;file cat' ; * write log to catalog member ;

data Log (drop=err);
    infile cat end=end truncover;
    input line $1-1000;
    retain Err;
    if substrn(strip(line),1,5) = 'ERROR' or substrn(strip(line),1,7) = 'WARNING'
    then do;
        LineNumber= _N_;
        err=1;
        output Log;
        call symput('obs',1);
    end;
    if end and err = 1 then do;
      window status rows=15 columns=40 color=blue
      #5 "Errors & Warnings were produced in the log. " color=White
      #7 "Press ENTER to continue." color=Black;
      display status ;
    end;
run;

Data _null_;
    if 0 then set Log nobs = nobs;
    if nobs = 0 then do;
         window status rows=15 columns=40 color=cyan
         #5 "No Errors or Warnings were produced in the log. " color=Black
         #7 "Press ENTER to continue." color=Black;
         display status ;
      end;
stop;
run;

filename cat ; *-- free catalog member ;
dm 'del work.chklog.chklog.log' ; *-- now delete it ;
```

USING PICTURE FORMATS TO DISPLAY DATES

SAS provides numerous ways to display dates. However many times we want to display dates in a special format which makes it easy for an investigator to read. Picture formats can help to display dates in almost any format you desire.

To create this picture we need to issue special characters called 'Directives' in the PICTURE statement. Some of the permitted directives are defined in the table. The examples are shown

for the date – 1st March 2009 and the value in brackets in the result column tell us what the directive will display.

Time period	*code*	*result*
Year	%Y	2009
	%y	9
	%0y	09
Month	%m	3
	%0m	03
Month Name	%B	March
	%b	Mar
Day	%d	1
	%0d	01
Day of Week	%A	Sunday
	%a	Sun
	%w	1
Hour	%0H	01
Minute	%0M	01
Second	%0S	01

For the full list of directives please see the SAS documentation linked in the References section.

Defining a picture format is similar to defining a normal format using a value statement. In our example, we will just define the 'Other' value and write the directives. We also need to write the `datatype` which is equal to `date` in our case.

Suppose we want to display 1st March, 2009 as 01/03/2009. We will choose our directive as
'%0d/%0m/%Y' which breaks down as:

`%0d` (to display date with a leading zero), followed by '/' then

`%0m` (to display numeric month with a leading zero), followed by '/' and then finally

`%Y` (to display year in full 4 letter format).

Run the code below and check the log for different types of date formats.

```
PROC FORMAT;
PICTURE SDDMMYY OTHER = '%0d/%0m/%Y'          (DATATYPE=DATE);
PICTURE SDDMONYY OTHER = '%0b.%d.%Y  '        (DATATYPE=DATE);
PICTURE SWEEK OTHER = '%A, %B %d, %Y         '(DATATYPE=DATE LANGUAGE=ENGLISH);
PICTURE SPLUS OTHER = 'Maybe -> %0d+%B+%0y   '(DATATYPE=DATE LANGUAGE=ENGLISH);
RUN;

Data _Null_;
    Date = date() ;
    put 'Date with Slashes        %0d/%0m/%Y    :' Date sddmmyy. ;
    put 'Date with Points         %0b.%d.%Y     :' Date sddmonyy.;
    put 'Date with Weekday Name   %A, %B %d, %Y :' Date sweek.   ;
    put 'Date with Plus           %0d+%B+%0y    :' Date splus.   ;
run;
```

USE HASH OBJECTS FOR TABLE LOOKUP

The hash object is an addition to the types of tools available to the SAS programmer. Hash objects are also known as 'content addressable arrays' and this is a useful way to think of them. Think of the task of counting the distinct values of a variable, `labparm`. In a data step we might use an array[1] to keep the distinct values in and another array to make a count, write some careful counting logic only adding a new value if not found. With a content addressable array we

[1] If we were programming this – normally – we should use proc SQL or another proc for this kind of task because they are faster and easier to use (and read).

merely have a statement like:

```
Disvalue[labparm] ++ 1 ;
```

Here we add one to the element of the array (`disvalue`) with the value of `labparm`. After we have added all the values we print the array index and counts. This is conceptually much simpler and with no tricky bits of programming to get right.[2]

HASH ARRAY REPLACING FORMAT LOOK-UP

The SAS hash array can be used as a look up technique. We will use a hash array to reprogram the format example we have already used.

```
*** merge using a hash table instead of a format *** ;
data houtput ;

    set dar ;
    if _n_ = 1 then do;
          rc = 0 ;
           length Subject $ 8  LastTRT $9;
          *--                    declare  the hash object. Ours is called endtrt
          *--                    load the hash obj by specifying the dataset name ;
          DECLARE HASH endtrt(dataset: "work.cmp",HASHEXP:16);
          rc + endtrt.DEFINEKEY ('SUBJECT');
          rc + endtrt.DEFINEDATA('SUBJECT', 'LastTRT');
          rc + endtrt.DEFINEDONE();
     end ;

         *--- load the value of lasttrt in from the hash that is stored with
                current value of subject;
    rc = endtrt.find() ;

       *-- we have one record per visit - now select the visits after last dose ;
       If input(DOSDT,??Date9.) GT input(LastTRT,??Date9.) and rc = 0 ;

    drop rc ;
run;
```

The program sets the main dataset (`dar`) then the hash object called `endtrt` is declared and loaded by specifying the `dataset`: keyword. The next three statements define the key (`subject`) and the data to be kept (`LastTRT`). Here we are keeping only one variable from `Work.Cmp` but it could be more than one (unlike the format solution). Then we call `definedone` to indicate the hash object is ready. This initialization and loading of the hash object should only be done once and therefore this whole section is enclosed in an `if _n_ = 1` test. The hash messages (`definekey, definedata, definedone`) all deliver a return code and these values should be collected and tested for errors. Here this code is omitted but the downloadable file contains more checking code to illustrate.

This method uses fewer lines of code than the format method and the lines are all integrated in one data step. According to SAS Institute the hash method also runs faster. The hash method can use multiple keys and return multiple variables which the format technique cannot do.

So should you always use the hash method? The short answer is – it depends. The hash object cannot be stored so if you have many programs using your format mapping you might prefer a format. Also if you are doing a range conversion a hash object cannot help you, nor can it help if you want to create a format to use in PROC SQL code.

---

2 Coding needed to set the dimensions of the array in the do loop using `dim(arrayname)` but it is so easy to leave that for later, or think the array will always be the same dimension. Not until later when an `array indexed out of bounds` message pops up on the day of testing will the pain be felt.

Hashes can be loaded directly from a dataset as shown above so in that sense they could be shared.

HASH OBJECTS TO REPLACE SURPRISING MERGE

The usage of the hash object is very similar to the first case we looked at. Instructively similar.

```
*** merge using a hash table instead of a data step model the datastep exactly *** ;
data hResult ;
   set lab ;
   if _n_ = 1 then do;
      rc = 0 ;
      length Subject $ 8  visitname $10;
           *-- declare  the hash object. Ours is called endtrt
           *-- load the hash obj by specifying the dataset name ;
           DECLARE HASH visnam(dataset: "work.vis",HASHEXP:16);
           rc + visnam.DEFINEKEY ('SUBJECT','VISITNO');
           rc + visnam.DEFINEDATA('SUBJECT', 'VISITNO', 'VISITNAME');
           rc + visnam.DEFINEDONE();
           if rc then
           put 'ERR' 'OR: phuse demo: cannot initialise hash' _n_= rc= subject=
              visitno= visitname=;
           *-- adding the next statement does nothing, ever, but it prevents an
           *-- 'visitname is uninitialised' message;
       if 0 then visitname = ' ' ;
    end ;
         *--- load the value of visit name in the from the hash that is stored with
         *--- current value of subject and visit;
    rc = visnam.find() ;
    drop rc ;
  run;
```

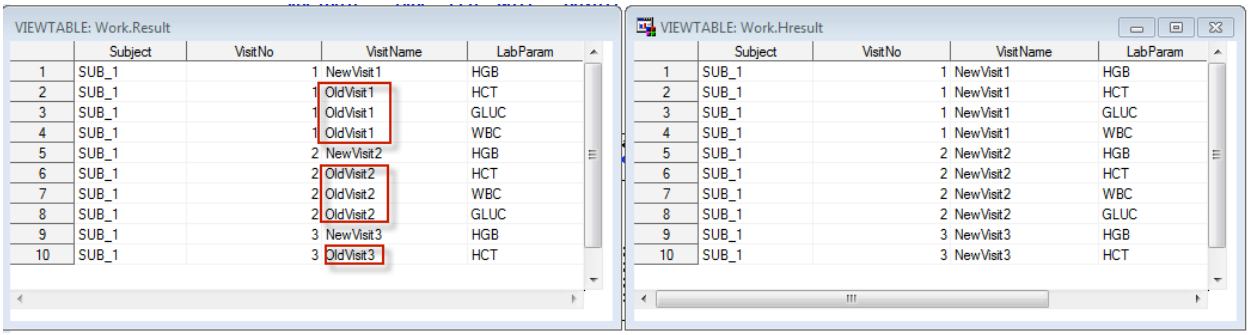
And the result is as expected – not with a mixture of sources for the visit name column. The above statement reveals an interesting aspect of SAS programming because surely we should guarantee the same visit names are used for all subjects? Our program above does not do that because the subject *and* visit number are used as keys. We can easily change the above program to use just the `visitnum` as a key because the program makes no assumptions about the order in the dataset used to load the data.

```
data hResult2 ;
   set lab ;
   if _n_ = 1 then do;
      rc = 0 ;
      length        visitno 8  visitname $10;
           *-- declare  the hash object. Ours is called endtrt
           *-- load the hash obj by specifying the dataset name ;
           DECLARE HASH visnam(dataset: "work.vis",HASHEXP:16);
           rc + visnam.DEFINEKEY ('VISITNO');
           rc + visnam.DEFINEDATA('VISITNO', 'VISITNAME');
           rc + visnam.DEFINEDONE();
           if rc then
           put 'ERR' 'OR: cannot initialise hash' _n_= rc= subject= visitno= visitname=;
           *-- adding the next statement does nothing, ever, but it prevents an
           *-- 'visitname is uninitialised' message;
       if 0 then visitname = ' ' ;
    end ;

         *--- load the value of visit name in the from the hash that is stored with
         *--- current value of  visit ;
    rc = visnam.find() ;
    drop rc ;
  run;
```

Why does not the original program do this? The reason is that SAS would need to re-sort the

`lab` dataset by visit number. This of course could take a long time.[3]



VIEWTABLE: Work.Result

	Subject	VisitNo	VisitName	LabParam
1	SUB_1	1	NewVisit1	HGB
2	SUB_1	1	OldVisit1	HCT
3	SUB_1	1	OldVisit1	GLUC
4	SUB_1	1	OldVisit1	WBC
5	SUB_1	2	NewVisit2	HGB
6	SUB_1	2	OldVisit2	HCT
7	SUB_1	2	OldVisit2	WBC
8	SUB_1	2	OldVisit2	GLUC
9	SUB_1	3	NewVisit3	HGB
10	SUB_1	3	OldVisit3	HCT

VIEWTABLE: Work.Hresult

	Subject	VisitNo	VisitName	LabParam
1	SUB_1	1	NewVisit1	HGB
2	SUB_1	1	NewVisit1	HCT
3	SUB_1	1	NewVisit1	GLUC
4	SUB_1	1	NewVisit1	WBC
5	SUB_1	2	NewVisit2	HGB
6	SUB_1	2	NewVisit2	HCT
7	SUB_1	2	NewVisit2	WBC
8	SUB_1	2	NewVisit2	GLUC
9	SUB_1	3	NewVisit3	HGB
10	SUB_1	3	NewVisit3	HCT

The `declare hash` statement can also take a `duplicate: 'e'` argument tag the effect of this is to cause an error if any key-value pairs are duplicated. The default action is to take the first occurrence and ignore any duplicates. Adding this to our program does not give the checking needed in this case (to ensure the same visit number always has the same label) unless the data for the hash are prepared using a PROC SQL query with a `distinct` keyword.

PROC FCMP TO REPLACE MACROS

Although it has been possible to call functions compiled with other languages in SAS for some time PROC FCMP (Function CoMPile) offers something new because it compiles functions from data step code. Well, there are a few limitations, but they are few.

SAS macro language was designed so programs could be parameterized and this is accomplished by modifying the text that the SAS parser will see. If you code a transformation as a SAS macro the code will be created at each call and then still compiled. This is not how a normal function works. A function is compiled and the binary code is called each time the function is invoked in a program. *It reduces the amount of code for compilation.* This is the kind of function that PROC FCMP can create.

Our example is a function that recodes character variables that are to be output into a file that will be processed by LaTeX into a PDF. Why this is a good idea could be the subject of a whole paper and a televised debate, but suffice it to say the output quality generated by LaTeX is second to none, it is very easy to generate PDF links that work, and straight forward to create documents with indexes, bookmarks and full cross-referencing[4].

The task this function does is to substitute characters that are special to LaTeX with their equivalent string that will render correctly in print.

This kind of substitution is better done using regular expressions or one of the family of `translate` functions, but in this case they cannot be used. The reason is that that the substituted strings contain characters to be substituted.

```
$ → \$
\ → \\
```

Therefore it is easiest to pass once through the string to be encoded substituting as many characters as needed. Here is an example of the original code, clever because it solves the sequence

3 Another solution would be to index the lab dataset with two indexes and avoid the double sorting

4 For example if you were documenting SAS datasets or XPT files you might appreciate an index showing every page each variable and dataset are mentioned.

problem by getting it right but it is not maintainable, especially when you know this code was repeated 20+ times in the program[5].

```
data _null_;
infile "dataset_list.txt" lrecl=10000 dsd dlm='|'
        end=lastrow length=linelen column=currcol;
input dsname $ descrip : $32. @; …

call symput ('descrip' || left (_n_),
  tranwrd((tranwrd ((tranwrd ((tranwrd ((tranwrd ((tranwrd
    ((tranwrd ((tranwrd ((tranwrd ((tranwrd
     (trim (descrip),'&','\&')),
          '{','\{')), '}','\}')), '_','\_{}')), '%','\%')),
         '~','\~{}')), '$','\$')), '<','$<$')), '>','$>$')),
         '^','\^{}')), '#','\#'));
```

We will not discuss PROC FCMP in detail but concentrate on what it can do for you. What does a barebones function definition look like?

```
proc fcmp outlib= sasuser.MySubs.davefunc ;

function Latexencode(var $ ) $ 1024;
   /* code goes here*/
endsub ;

quit;

options CMPLIB = sasuser.Mysubs;
/* run some tests */
...
```

Only five lines to be added to your code. The call of the `proc` and the `function` and `endsub` statements that enclose the function. Identifiers in the parentheses are the parameter names to be used inside the function. Arrays can also be passed. The `return` statement will define what value will be delivered by the function.

To call it write `latexencode(string)`.[6] How does the code look? Just like it would in a data step.

```
function Latexencode(var $ ) $ 1024;
length result $ 1024 c tab sp $ 1 ;
result = '' ; c='' ; sp = ' ' ; tab = byte(5);
do i= 1 to length(var) ;
  c = substr(var,i,1) ;
  select (c);
     when (' ') result = catt(result, TAB) ;
     when ('\') result = catt(result, '$\backslash$');
     when ('{') result = catt(result,'\{');
     when ('}') result = catt(result, '\}');
     when ('%') result = catt(result,'\%' );
     when ('&') result = catt(result,'\&' );
     when ('~') result = catt(result,'\~{}');
     when ('$') result = catt(result, '\$');
     when ('^') result = catt(result, '\^{}');
```

[5] I am not saying the original program was not an achievement. The author was aware of this issue because a 8 year old comment said /* turn this code into a function later */

6 Notice there are some complications with arguments that are long character strings, also put statements should not be used.

```
      when ('_') result = catt(result, '\_{}');
      when ('#') result = catt(result, '\#');
      when ('<') result = catt(result, '$<$');
      when ('>') result = catt(result, '$>$');
      otherwise result = catt(result, c );
    end;
  end;
  result = translate(compress(result), sp,tab) ;
  return(result);
  endsub ;
```

Is it effective? Judge for yourself by comparing the two programs:

```
/* put NAME & Label & Type & Length & format (name and link) */
col1 = latexencode(upcase(varname) ) ;
col2 = latexencode(varlabel);
if type = 1 then
       col3 = ' & Num & ' ;
   else
       col3 = ' & Char & ' ;
col4 = left(put(length, 10.0)) ;
col5 = latexencode(formatname) ;

put col1 '& ' col2 col3 col4 ' & ' '\hyperref['
    col5 ']{' col5 '}' ;
put '} \\';
put' \hline' ;
```


Here is a program extract before and after we changed it to call `latexencode`. Over 220 lines of code that recoded 5 variables and `put` them to a file were condensed to 5 lines. Simpler, more readable, easier to maintain and compiled once.

Functions are not supported in `put` statements so we cannot reduce it to one line, but using the

new cat family of functions we can make it two! The `cat` functions make a common pattern of SAS programming simpler: concatenating strings. Because SAS Character variables are fixed length strings many constructions you might expect to work do not. Using cat functions can save you from nested `trim(left(` function calls thus saving, with one call, your sanity – and your fingers. Here is a simple example of what the `cats` can do.

```
data lengthn;
   input string $char8.;
   original = '*' || string || '*';
   stripped = '*' || strip(string) || '*';
   catSed = cats('*',string,'*') ;
   catTed = catt('*',string,'*') ;
   catseped= catx(' ','*',string,'*') ;
   catsepfn= catx(' ','*',upcase(string),'*') ;
datalines;
abcd
  abcd
    abcd
abcdefgh
 x y z
;

proc print data=lengthn;
run;
```

The results are like this, with the input on left (having various combinations of leading and trailing blanks) and results of `concatenate, strip, cats, catt,` and `catx` going from left to right.

```
 The SAS System                                    1

O
B                                                                  --------CATX----------
s Input       ||      strip      catS       catT       Space sep    Uppercase
1 abcd     *abcd    * *abcd*     *abcd*     *abcd*     * abcd *     * ABCD *
2   abcd   *  abcd  * *abcd*     *abcd*     *  abcd*   * abcd *     * ABCD *
3     abcd *    abcd* *abcd*     *abcd*     *    abcd* * abcd *     * ABCD *
4 abcdefgh *abcdefgh* *abcdefgh* *abcdefgh* *abcdefgh* * abcdefgh * * ABCDEFGH *
5  x y z   * x y z  * *x y z*    *x y z*    * x y z*   * x y z *    * X Y Z *
```

The way to remember which is which is that `Cat S` takes off leading and trailing spaces, `Cat T` only trims, and `Cat X` adds the X factor between each string. X, being, obviously what separates our strings from the ordinary cats. The last column is to remind us that `cat` can be passed function calls.

How does this get us to two lines? We can just put the function calls used to create each column into arguments of one `cats` call. We push this idea to the max by also using the `ifc` function to embed a conditional as well. A conditional, note, that easily handles missing values if they occur. The `&` character used is the separator for table cells in a LaTeX table, `\\` is the end of the row, and `\hline` draws a horizontal rule under this table row.

```
Thisline = cats(
    latexencode( upcase(varname) ) ,
    '&',
    latexencode(varlabel),
    ifc( type , 'Num ('||strip(put(length, 10.0))||')',
         'Char (' || strip(put(length, 10.0))||,
        '???No Type defined!') ,
```

```
    ' & \hyperref(' , latexencode(formatname),
    '}{', latexencode(formatname),
    '} \\ \hline'  );
Put Thisline ;
```

With this code (of two logical lines) we output a line in the LaTeX file like this[7]

```
Age\_s & Age at study start & Num (8) & \hyperref{age\_st.}{age\_st.} \\ \hline
```

Which, when turned into PDF by LaTeX has a table row that looks like:

Variable	*Label*	*Type (Length)*	*Format*
Age_s	Age at study start	Num (8)	age_st

The headers were done in a different part of the `data _null_`. The format name is coloured because it is a clickable link to the page of the document where the format content is listed. In the LaTeX code we have created we do not need to know where that is. At the point where the format is output we just add a `\label` to mark that location as one that will be referred to.

PROC FCMP is a powerful addition to SAS because it implements a real subprogram with local variables for the first time, this makes it worth looking at if you are writing a large system with SAS, e.g. A reporting system. For more information see the further reading section.

IMPORTING XML USING SAS UNICODE SERVER

The default coding for XML files is generally UTF-8 when reading XML with the SAS XML engine you should make sure your SAS session is running with the options

```
-DBCS
-ENCODING UTF-8
```

If you do not do this you may get strange characters in your datasets and worse you may get nothing read at all and a message complaining your XML file is malformed. The above options set the default encoding of the SAS session to Unicode. Other files with different encodings can still be read from such a session but the encoding must be specified on the `libname` or `filename` statements. There are also new functions for recoding in which are documented in the NLS support manual. Note that these do not map characters in the same way that specifying an encoding does. So be aware and test carefully. And comprehensively.

UTF MADE SIMPLE(R)

UTF8 encoding takes 1 to 4 bytes per character and the first 128 ASCII characters are the same and are no problem.

Problems do occur with character codes above 128. Wrongly encoded they can result in empty spaces, or control or graphic characters in runs of 2-4. The issue is complicated by differences between Windows code pages and ISO standards.

Some often used characters are allowed in Windows LATIN but not ISO LATIN1:

- curly quotes **“LIKE** THIS she said”
- Plus/minus, etc. **±, ≥, ≤, ≠, µ**
- Long dashes (em and en)

These characters get translated to codes above 256 in Unicode and may or may not appear correctly in the SAS dataset.

---

[7] Here for clarity I show the table row as one line, however this not a Latex requirement. Extra spaces and newlines are just discarded.

BONUS 1 - DO OVER LOOP

The do over loop is a deprecated feature[8] that predates the normal do loop with the `by, to` and `from` keywords. It is used with arrays and defines it own index variable (`_i_`) which is automatically dropped. The syntax is simple:

```
do over <arrayname> ;
…
end;
```

BONUS 2 - USE PUTLOG FOR ERROR MESSAGES IN DATA STEPS OUTPUTTING TO FILES

Error messages are a vital part of any program that will be used more than once or by another person. But writing to the log while writing to other files from a `data _null_ ;` has always been awkward. There is now a solution – use `putlog` it will always send its output to the log even if a file destination is in place.

CONCLUSION

There are many ways to accomplish your goals with SAS, it is always worth exploring new ways because all the methods fit best in different situations.

ACKNOWLEDGMENTS

We thank our colleagues at BIOP for useful input and comments that sharpened these tips and our programming. We thank our families for their continued support and forbearance.

RECOMMENDED READING

Hash objects

There are many papers about using hash objects and a good example is *Why Hash?* by Glen Becker which clearly explains the benefits and limitations of the hash object. Downloadable from the SCSUG site. http://www.scsug.org/SCSUGProceedings/2009/Glen_Becker.pdf.

Arrays and Do over.

See the paper by Jennifer Waller obtainable from
http://support.sas.com/resources/papers/proceedings10/158-2010.pdf

`Proc fcmp`

This procedure can really change the way SAS macro systems are written and the word is spreading see http://support.sas.com/resources/papers/proceedings11/291-2011.pdf and Charlie Huang's blog post here:
http://www.sasanalysis.com/2011/08/macro-design-pattern-by-proc-fcmp.html

Picture statement

Details in the online manual here:
http://support.sas.com/documentation/cdl/en/proc/61895/HTML/default/viewer.htm#a002473467.htm

Szilagyi and Binder, 2006, *Watch out, a MERGE ahead* - http://www.lexjansen.com/phuse/2006/cs/cs02.pdf

[8] Removed from the documentation in version 8. See further reading for more information.