Paper 064-2011

Recreating SAS® Data Sets via SAS Code Generation

Richard D. Langston, SAS Institute Inc., Cary, NC

ABSTRACT

This paper describes a recent project where SAS code was needed that replicated exactly the contents of a SAS data set. The SAS code generator for this project takes into account various important issues such as n-literals and encoding problems. Several specialized functions such as CATS, KCVT, and VNAME are used and explained in the generator, along with special techniques for using _NUMERIC_ and _CHARACTER_.

INTRODUCTION

I was contacted by a development group where there was a need to convert an existing SAS data set into the corresponding SAS code that would recreate this SAS data set. This code might need to be encapsulated within a macro definition, so DATALINES would not be used. The code might need to run on another operating system, so precision of the data values would be important to maintain, and there could not be an assumption about whether the program was run on an ASCII-based or EBCDIC-based system. The program described herein is the macro I created to generate the desired SAS code.

EXAMPLE DATA SET TO CONVERT

Let's first consider a SAS data set that we would want to convert. Here is the SAS code that would have been used to create our example SAS data set:

```
options validvarname=any;
data my_dataset(label='my label here');
     x=1; y='2'; 'a b'n=3; abc='zzz';  output; x=1/3; abc='\%y'; output;
     label y ='label for y'; format x date9.; x=.; 'a b'n=.a; output;
run;
```

Note that we are using VALIDVARNAME=ANY, which enables the use of variable names with embedded blanks and other special characters, such as 'a b'n. Such variables need to be specified as n-literals in SAS syntax, therefore our SAS code generator must be able to accommodate such variable names. The value of x that is computed from 1/3 cannot be represented as a decimal value such as .33333 because there will not be sufficient precision. Therefore, our generator will need to ensure proper precision, but we will not know the original expression used to compute the value. The value of 'x%y' for the variable abc contains a \ – a variant character – and we must be careful that the value is recreated properly regardless of the ENCODING= setting. And because the % and the & characters have special meaning to the macro processor when used in strings with double quotation marks, we must ensure that any quoted strings containing them use single quotation marks in our generated code. And we also need to ensure that variable labels and formats are properly defined, and that the variable lengths are defined as early as possible.

WHAT THE PROGRAM WILL PRODUCE

Let's look at what the resultant SAS code would look like, keeping in mind the various concerns.

```
%let orig_encoding = LATIN1 ;
%let current_encoding = %sysfunc(getoption(encoding));
%macro hexassign(varname,hexvalue);
  %if &orig_encoding = &current_encoding %then %do;
  &varname. = &hexvalue;
  %end; %else %do;
  &varname. = kcvt(&hexvalue,"&orig_encoding","&current_encoding.");
  %end;
%mend hexassign;
options validvarname=any;
data my_dataset(label ="my label here"  );
  length x  8 ;
  length "a b"N  8 ;
  length y  $1 ;
  length abc  $3 ;
```

```
  format x  DATE9.0 ;
  label y ="label for y" ;
  x =1 ;
  "a b"N =3 ;
  y ="2" ;
  abc ="zzz" ;
  output;
  x =input('3FD5555555555555'x,ieee8.);
  "a b"N =3 ;
  y ="2" ;
  %hexassign(abc ,'5C2579'x );
  output;
  x =. ;
  "a b"N =.A ;
  y ="2" ;
  %hexassign(abc ,'5C2579'x );
  output;
run;
```

We need to indicate the ENCODING= option value at the time the conversion program is run. It might be necessary for us to convert to a different encoding when the generated code is executed. We use the %HEXASSIGN macro for all cases where there might be a problem with conversion, such as when variant characters are seen.

If there is a precision issue with numeric values, we will use a hex literal with the INPUT function so that we can maintain the same precision. The IEEE8. informat is used because it is consistent with the big endian layout of SAS hex literals.

The KCVT function might not be familiar to most readers. This function will convert a character string from one encoding to another. For example:

```
x = kcvt('313233'x, 'wlatin1', 'ebcdic1047' );
```

would convert a Windows Latin1 string of 123 (in hex, 313233) into the EBCDIC equivalent (in hex, F1F2F3).

Note also the GETOPTION function, used in this instance with %SYSFUNC. The GETOPTION function will return the value of a specified option. We want to set the macro variable &CURRENT_ENCODING equal to the value of the ENCODING= option, and GETOPTION is the simplest way to do this.

HOW TO PRODUCE THE GENERATED CODE

Now that we see what needs to be generated, we need to produce the code generator itself as a macro. We'll call it %make_data_step and here is our definition:

```
%macro make_data_step(dataset,fileref);
```

We will supply a SAS data set to the macro, along with a fileref target where the generated SAS code will be written.

First we will do some setting up. One of our problems will be the formatting of numeric values that might be special missing values. Typical formats such as BEST will produce just the character A when formatting .A . But DATA step syntax requires a leading dot, so we must be able to format special missing values that way. This is done by producing a format via PROC FORMAT. This SAS code produces a CNTLIN= data set for all missing values, and uses an OTHER= condition for all other numeric values to be formatted using BEST12.

```
missing _ABCDEFGHIJKLMNOPQRSTUVWXYZ;
data cntlin;
     text = '_ ABCDEFGHIJKLMNOPQRSTUVWXYZ';
     retain fmtname 'N_FMTED';
     length c $1 label $7 hlo $2;
     do i = 1 to 28;
        c = substr(text,i,1);
        start = input(c,1.);
        label = '.'||c;
        output;
        end;
     hlo = 'OF'; /* indicates other, and format-as-label */
```

```
      label = 'best12.';
      output;
      keep fmtname start label hlo;
      run;
proc format cntlin = cntlin; run;
proc datasets dd = work nolist; delete cntlin; quit;
```

Here we emit the definition for the %HEXASSIGN macro that will be subsequently used in the generated code. We record the original encoding and use that as the “from” argument in the KCVT function.

```
data _null_; file &fileref.;
      orig_encoding = getoption('encoding');
      put '%let orig_encoding = ' orig_encoding ';';
      put '%let current_encoding = %sysfunc(getoption(encoding));';
      put '%macro hexassign(varname,hexvalue);';
      put '%if &orig_encoding = &current_encoding %then %do;';
      put '&varname. = &hexvalue;';
      put '%end; %else %do;';
      put '&varname. = kcvt(&hexvalue,"&orig_encoding","&current_encoding.");';
      put '%end;';
      put '%mend hexassign;';
      put 'options validvarname=any;';
run;
```

We will need to obtain the metadata for the SAS data set, which can be done via an output data set from PROC CONTENTS. We need to process the metadata in original variable order, indicated by the npos variable. The output data set is sorted by variable name, so we must sort by npos.

```
proc contents data = &dataset. noprint out=contents_output; run;
proc sort data = contents_output; by npos; run;
```

Here we begin to generate the SAS code, starting with the DATA statement. If there is a data set label or a data set type, we will need to have a LABEL= and TYPE= option in the generated DATA statement. The label needs to be in quotation marks, but we don’t use the $QUOTE format because it will produce a string using double quotation marks, which could cause unwanted macro substitution if the label contains % or & characters. The TRANWRD function allows us to convert each single quotation mark into two single quotation marks.

```
data _null_; set contents_output(obs=1); file &fileref. mod;
      length text $32767;
      put "data &dataset." @;
      if memlabel ne ' ' or memtype ne 'DATA' then do;
         put '(' @;
         if memlabel ne ' ' then do;
            text = tranwrd(memlabel,"'","''");
            l = length(text);
            put "label='" text $varying32767. l "' ";
            end;
         if memtype ne 'DATA' then do;
            put 'type =' memtype @;
            end;
         put ')' @;
         end;
      put ';';
run;
```

Next are the LENGTH statements. They need to appear first and in the order of defining instance in the original data set. The NLITERAL function will produce an n-literalized name if necessary.

```
data _null_; set contents_output; file &fileref. mod;
      nlit = nliteral(name);
      put 'length' +1 nlit +1 @;
      if type=2 then do;
```

```
         put '$' @;
         end;
      put length ';';
run;
```

Next are the LABEL statements. No LABEL statement is needed for blank labels. We produce the quoted label text in the same way as the LABEL= option in generating the DATA statement, avoiding $QUOTE. because of the double quotation marks.

```
data _null_; set contents_output; file &fileref. mod;
     length text $32767;
     if label ne ' ';
     text = tranwrd(label,"'","''");
     l = length(text);
     nlit = nliteral(name);
     put 'label ' nlit "='" text $varying32767. l "';" ;
run;
```

Next are the FORMAT statements. No FORMAT statement is needed unless a format name or nonzero format length appears in the metadata.

```
data _null_; set contents_output; file &fileref. mod;
     length text $32767;
     if format ne ' ' or formatl ne 0;
     nlit = nliteral(name);
     text = cats(format,formatl,'.',formatd);
     put 'format' + 1 nlit +1 text ';' ;
run;
```

We are now done with the contents_output data set and it can be deleted.

```
proc datasets dd=work nolist; delete contents_output; quit;
```

We now produce the assignment statements for each of the variable values. We will process all numeric variables first, using an old-style array with the variable list _numeric_. To ensure that we have at least one numeric variable to avoid warning messages, we add one dummy numeric variable called goobly_n. (The ‘goobly’ prefix is used in multiple places here as a unique prefix that will most likely not collide with any names in the data set being read.) The VNAME function is useful here in order to obtain the variable name from an array element. NLITERAL is used here as it has been in several places to ensure n-literalized names are produced when needed. Here is where we use our special FMTED. format to produce proper text for an arbitrary numeric value. And in order to verify that the text produced matches exactly with our numeric value, we informat the text back in via the BEST12. informat to see if the value matches. If not, we will need to produce a hex literal within an INPUT function, using the IEEE8. informat if we are producing the SAS code on an IEEE system, or the S370FRB8. informat if we are producing the code on an IBM mainframe system. For character values, if the value contains anything other than blanks, alphabetic or numeric characters, or a few punctuation characters, the value should not be provided simply as a string in quotation marks due to potential encoding problems in other environments. Therefore, the %HEXASSIGN macro is used with the hex representation of the value. If the value can be placed in quotation marks, we use the $QUOTE. format because we know that the value cannot contain & or %.

Once all values are emitted, we ensure there is an OUTPUT statement. And after all observations are emitted, we have a final RUN statement.

```
data _null_; set &dataset.; file &fileref. mod;
     length goobly_n 8 goobly_c $32767;
     array goobly_na _numeric_;
     array goobly_ca _character_;
     do over goobly_na;
        if _i_ = dim(goobly_na) then leave; /* skip goobly_n */
        goobly_c = nliteral(vname(goobly_na));
        put goobly_c '=' @;
        goobly_c = put(goobly_na,n_fmted.);
        if input(goobly_c,best12.) = goobly_na
```

```
            then put goobly_c @;
         else do;
            %if &sysscp ne OS %then %do;
            put "input('" goobly_na hex16. "'x,ieee8.)" @;
            %end; %else %do;
            put "input('" goobly_na hex16. "'x,s370frb8.)" @;
            %end;
            end;
         put ';';
         end;
      do over goobly_ca;
         if _i_ = dim(goobly_ca) then leave;  /* skip goobly_c */
         goobly_c = nliteral(vname(goobly_ca));
         if verify(upcase(goobly_ca),
                   ' ABCDEFGHIJKLMNOPQRSTUVWXYZ_1234567890()*:;"'<>,./') then do;
            put '%hexassign(' goobly_c ',' @;
            goobly_c = cats("'",put(goobly_ca,$hex.),"'x");
            put goobly_c ');';
            end;
         else do;
            put goobly_c '=' @;
            goobly_c = put(goobly_ca,$quote.);
            put goobly_c @;
            end;
         put ';';
         end;
      put 'output;';
run;
data _null_; file &fileref. mod;
      put 'run;';
run;
```

We are now at the end of our macro.

```
%mend make_data_step;
```

We want to test this macro to ensure the data sets match. Here is the test code for that, using our original SAS code that produces our special SAS data set.

```
options validvarname=any;
data my_dataset(label='my label here');
     x=1; y='2'; 'a b'n=3; abc='zzz';  output; x=1/3; abc='x%y'; output;
     label y ='label for y'; format x date9.; x=.; 'a b'n=.a; output; run;
run;
data save; set my_dataset; run;
filename saspgm temp;
options mprint;
%make_data_step(my_dataset,saspgm);
proc datasets dd=work nolist; delete my_dataset; quit;
%include saspgm/source2; run;
filename saspgm clear;

proc compare data=save compare=my_dataset; run;
```

The output from PROC COMPARE should show that the data sets are identical.

AN ALTERNATE ENDING

An interesting trend on movie DVDs is the addition of alternate endings. These consist of footage that was shot and considered as the ending for the movie at some point, but was eventually abandoned based on certain criteria. Likewise, we have an alternate ending for this SAS code generator. This approach generates the same code as above, except for the last DATA step which uses old-style arrays. Instead, the data set is opened with the OPEN function, observations are read with the FETCH function, and individual variable values are read with the GETVARN

and GETVARC functions. There is no problem with name space requiring the use of special names. The generated code is exactly the same.

```
data _null_; file &fileref. mod;
     length c $32767;
     ds = open("&dataset.");
     if ds eq 0 then stop;
     nvars = attrn(ds, "NVARS");
     do while(fetch(ds)=0);
        do i = 1 to nvars;
           name = nliteral(varname(ds, i));
           type = vartype (ds, i);
           if type = 'N' then do;
              put name '=' @;
              n = getvarn(ds,i);
              c = put(n,n_fmted.);
              if input(c,best12.) = n
                 then put c @;
              else do;
                 %if &sysscp ne OS %then %do;
                 put "input('" n hex16. "'x,ieee8.)" @;
                 %end; %else %do;
                 put "input('" n hex16. "'x,s370frb8.)" @;
                 %end;
                 end;
              end;
           else do;
              c = getvarc(ds,i);
              if verify(upcase(c),
                 ' ABCDEFGHIJKLMNOPQRSTUVWXYZ_1234567890()*:;"'''<>,./') then do;
                 put '%hexassign(' name ',' @;
                 c = "'"||trim(put(trim(c),$hex.))||"'x";
                 put c ')';
                 end;
              else do;
                 put name '=' @;
                 c = put(c,$quote.);
                 put c @;
                 end;
              end;
           put ';';
           end;
        put 'output;';
        end;
     put 'run;';
     rc = close(ds);
run;
```

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author:

Richard D. Langston
SAS Campus Drive
SAS Institute Inc.
E-mail: Rick.Langston@sas.com