Paper 4659-2020

Write SAS® to Generate SAS: Three Code-Generating Techniques for Many Automation Solutions

Yun (Julie) Zhuo, PRA Health Sciences

ABSTRACT

While SAS has no shortage of automation techniques, it does not always occur to SAS programmers to write SAS codes that generate SAS codes. This paper introduces three SAS techniques having the capacity of generating SAS codes from driver data sets. Driver data sets can be external specification files, the readily-available SAS dictionary metadata, or even the data values in the original data sets. Through real-world examples, we will demonstrate that the code generators can be effective tools in many applications where there are either repetitive programming or immature and changing data. Additionally, this paper will also provide programming tips and compare the three techniques using the simplicity, flexibility, and efficiency metrics.

INTRODUCTION

Although typing is an inseparable part of programming, repetitive typing does not have to be. SAS has an abundance of techniques available to automate repetitive tasks, but it does not always occur to SAS programmers that we can write a few SAS codes to generate a bunch of SAS codes.

In this paper, we are going to introduce, demonstrate, and compare the following three SAS techniques with code-generating capabilities:

- Create and then resolve a macro variable
- Call Execute routine
- PUT statements that write codes to an associated file

This paper will demonstrate the capacities of the code generators through three practical examples. The examples utilize different but all readily available driver data sets. We will demonstrate that the code generating techniques will improve efficiency and accomplish automation tasks that may not be easily accomplished otherwise.

EXAMPLE 1: AUTOMATE THE CREATION OF VARIABLE LABELS

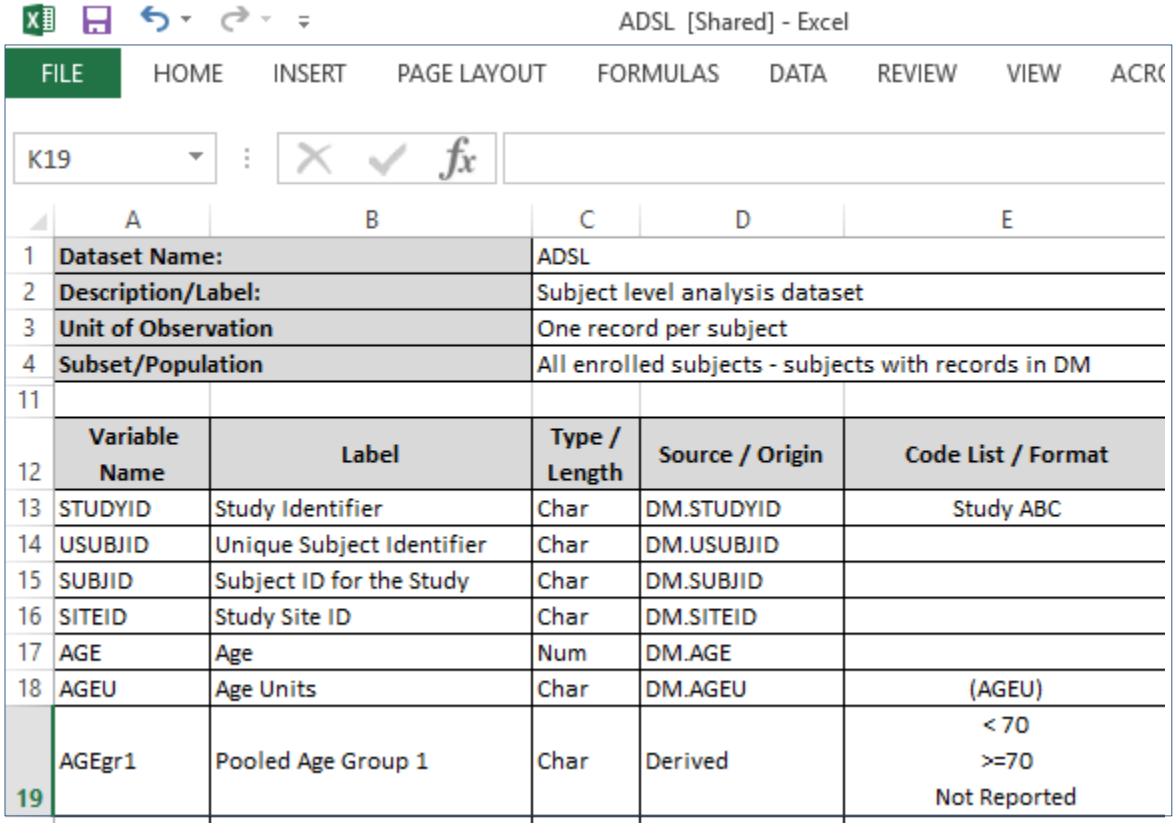
SAS programmers create a large amount of data sets. Adding descriptive variable labels to SAS data sets is necessary for them to be useful downstream. Traditionally, programmers type up variable labels using label statements, as shown in Display 1. The ability to automatically label variables instead of manually typing up many lines of codes in a label statement will be a great time saver.

DRIVER DATA SET IN EXAMPLE 1

The driver data set used in this example is imported from a specification file in the format of an EXCEL® spreadsheet. As a standard practice, the specification spreadsheet was created before programming works start, therefore it is readily available to the programmers. This useful spreadsheet stores specification information such as data set variable name, variable label, variable type, variable origin, and variable derivation logic. A partial snapshot of the spreadsheet is illustrated in Display 2.

```
create table adam.adsl as
select STUDYID label='Study Identifier', USUBJID label='Unique Subject Identifier',
       AGE label='AGE', AGEU label='Age Units', SEX label='Sex',
       RACE label='Race', RACEN label='Race (N)', ETHNIC label='Ethnicity',
       ETHNICN label='Ethnicity (N)', COUNTRY label='Country',
       HEIGHT label='Height (cm)', WEIGHT label='Weight (kg)',
       BMI label='Body Mass Index', SYSBP label='Systolic Blooc Pressure (mmHg)',
       DIABP label='Diastolic Blood Pressure (mmHg)',
       HR label='Heart Rate (beats/min)', TEMP label='Temperature (c)',
       ECOG label='ECOG Performance Status', HIST label='Histology',
```

Display 1. Create Variable Labels using Label Statements



	A	B	C	D	E
1	**Dataset Name:**		ADSL		
2	**Description/Label:**		Subject level analysis dataset		
3	**Unit of Observation**		One record per subject		
4	**Subset/Population**		All enrolled subjects - subjects with records in DM		
11					
12	**Variable Name**	**Label**	**Type / Length**	**Source / Origin**	**Code List / Format**
13	STUDYID	Study Identifier	Char	DM.STUDYID	Study ABC
14	USUBJID	Unique Subject Identifier	Char	DM.USUBJID	
15	SUBJID	Subject ID for the Study	Char	DM.SUBJID	
16	SITEID	Study Site ID	Char	DM.SITEID	
17	AGE	Age	Num	DM.AGE	
18	AGEU	Age Units	Char	DM.AGEU	(AGEU)
19	AGEgr1	Pooled Age Group 1	Char	Derived	< 70 >=70 Not Reported

Display 2. Specification Data in Excel spreadsheet format.

With the following IMPORT procedure, we create a driver SAS data set (META.SAS7BDAT) out of the spreadsheet:

```
PROC IMPORT OUT= WORK.META
            DATAFILE= "<file directory and name>"
            DBMS=EXCEL REPLACE;
     RANGE="<sheet name>.$A12:B200";
     GETNAMES=YES;
     MIXED=NO;
     SCANTEXT=YES;
     USEDATE=YES;
     SCANTIME=YES;
RUN;
```

TECHNIQUE 1: THE MACRO VARIABLE METHOD

The Macro Variable method creates a SAS macro variable to store the codes. Resolving the macro variable after a LABEL statement in a DATA step will generate the SAS codes we need to create labels for all the variables from the driver data set. With macro options such as SYMBOLGEN, we are able to view the generated codes in the log. The diagram below in Figure 1 demonstrates the three-step process of the technique.

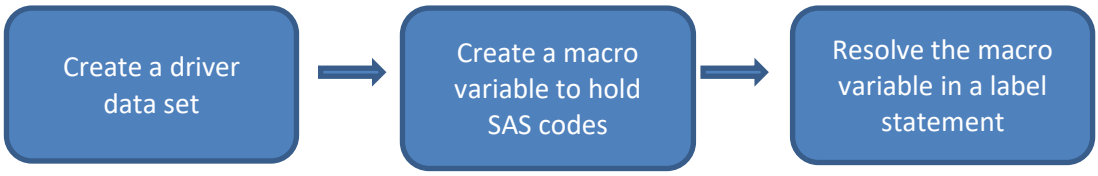


Figure 1. Steps to Create Labels with the Macro Variable Technique.

Display 3 below shows the SAS program, the SAS log, and the SAS output involved in the process. The input sources are:

1. META.SAS7BDAT: the driver data set imported from the specification spreadsheet. It contains two columns. One is variable name (vname), and the other is variable label (label).
2. INTERIM_DATA.SAS7BDAT: an interim SAS data set with all the variables created and the variable labels still missing.

```
* create a macro variable to hold label statements *;
%global dataset_label_list;

proc sql noprint;
  select catx("=", vname, quote(trim(label)))
    into :dataset_label_list separated by " "
    from meta;
quit;
```

VIEWTABLE: Work.Meta

	vname	Label
1	STUDYID	Study Identifier
2	PCHG	% Change from Baseline
3	STATUS	Subject's Status

```
* resolve the macro variable in a data step *;
data final_data;
    set interim_data;
    label &dataset_label_list;
run;
```

```
62   * resolve the macro variable in a data step *;
63   data final_data1;
64       set interim_data;
SYMBOLGEN:  Macro variable DATASET_LABEL_LIST resolves to STUDYID="Study Identifier" PCHG="% Change
            from Baseline" STATUS="Subject's Status"
65       label &dataset_label_list;
66   run;
```

Alphabetic List of Variables and Attributes

#	Variable	Type	Len	Label
2	PCHG	Num	8	% Change from Baseline
3	STATUS	Char	7	Subject's Status
1	STUDYID	Char	10	Study Identifier

Display 3. Create Variable Labels with the Macro Variable Technique.

The first block in Display 3 contains SAS codes that create and then resolve the macro variable. As shown in the display, we start off creating a global macro variable named 'dataset_label_list' using the SQL procedure. During the execution of the SQL procedure, SAS interacts with the META data set to retrieve variable names and variable labels, and stores them in the 'dataset_label_list' macro variable. Lastly, in a simple DATA step, we resolves the 'dataset_label_list' macro variable after a label statement. When the DATA step executes, the label statement will generate variable labels.

The second block in Display 3 is a snapshot from the log file. With the help of the SYMBOLGEN macro option, we will be able to take a visual look at the SAS codes automatically generated by the macro variable 'dataset_label_list'.

The last block is the output from a CONTENTS procedure showing the result: a list of SAS variables and their newly created labels. Note that although the label description text contains special characters such as the percent sign and the apostrophe, they do not trigger errors as they are correctly enclosed in quotation marks.

TECHNIQUE 2: THE CALL EXECUTE ROUTINE

The CALL EXECUTE routine is a DATA step facility. The syntax is very simple:

```
Call execute('argument');
```

The argument here in the syntax can be any SAS code or any variable from a driver data set. All arguments will be executed as regular SAS codes.

The CALL EXECUTE routine builds SAS codes dynamically as DATA step iterates, making it an effective SAS code generator. It allows us to create SAS codes that are completely controlled by the driver data set that it processes.

Figure 2 illustrates the work process of the technique. It is of note that the Call Execute routine has the capacity of combining the following two steps in one DATA step: it builds SAS codes while iterating through the driver data set, then it executes the codes automatically.

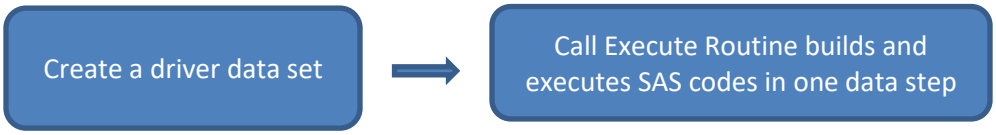


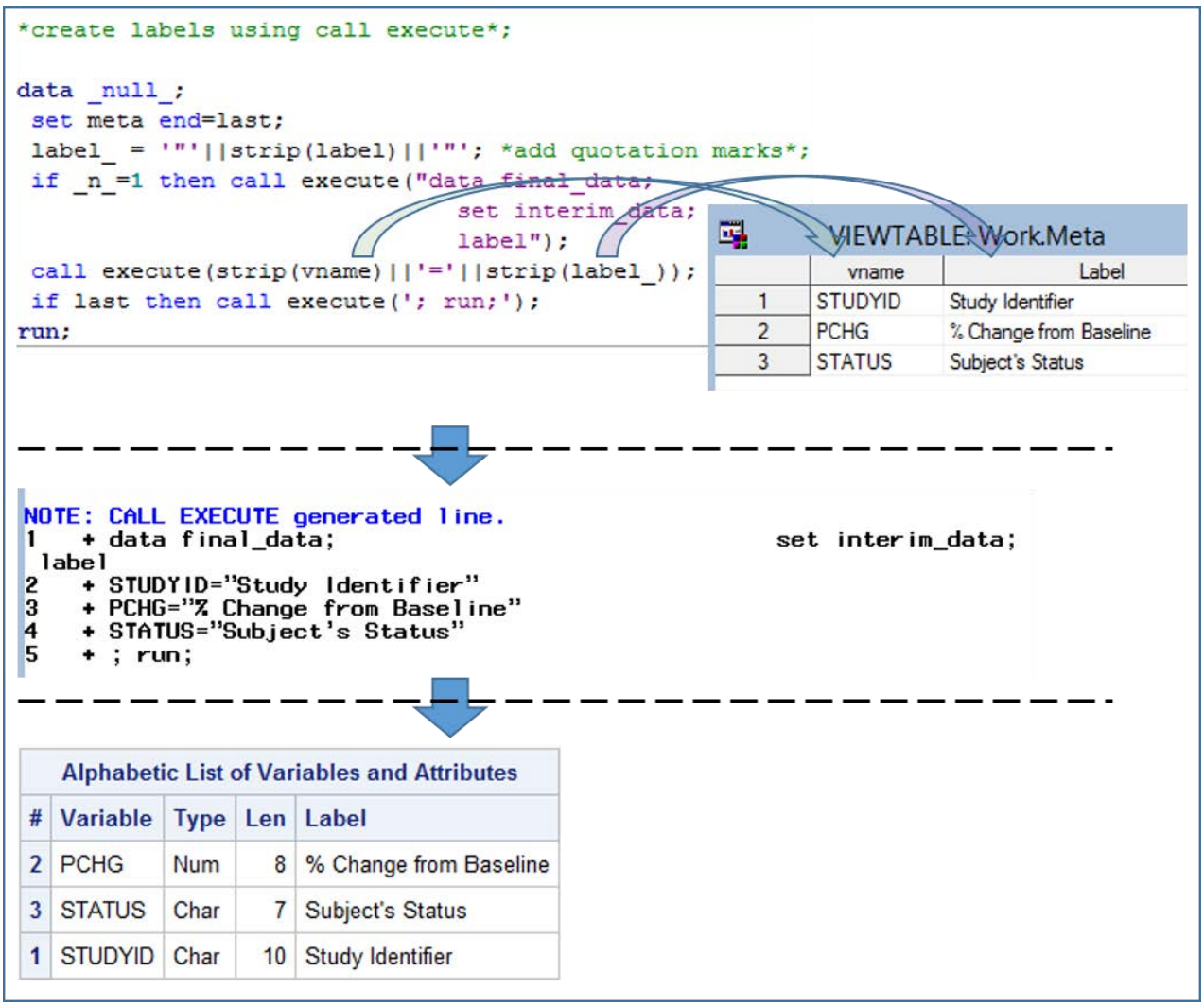
Figure 2. Steps to Create Labels with the CALL EXECUTE Routine.

In Display 4 below, we show the SAS program, the SAS log, and the SAS output involved in the process.

In the first block of Display 4, in a DATA step that reads the driver data set (meta), we use the Call Execute routine three times to generate SAS codes. In the first (opening) and the third (closing) Call Execute routines, the arguments contain simple SAS codes. In the second Call Execute routine, the argument contains variables from the driver data set. It interacts with the driver data set, executes for every observation read from the driver data set, and then dynamically builds code using the variable names and variable labels retrieved from the driver data set.

The second block shows the CALL EXECUTE generated codes in the log. By default, the Call Execute routine will automatically print generated codes to the log.

The last block is the PROC CONTENTS output showing variable names and the newly created variable labels. We can see that the result is identical to the result using the macro variable technique in the previous section.



Display 4. Create Variable Labels with the CALL EXECUTE Routine.

TECHNIQUE 3: THE PUT STATEMENT

The PUT statement has the capacity to write out text, making it possible to generate SAS codes. The third code-generating technique we are going to introduce uses PUT statements to write SAS codes to a file, and then use %INCLUDE to include the codes for execution.

The PUT statement method is the only method that automatically creates a physical SAS program and saves it to a pre-specified file directory with a pre-specified SAS program name.

Figure 3 below demonstrates the work process. This method involves three steps. Firstly we associate a file name to a SAS program. Then we build SAS codes into the associated file using the PUT statement. Finally, we use %INCLUDE to include the generated codes for execution.

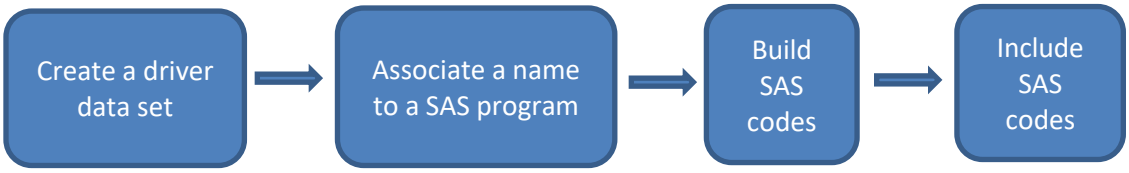
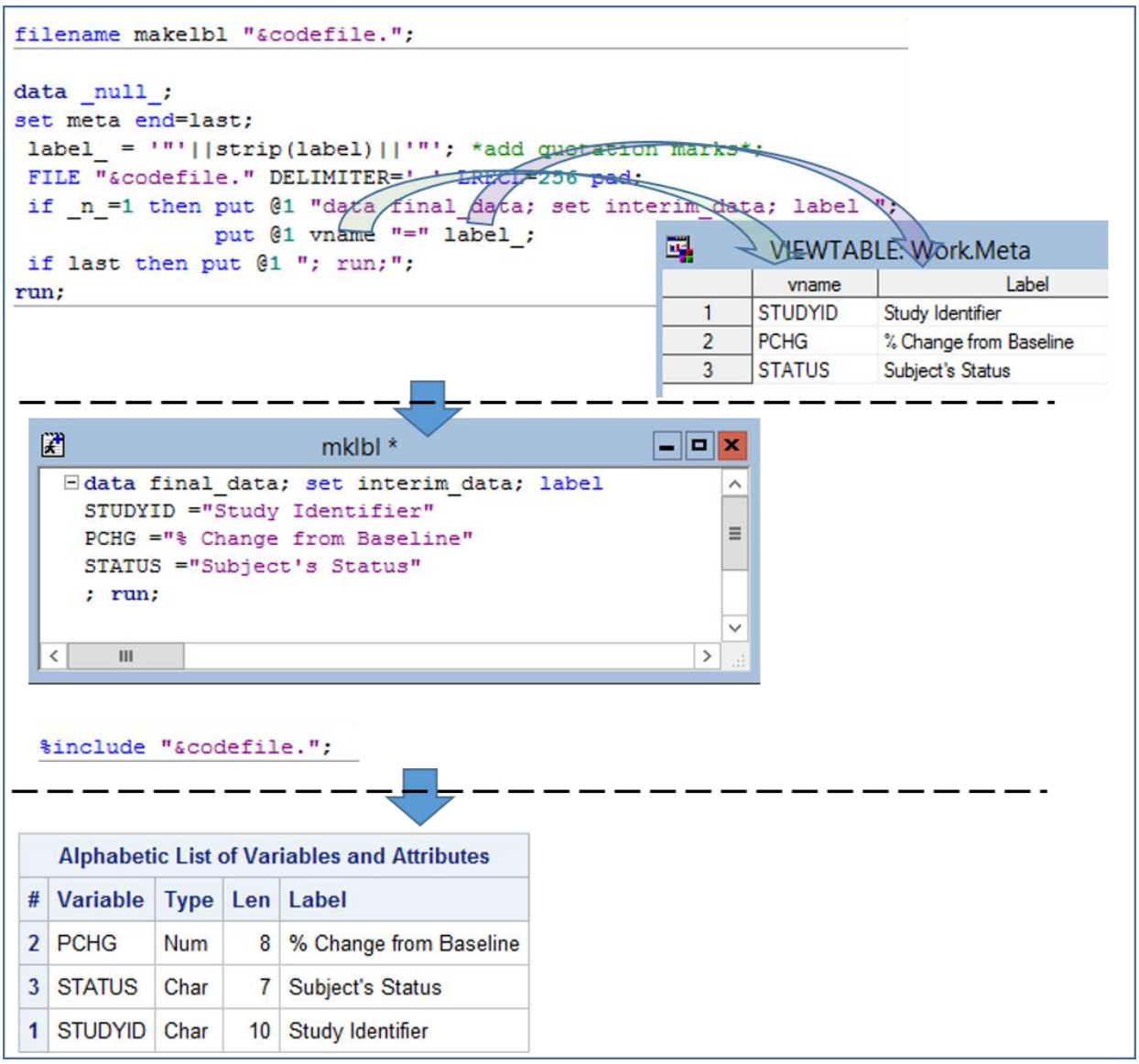


Figure 3. Steps to Create Labels with the PUT Statement Method.



Display 5. Create Variable Labels with the Put Statement Method.

In Display 5, we show the code-generating SAS program, the SAS-generated SAS program, and the SAS output involved in the process.

The first line of the SAS code uses the FILENAME statement to associate a SAS name with an external file. The macro variable &codefile in our example will resolve to a full valid file specification (directory name and file name) referencing the SAS program that we are going to generate.

Following the FILENAME statement, we use a DATA step to interact with the driver data set (meta). We then use a series of PUT statements to write SAS codes, which will be written to the output file specified in a FILE statement.

We can open the associated file to view the generated SAS codes, which is demonstrated in the middle block of the display. The middle block of the display also contains the %INCLUDE which we use to include and execute the SAS program.

Lastly the third block of the display contains the PROC CONTENTS output which is the same as the outputs we generated using the previous two methods.

EXAMPLE 2: CONVERT VARIABLE TYPES AND ATTRIBUTES

In the previous example, we have a specification spreadsheet that serves as a convenient driver data set. In this example, we will demonstrate the techniques using another convenient, easily accessible source of driver data sets – the DICTIONARY tables.

SAS programmers often need to convert variables to a different type: numeric to character or vice versa. In our example, as shown below in Display 6, we aim to convert a number of date variables from the traditional numeric format to the ISO 8601 international standard date format, which is of the character type. For demonstration purpose, we only display the first three date variables: firstdt, seconddt, and thirddt.

old_dates.sas7bdat

	FIRSTDT	SECONDDT	THIRDDT
1	01DEC1964	19FEB2018	09JUL2018
2	11JUN1956	26FEB2018	26FEB2018
3	28JAN1985	09APR2018	09APR2018
4	18SEP1937	18APR2018	18APR2018
5	08FEB1972	30MAY2018	05NOV2018

new_dates.sas7bdat

	FIRSTDTC	SECONDDTC	THIRDDTC
1	1964-12-01	2018-02-19	2018-07-09
2	1956-06-11	2018-02-26	2018-02-26
3	1985-01-28	2018-04-09	2018-04-09
4	1937-09-18	2018-04-18	2018-04-18
5	1972-02-08	2018-05-30	2018-11-05

Display 6. Convert Date Variables from Numeric to Character.

In addition to the conversion of data types shown above, Display 7 below shows that we also want to modify variable names and variable labels to indicate that the variable type is now character. In addition, we also set the length of the character variables to 10.

Variable Attributes in old_dates.sas7bdat

#	Variable	Type	Len	Format	Label
1	FIRSTDT	Num	8	DATE9.	First Date
2	SECONDDT	Num	8	DATE9.	Second Date
3	THIRDDT	Num	8	DATE9.	Third Date

Variable attributes in new_dates.sas7bdat

#	Variable	Type	Len	Label
1	FIRSTDTC	Char	10	First Date (C)
2	SECONDDTC	Char	10	Second Date (C)
3	THIRDDTC	Char	10	Third Date (C)

Display 7. Convert Date Variable Attributes.

DRIVER DATA SET IN EXAMPLE 2

The DICTIONARY tables are a number of SQL tables and views containing valuable metadata information related to SAS libraries, SAS data sets, SAS system options, etc. With the SAS SQL procedure, we can easily retrieve data from the DICTIONARY tables, and use them to write SAS codes with the help of our code-generating techniques.

Display 8 is a partial snapshot of the DICTIONARY.COLUMNS table for our input data set (old_dates.sas7bdat). We will use this table as the driver data set.

	libname	memname	memtype	name	type	length	npos	varnum	label	format
1	WORK	OLD_DATES	DATA	FIRSTDT	num	8	0	1	First Date	DATE9.
2	WORK	OLD_DATES	DATA	SECONDDT	num	8	8	2	Second Date	DATE9.
3	WORK	OLD_DATES	DATA	THIRDDT	num	8	16	3	Third Date	DATE9.

Display 8. Partial Snapshot of the DICTIONARY.COLUMNS Table.

TECHNIQUE 1: THE MACRO VARIABLE METHOD

Firstly, we use a SQL procedure to create a macro variable named 'convert_code' to store the generated SAS codes:

```
proc sql noprint;
  select distinct cat('attrib ', strip(name), 'C length=$10 label="',
                      strip(label), ' (C)";',
                  strip(name), 'C=put(', strip(name), ', yymmdd10.)')
     into :convert_code separated by '; '
  from DICTIONARY.COLUMNS
  where libname='WORK' and memname='OLD_DATES';
quit;
```

Secondly, we resolve the macro variable 'convert_code' in a simple DATA step. In the meanwhile, we turn on the SYMBOLGEN option so that we can review the generated code in the log.

```
options symbolgen;

data new_dates;
  set old_dates;
  &convert_code.;
run;
```

After executing the above codes, SAS will create the new_dates.sas7bdat SAS data set with the desired results, as well as a SAS log. As shown in Display 9 below, the SYMBOLGEN option gives us the opportunity to review the SAS-generated SAS codes in the log.

```
26         options symbolgen;
27         data new_dates;
28          set old_dates;
29          &convert_code.;
SYMBOLGEN:  Macro variable CONVERT_CODE resolves to
attrib FIRSTDTC length=$10 label="First Date (C)";FIRSTDTC=put(FIRSTDT, yymmdd10.);
attrib SECONDDTC length=$10 label="Second Date (C)";SECONDDTC=put(SECONDDT, yymmdd10.);
attrib THIRDDTC length=$10 label="Third Date (C)";THIRDDTC=put(THIRDDT, yymmdd10.)
30         run;
```

Display 9. SAS-generated Codes in the Log.

TECHNIQUE 2: THE CALL EXECUTE ROUTINE

Since CALL EXECUTE is a DATA step routine and the DICTIONARY table can only be accessed with SQL procedures, our first step will be the following SQL procedure. It retrieves the DICTIONARY.COLUMNS data set and saves it in the temporary WORK library so that it can be accessed by SAS DATA steps.

```
proc sql noprint;
  create table columns as
  select * from DICTIONARY.COLUMNS
  where libname='WORK' and memname='OLD_DATES';
quit;
```

Next, we build SAS codes using three CALL EXECUTE routines. The second CALL EXECUTE routine is the one that interacts with the driver data set:

```
data _null_;
  set columns end=last;
  if _n_=1 then call execute('data new_dates; set old_dates;');
  call execute( cat('attrib ', strip(name), 'C length=$10 label="',
                    strip(label), ' (C)"; ',
                strip(name), 'C=put(', strip(name), ', yymmdd10.);')
                );
  if last then call execute('run;');
run;
```

After executing the above codes, we will obtain the same desired results as we did using the previous macro variable technique. By default, the generated codes will be printed to the SAS log as shown below in Display 10.

```
NOTE: CALL EXECUTE generated line.
1   + data new_dates; set old_dates;
2   + attrib FIRSTDTC length=$10 label="First Date (C)"; FIRSTDTC=put(FIRSTDT, yymmdd10.);
3   + attrib SECONDDTC length=$10 label="Second Date (C)"; SECONDDTC=put(SECONDDT, yymmdd10.);
4   + attrib THIRDDTC length=$10 label="Third Date (C)"; THIRDDTC=put(THIRDDT, yymmdd10.);
5   + run;

NOTE: There were 6 observations read from the data set WORK.OLD_DATES.
NOTE: The data set WORK.NEW_DATES has 6 observations and 6 variables.
```

Display 10. CALL EXECUTE Generated Codes in the Log.

TECHNIQUE 3: THE PUT STATEMENT

The third technique is writing text to an associated file using the PUT statements.

Similar to the previous CALL EXECUTE technique, our first step is to retrieve the driver data set – the DICTIONARY.COLUMNS table - for DATA step processing:

```
proc sql noprint;
  create table columns as
  select * from DICTIONARY.COLUMNS
  where libname='WORK' and memname='OLD_DATES';
quit;
```

Secondly, we use the FILENAME statement to associate a SAS fileref with an external file's complete name (directory and file name). This is an optional step where we take advantage of the FILENAME statement so that we can use the SAS fileref in the FILE statement and the %INCLUDE in the next steps.

```
filename cc "\\sasprod\level1\level2\username\convert_code.sas";
```

Next, we use a DATA step to accomplish the following:

1. Read the "columns' driver data set for processing.
2. Use STRIP and CONCATENATION functions to prepare variables for PUT statements.
3. Use the FILE statement to specify the output file for PUT statements.
4. Use four PUT statements to write codes.

The full code for this step is shown below:

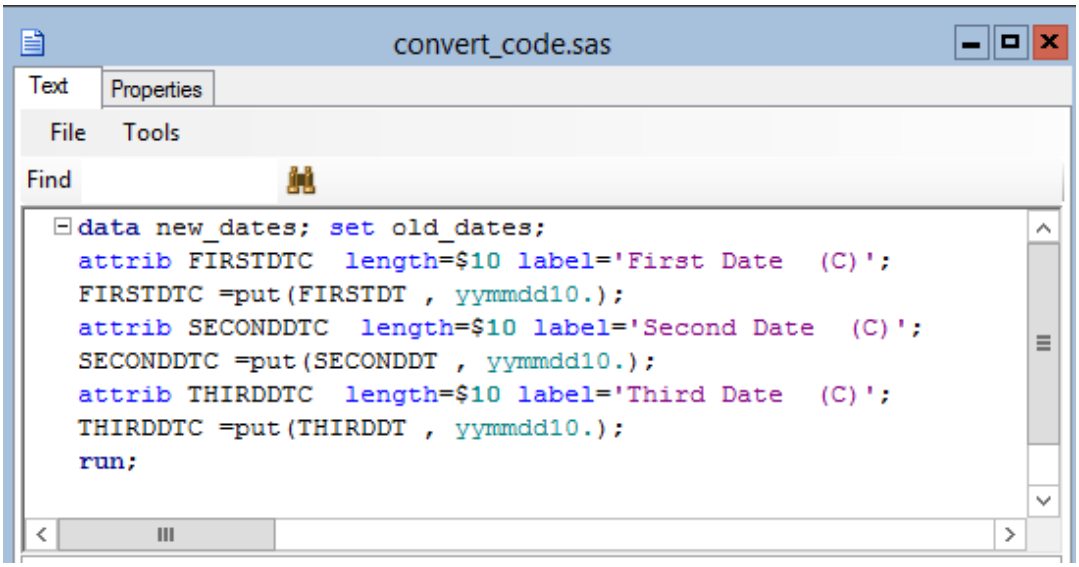
```
data _null_;
 *read the driver data set*;
  set columns end=last;
 *prepare variables in the driver data set*;
  name_=strip(name);
```

```
    label_=strip(label);
    namec=strip(name)||'C';
   *specify the output file*;
    file cc DELIMITER=' ' LRECL=256 pad;
   *use PUT statements to write codes*;
    if _n_=1 then put @1 "data new_dates; set old_dates;";
         put @1 "attrib " namec " length=$10 label='" label_ " (C)';";
         put @1 namec '=put(' name_ ', yymmdd10.);';
    if last then put @1 "run;";
  run;
```

Finally we use %INCLUDE to access and include the output file for SAS execution:

```
  %include cc;
```

After we run the above codes, we will be able to see a SAS program named 'convert_code.sas' in the file directory specified in the FILENAME statement. If we open the SAS program, we will be able to review the SAS code generated by this technique. A snapshot of the SAS program is shown below in Display 11.



Display 11. A SAS Program Generated by PUT Statements.

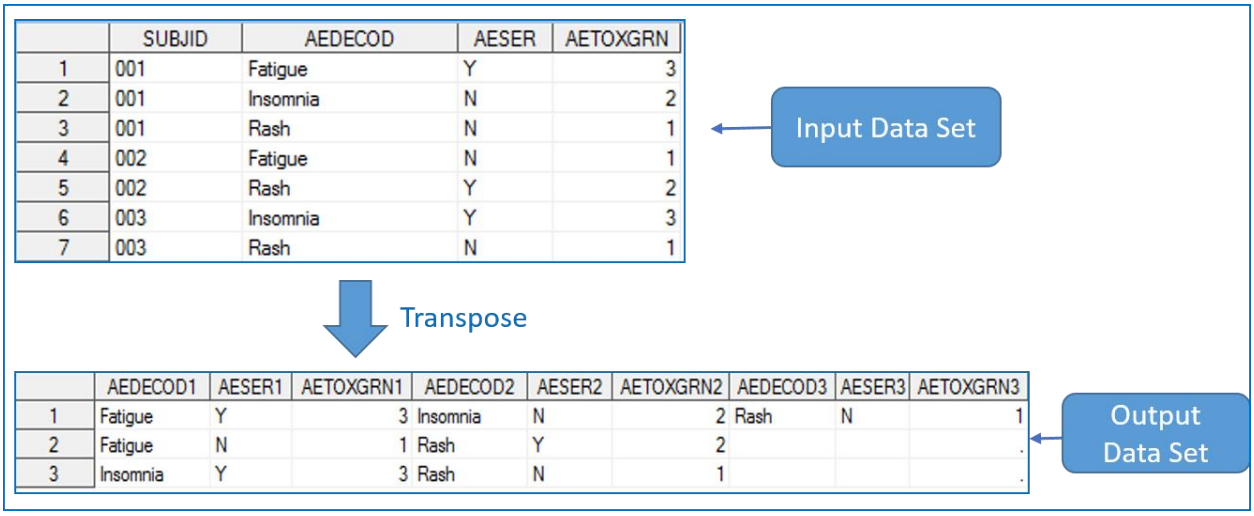
In conclusion, the convenient, easily accessible SAS DICTIONARY tables prove to be a useful source of driver data set for automatic code generators. The DICTIONARY.COLUMNS table is a most popular and widely-used table because it contains valuable information about variable names and attributes. Note that in addition to DICTIONARY.COLUMNS, there are also other data tables containing valuable information such as SAS libraries, SAS data sets, and SAS system options.

In this example, we again demonstrated that all three code-generating codes will be able to accomplish the tasks with a relatively small amount of manual coding. For demonstration purpose, our codes in this example only generated three attribute statements and three assignment statements. In reality, in the cases where we are tasked to program a large number of variable conversions, the techniques above will save us from repetitively typing up numerous lines of codes.

EXAMPLE 3: TRANSPOSE VARIABLES

In the previous examples, we use driver data sets from external sources that are separate from the input data set. In this example, we would like to demonstrate that, in cases where a convenient external source is not available, with some creativity, it is possible to generate a driver out of the input data set to work with the code generating techniques.

In this example, our input data set (AE.SAS7BDAT) is a SAS data set in vertical format, as shown below in Display 12. For illustration purpose, we only display three hypothetical subjects (SUBJID 001, 002, and 003) with three variables related to their adverse events (AEs) in a clinical trial. The goal is to transpose all the AE variables for the subjects to facilitate the creation of a listing output which is in horizontal format.



	SUBJID	AEDECOD	AESER	AETOXGRN
1	001	Fatigue	Y	3
2	001	Insomnia	N	2
3	001	Rash	N	1
4	002	Fatigue	N	1
5	002	Rash	Y	2
6	003	Insomnia	Y	3
7	003	Rash	N	1

	AEDECOD1	AESER1	AETOXGRN1	AEDECOD2	AESER2	AETOXGRN2	AEDECOD3	AESER3	AETOXGRN3
1	Fatigue	Y	3	Insomnia	N	2	Rash	N	1
2	Fatigue	N	1	Rash	Y	2			.
3	Insomnia	Y	3	Rash	N	1			.

Display 12. Transpose Data Set from Vertical Format to Horizontal Format.

DRIVER DATA SET IN EXAMPLE 3

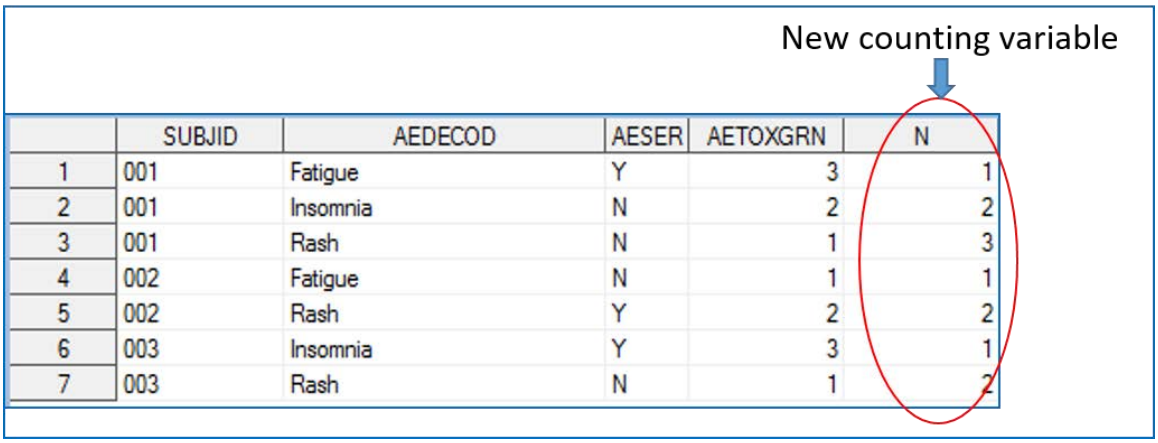
In this example, we are going to manipulate the original input data set by adding a counting variable named 'N'. We will use the data set with the counting variable as driver for our automatic code generators.

We run the following codes to generate the driver data set:

```
proc sort data=AE;
  by subjid aedecod;
run;

data driver;
 set AE;
 by subjid;
 *add a counting variable N*;
 N+1;
 if first.subjid then N=1;
run;
```

Display 13 below is a snapshot of the driver data set we created from the above codes.



	SUBJID	AEDECOD	AESER	AETOXGRN	N
1	001	Fatigue	Y	3	1
2	001	Insomnia	N	2	2
3	001	Rash	N	1	3
4	002	Fatigue	N	1	1
5	002	Rash	Y	2	2
6	003	Insomnia	Y	3	1
7	003	Rash	N	1	2

Display 13. Snapshot of the Driver Data Set with the Counting Variable.

TECHNIQUE 1: THE MACRO VARIABLE METHOD

In order to use the macro variable technique, we create a macro variable named 'newcode' to store the SAS codes.

```
proc sql noprint;
 select distinct catt('driver (where=(n=', N,
                ') rename=(AEDECOD=AEDECOD', N,
                ' AESER=AESER', N,
                ' AETOXGRN=AETOXGRN', N,
                '))')
    into :newcode separated by ' '
    from driver;
quit;
```

We would like to take a look at the new macro variable with the %PUT statement:

```
%put &newcode.;
```

In the log, we will see that the macro variable &newcode stores the following text:

```
driver (where=(n=1) rename=(AEDECOD=AEDECOD1
AESER=AESER1 AETOXGRN=AETOXGRN1))
driver (where=(n=2) rename=(AEDECOD=AEDECOD2
AESER=AESER2 AETOXGRN=AETOXGRN2))
driver (where=(n=3) rename=(AEDECOD=AEDECOD3
AESER=AESER3 AETOXGRN=AETOXGRN3))
```

Display 14. SAS Codes Stored in the Macro Variable.

Finally we resolve the macro variable 'newcode' after a MERGE statement in a DATA step:

```
data ae_transposed(drop=n) ;
 merge &newcode.;
 by subjid;
run;
```

After running the above codes, we will obtain the desired output data set in horizontal format as shown in Display 12.

TECHNIQUE 2: THE CALL EXECUTE ROUTINE

Firstly, we use a SORT procedure with the NODUPKEY option to obtain a data set with distinct numbers in the variable N:

```
proc sort data=driver(keep=n) out=n nodupkey;
  by n;
run;
```

Then we use the CALL EXECUTE routine three times in a DATA step to build codes:

```
data _null_;
  set n end=last;
  if _n_=1 then call execute('data ae_transposed(drop=N); merge ');
   call execute(catt('driver(where=(n=', N,
                     ') rename=(AEDECOD=AEDECOD', N,
                     ' AESER=AESER', N,
                     ' AETOXGRN=AETOXGRN', N,
                     '))'));
  if last then call execute('; by subjid; run;');
run;
```

After execution, the codes will generate the same desired output as shown earlier in Display 12. In the log, we will see the CALL EXECUTE generated codes as shown below:

```
data ae_transposed(drop=N); merge
driver(where=(n=1) rename=(AEDECOD=AEDECOD1 AESER=AESER1 AETOXGRN=AETOXGRN1))
driver(where=(n=2) rename=(AEDECOD=AEDECOD2 AESER=AESER2 AETOXGRN=AETOXGRN2))
driver(where=(n=3) rename=(AEDECOD=AEDECOD3 AESER=AESER3 AETOXGRN=AETOXGRN3))
; by subjid; run;
```

Display 15. CALL EXECUTE Generated Codes in the Log.

TECHNIQUE 3: THE PUT STATEMENT

Similar to the previous two examples, we can accomplish the task easily with the PUT statement method.

Same with the previous technique, the first step is to create a data set with distinct numbers in the variable N:

```
proc sort data=driver(keep=n) out=n nodupkey;
  by n;
run;
```

Secondly, we associate a SAS fileref with an external file's complete name including a full directory and a SAS program name:

```
filename tc "\\sasprod\level1\level2\username\transpose_code.sas";
```

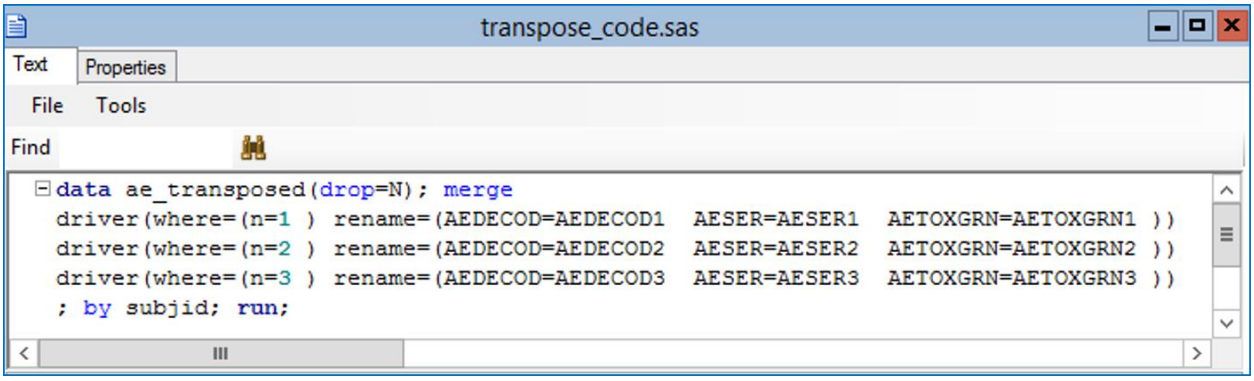
Then we build the codes using the PUT statements and write the codes to the external file. We use the SAS fileref 'tc' in the FILE statement:

```
data _null_;
 set n end=last;
  N=strip(N);
  file tc DELIMITER=' ' LRECL=256 pad;
  if _n_=1 then put @1 "data ae_transposed(drop=N); merge ";
                put @1 "driver(where=(n=" N ")
    rename=(AEDECOD=AEDECOD" N " AESER=AESER" N " AETOXGRN=AETOXGRN"
    N ")) ";
  if last then put @1 "; by subjid; run;";
run;
```

Lastly, we use %INCLUDE to include the external SAS program we just generated. The SAS program is referenced by the fileref 'tc':

```
%include tc;
```

If we navigate to the file directory specified in the FILENAME statement, we will find a newly generated SAS program named 'transpose_code.sas'. A snapshot of the SAS program is displayed below.



Display 16. A SAS Program Generated by the PUT Statement Method.

PROGRAMMING TIPS

A common feature of the three techniques is the creation, concatenation, and manipulation of text strings which make up the SAS codes. In this section, we will share a few programming tips to help avoid some potential pitfalls commonly encountered during text manipulation.

1. Quotation marks:

   In our first two examples, the generated codes contain quotation marks. In the first example, the generated codes also contains special characters such as the apostrophe. For example:

   status="Subject's Status"

   In the first example, quotation marks are required for both the Macro Variable method and the PUT Statements method to enclose the label variable. Without quotation marks, the generated codes will hang while execution. Although quotation mark is optional for the CALL EXECUTE routine, it is recommended in order not to lose apostrophe marks in the labels.

   Here are a few examples of adding quotations shown in previous sections:

```
select catx("=", vname, quote(trim(label))) into :dataset_label_list
label_ = '"'||strip(label)||'"';
put @1 "attrib " namec " length=$10 label=' label_ " (C)';";
```

2. %NRSTR quoting function:

   If we use the CALL EXECUTE method, and the generated codes contain non-macro language constructs for assigning macro variables during run time (e.g., call symput or proc sql into:), the creation of the macro variables will be delayed until the execution is finished. In this case, we recommend using the %NRSTR quoting function around the argument to Call Execute in order to avoid errors on unresolved macro variable references.

3. Text concatenation functions:

   A good understanding of the concatenation functions is necessary in order to write SAS from SAS. Please note the following default behaviors of the four concatenation functions.

   - CATS: removes both leading and trailing blanks from each string before concatenation.
   - CATX: removes both leading and trailing blanks from each string before concatenation; in addition, a delimiter to be added between the strings is specified in the first function parameter.
   - CATT: removes only trailing blanks from strings to be joined
   - CAT or the traditional double bars ||: does not remove any blank, leading or trailing

   Although the ability to automatically remove blanks come in handy in many situation, for our purpose of writing SAS codes, leading and/or trailing blanks are often necessary. We need to be aware of situations where we should use CAT or the traditional double bars in order to keep, instead of remove, the blanks. For example, the following code-generating codes cannot use CATS or CATX because we need the leading blanks (in yellow highlight).

```
proc sql noprint;
  select distinct catt('driver (where=(n=', N,
                ') rename=(AEDECOD=AEDECOD', N,
                ' AESER=AESER', N,
                ' AETOXGRN=AETOXGRN', N,
                '))')
    into :newcode separated by ' '
    from driver;
quit;
```

4. Space-stripping functions

   Since it is possible for the variables in the driver data set to contain leading and/or trailing blank spaces, we rely on space-stripping functions such as TRIM() and STRIP() to prepare the variables. In the first two techniques (macro variable and Call Execute), we can embed these functions in the code-building sections. For example:

```
proc sql noprint;
  select distinct cat('attrib ', strip(name), 'C length=$10
  label="', strip(label), ' (C)";', strip(name), 'C=put(',
  strip(name), ', yymmdd10.)')
  into :convert_code separated by '; '
```

By contrast, for the third technique (PUT Statements), we need to prepare the variables before feeding them into the code-building sections (that is, after the PUT statements). For example:

```
data _null_;
 set columns end=last;
 name_=strip(name);
 label_=strip(label);
 namec=strip(name)||'C';
 file cc DELIMITER=' ' LRECL=256 pad;
 if _n_=1 then put @1 "data new_dates; set old_dates;";
 put @1 "attrib " namec " length=$10 label='" label_ " (C)';";
 put @1 namec '=put(' name_ ', yymmdd10.);';
 if last then put @1 "run;";
run;
```

Perform STRIP() and Concatenations before the code-building sections

COMPARISON

All the three techniques demonstrated in this paper are equally capable of generating SAS codes automatically by interacting with a driver data set. Selection of a code-generating technique for your project depends on a number of factors including your comfort level with each technique. This paper will discuss the simplicity, flexibility, and system efficiency of each technique to aid in your decision.

The Macro Variable method and the Call Execute method both require similar amount of coding. By contrast, the PUT Statement method requires a slightly larger amount of coding. For example, in our example of creating variable labels, the macro variable technique takes 26 words and 201 characters to do the job, the Call Execute technique takes 25 words and 192 characters, while the PUT Statement technique takes 41 words and 248 characters.

The Call Execute technique has the advantage of combining SAS code generation and execution in one data step, making a more simplified process. By contrast, the PUT Statements has a more complex process as it involves three distinct steps including file association, code generation, and code inclusion and execution.

The Macro Variable technique is subject to the limitation that the length of macro variables cannot exceed 65,534 characters. By contrast, the other two techniques do not have such limitations.

The PUT Statement method is the only method that actually generates a physical SAS program which we can open, review, and even modify. As a result, the technique gains a significant amount of flexibility and clarity by sacrificing simplicity.

To evaluate and compare system efficiency of the three techniques, we use the codes in the first example (automate variable label creation), and run each technique 20 times, 10 times in interactive mode and 10 times in batch mode, on an interim SAS data set with 121 variables and 40,000 records during various times when host system resources are unlikely to be in use. We use the following codes as stopwatch placed in the beginning and the end of each SAS run.

```
%PUT Start: %SYSFUNC(datetime(),datetime23.3);
%let Start = %SYSFUNC(time());
%PUT End: %SYSFUNC(datetime(),datetime23.3);
%let End = %SYSFUNC(time());
```

Then we use the following formula to calculate real run time of each SAS run.

```
Runtime = %SYSFUNC(round(%SYSEVALF(&end-&start),.001))
```

The results are illustrated in Table 1. Within the interactive mode, the Macro Variable method runs more efficiently compared to the other methods. Within the batch mode, the Call Execute method is the most efficient.

The Call Execute method runs faster in the batch mode, but slower in the interactive mode, which is most likely due to the fact that the Call Execute routine slows down when writing generated codes to the log window in the interactive mode.

Mode	Method	Mean	SD	Minimum	Maximum
Interactive	Macro Variable	0.092	0.004	0.083	0.098
	Call Execute	0.120	0.004	0.111	0.125
	Put Statement	0.107	0.007	0.101	0.124
Batch	Macro Variable	0.082	0.008	0.077	0.104
	Call Execute	0.079	0.002	0.077	0.084
	Put Statement	0.089	0.002	0.087	0.094

Table 1. Comparison of Real Run Time in Seconds

It is of note that each SAS port yields different performance statistics based on the host operating system and the network resources. Therefore, the measurements above only serve the purpose of comparing efficiency for the techniques in discussion.

CONCLUSION

In this paper, we chose three simple examples to demonstrate the code-generating techniques. We showed that there are many potential sources of driver data sets. It can be an external specification file, the easily-accessible DICTIONARY tables, or even the input data set itself. Through interactions with the driver data sets, the code generators prove to be reliable automation solutions for repetitive tasks or ever-changing data. They will significantly cut down on programming time. Higher efficiency will also lead to better quality. We demonstrated in this paper that all the three techniques will generate the same results, with each technique having its own advantages and disadvantages.

The code generating techniques have a wide array of applications for both simple and complex programming tasks. A review of the SAS literature has found the following successful applications in addition to the demonstration examples in this paper:

1. Import a variety of raw data files into SAS with one code-generating SAS program
2. Automatic adverse event toxicity edit check in clinical programming
3. Specification driven lab CTC grading in clinical programming
4. Meta data driven mapping for CDISC compliant clinical SAS data sets
5. Assign variable values without writing hundreds of 'IF… THEN…' statements.

The code-generating approach has served us well. It can be applied to a wide range of programming tasks, with the potential of eliminating repetitive programming chores for SAS programmers. Therefore, when encountered with repetitive tasks or non-static data, programmers are encouraged to consider the code generating approach as an automation option.

REFERENCES

Batkhan, L. 2017. "SAS Blogs: CALL EXECUTE Made Easy for Data-Driven Programming." Accessed July 26, 2018. https://blogs.sas.com/content/sgf/2017/08/02/call-execute-for-sas-data-driven-programming/

Dai, Y and Li, Y 2018. "SAS® automation techniques – specification driven programming for Lab CTC grade derivation and more" Proceedings of the PharmaSUG Conference 2018. Cary, NC: SAS Institute Inc. Available at https://www.pharmasug.org/proceedings/2018/BB/PharmaSUG-2018-BB12.pdf

Fraeman, K.H. 2010. "Let SAS® Write and Execute Your Data-Driven SAS Code." Proceedings of the Northeast SAS User Group Conference 2010. Cary, NC: SAS Institute Inc. Available at https://www.lexjansen.com/nesug/nesug10/cc/cc12.pdf

Gau, L. 2004. "Write SAS® Code to Generate Another SAS® Program A Dynamic Way to Get Your Data into SAS®" Proceedings of the SAS User Group International 2004, Cary, NC: SAS Institute Inc. Available at https://support.sas.com/resources/papers/proceedings/proceedings/sugi29/175-29.pdf

Ivanushkin, V. 2019. "One More Paper on Dictionary Tables and Yes, I Think it Is Worth Reading." Proceedings of the PharmaSUG Conference 2019. Cary, NC: SAS Institute Inc. Available at https://www.pharmasug.org/proceedings/2019/BP/PharmaSUG-2019-BP-286.pdf

Shan, X. 2015. "Transpose Dataset by MERGE." Proceedings of the SAS Global Forum 2015, Cary, NC: SAS Institute Inc. Available at http://support.sas.com/resources/papers/proceedings15/

Zhuo, Y. 2018. "Automate Repetitive Programming Tasks: Effective SAS® Code Generators." Proceedings of the Western Users of SAS Software 2018, Cary, NC: SAS Institute Inc. Available at https://www.lexjansen.com/wuss/2018/73_Final_Paper_PDF.pdf

Zhuo, Y. 2019. "End of Computing Chores with Automation: SAS® Techniques That Generate SAS® Codes." Proceedings of PharmaSUG Conference 2019, Cary, NC: SAS Institute Inc. Available at https://www.pharmasug.org/proceedings/2019/BP/PharmaSUG-2019-BP-255.pdf

ACKNOWLEDGEMENTS

Many thanks to the 2020 SAS Global Forum Conference team for the invitation to present this paper.

CONTACT INFORMATION

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

Yun (Julie) Zhuo
PRA Health Sciences
ZhuoYun@prahs.com
julie.zhuo@gilead.com