# PROGRAMS THAT WRITE PROGRAMS: CREATING CODE GENERATORS IN SAS®

Michael C. Harris
Amgen Inc.
Thousand Oaks, California

## Abstract

SAS code generators are a special class of programs that automate the creation of source code for PROC steps, DATA steps and host-specific procedural languages, such as the DOS batch language or Unix scripts. They may depend on predefined data dictionaries for input, or may extract information from non-SAS files whose general structures are documented, but whose actual content may be application-dependent. Database files are one example of such files. This paper discusses issues involved in planning, implementing, testing and debugging code generators written in the SAS DATA step language. Examples of different approaches are presented, and points to consider when packaging a code generator in SAS macro language are touched upon.

## Why Write a Code Generator?

In general, computer programs are written for a few good reasons. We may want to automate an error-prone process that was previously performed manually. Perhaps a task has gotten so large or complex that it can no longer be accomplished in a timely fashion. Or maybe a task has been identified as highly structured and repetitive, but human resources are not available to perform it. In many cases several factors influence the decision to automate a process. Determining whether a code generator is the right tool for the job, then, is essentially the same as determining whether any other type of program should be written. The only difference is that it is a programming task that we seek to automate.

Certain benefits may accrue from automating the creation of programs. You can be certain that the generated code will consistently meet standards that have been previously agreed upon. You may find that you can automatically generate documentation that was previously written manually or perhaps not at all. Finally, through thorough analysis of the task to be automated, you may come to a fuller understanding of the task and as a direct result write more robust and maintainable code.

## Implementation

Before you can write any program, you must know exactly what it is supposed to do. In the case of a code generator, you must know exactly what kind of code it will write, though you may not (and need not) know the details of every program it will write. A good first step is to write a template program that looks like what you want your code generator to produce. Consider a simple example: You want to PROC PRINT selected variables from all of an unknown number of data sets residing in the same dedicated SAS data library. You could run a PROC CONTENTS to discover the contents of the library, then write as many PROC PRINTs as necessary to print the data sets. All of your PROC PRINTs would look something like the following.

```
proc print data=mydata;
  var age sex race;
run;
```

This is fine if there are only a few data sets to print, but suppose there are many, or they don't always have the same names. Here is one way to implement a simple code generator.

- Run a PROC CONTENTS, saving the output to a data set by using the OUT= option.
- Write a DATA _NULL_ step which

SETs the saved CONTENTS output and writes PROC PRINT steps to an external file. The code might look something like the following.

```
/* find all data sets in library */
proc contents data=sasdata._all_
      memtype=data
      out=contents;
run;

/* write proc print statements */
data _null_;
  length cmd $ 80;
  file 'printer.sas';
  set contents;
  by memname;
  if(first.memname) then do;
    cmd='proc print data=' ||
    memname || ';';
    put cmd;
    cmd='var age sex race;';
    put cmd;
    cmd='run;';
    put cmd;
  end;
run;

/* run generated code */
%include 'printer.sas';
```

The PRINT procedure could just as easily have been any other SAS procedure, such as UNIVARIATE or FREQ. The point of this simple example is that we need to know neither the names nor the number of data sets in the library of interest, and indeed they can change each time. We must, however, be certain that the variables we want to print will always be present.

## A Dictionary-Based System

Let's consider a more complex and realistic example. Assume that our organization collects and manages clinical study data in a relational database management system (RDBMS) and the data management group periodically provides us with dumps of selected tables as ASCII files delimited with bar characters (|). There may be as many as forty tables and each table may contain as many as thirty variables. Furthermore, we want to permanently associate formats with the data, we want to explicitly assign lengths, and since the raw data changes frequently, we want the flexibility to create either permanent ordinary SAS data sets or permanent data step views to save storage space by reading the raw data in place. Also assume that from time to time a database analyst may add, modify or drop columns from the RDBMS. Clearly, if we chose to manually create the data step code to process the raw data files, we would have a significant code management problem. Each time a variable was added, dropped or modified it would affect the position of every variable that followed and a tedious code revision would be necessary. Our solution is to build a small data dictionary-based system to write data step code for us. The data dictionary will be an ordinary SAS data set and we will perform most of our management with a simple FSEDIT screen. We want to use formatted input, but don't want to manually determine SAS informats. Therefore we will write small programs that scan the raw data files, determine the length of each field, build informats and populate a variable in the data dictionary. The dictionary will contain variables describing protocol names, SAS data set names, SAS variable names, formats, data types, informats, labels, and position in the data set. Note that we could add variables or create another data set containing additional information to help automatically document the system.

Here is one way of implementing a code generator to meet the requirements we have set forth. For flexibility the code is packaged in a SAS macro. To save space, italicized pseudocode will be used for trivial sections.

```
/* define a macro called codegen
arguments are: protocol, sasds (the
data set to be created), dictlib
(data library holding the
dictionary), dslib (library to hold
created data set), inp (raw data
file), pgm (name of generated
program) and view (a flag indicating
that a data step view is to be
created)
*/
%macro codegen(protocol=. sasds=,
dslib=, inp=, pgm=, view=Y);
%let subsetting vars(%upcase
&vars);

/* subset dictionary to get table
and study of interest */
```

```
proc sort
data=&dictlib..dict(where=(protocol=
"&protocol" and sasds="&sasds"))
out=dict;
by order;
run;

/* determine number of vars in table
for array dimensioning with
pre-defined nobs macro */
%nobs(dict, numvars)

/* define filename to write to */
filename pinc "&pgm";
```

At this point we are going to read the first line of the raw data file and build a set of arrays to gather information we will need to write a program. Since SAS limits us to 200 byte character variables, we run the risk of not reading all of the line if we use one maximum-length variable. Instead, we will read all of the first line into one-byte character variables. We know that no data lines will exceed 512 bytes, so we need that many variables in our input statement. We define a macro to relieve us of the tedium of writing the input statement,and to allow for longer input lines in the future with only minor changes in the code.

```
%macro readit;
  %do i=1 to 512;
    @&i v&i $1.
  %end;
%mend readit;
```

Now we will write the DATA _NULL_ step that is the heart of the code generator. We will combine information gleaned from the first line of the raw data file with that from the data dictionary and with PUT statements write a DATA step program.

```
data _null_;
   length v1-v512 $1;
   retain arndx 2 num 1;
   /* define array to hold raw data
   line */
   array _inbuf{*} v1-v512;
   /* array to hold positions */
   array _pos{&numvars};
   /* to hold formats */
   array _fmt{&numvars} $ 15;
   /* to hold lengths */
   array _len(&numvars) 3;
   /* to hold types */
   array _type{&numvars} $ 1;
   _pos{1}=1;
   infile "&inp" missover lrecl=512
   pad;
if _n_=1 then do;
   input %readit;

  /* build array of starting pos */
  do strpos=2 to dim(_inbuf);
    if (num le  (&numvars-1)) then
      do;
      if(_inbuf{strpos}='|' then do;
         _pos{arndx}=strpos+1;
         arndx+1;
         num+1;
      end;
    end;
  end;

  file pinc;

PUT standard program header lines to
document the generated code;

 put "data &dslib..&sasds ";
 %if &view=Y %then %do;
    put "/ view=&dslib..&sasds";
 %end;
 put ';';
 put "infile '&inp;";
 put 'input';
end;
/* read the dictionary */
do i=1 to dim(_pos);
   set dict end=eof;
   put "@" _pos{i} sasname informat;
   _fmt{i}=format;
   _len{i}=sasname;
   _type{i}=translate(type,'$','C',
   '', 'N');
end;
```

At this point, the input statement has been written and format and length information has been stored in arrays. We will use that information to write length statements for numeric variables, and then write format statements.

```
if (eof) the do;
   put 'length ';
   do i=1 to dim(_pos);
      if(_len{i}^=. and _type{i}=''
      then
      put _name{i} _type{i} _len{i};
end;
put ';';

put 'format ';
do i=1 to dim(_pos);
   if(_fmt[i}^=' ') then
       put _name{i} _fmt{i};
end;
put 'run;';
end;
```

```
run;
%mend codegen;
```

That's all the code we need to write data step programs based on our dictionary.

Code generators can be used to simplify the task of reading binary files of known format, such as dBASE® files (1). The implementation of such programs consists, in general, of two phases. The first relies on our knowledge of the general structure of the file type and is characterized by information gathering. Typically, a header of well-defined structure is scanned to learn general information about the file, such as the number of records it contains and the length of the header itself. But before any information gathering takes place, the file should be scanned for a signature identifying it as the type of file we think it is. The signature might be a binary value at a known offset into the file, or perhaps a text string. If the file is not valid, the user must be notified and no further processing should take place. If the file is valid, information about it can be stored in a SAS data set, in macro variables, or in arrays as appropriate. In the second phase, a DATA _NULL_ step is used to extract application-specific data necessary to write a DATA step program. In particular, we need to know variable names, data types and SAS informats. We must develop algorithms for creating valid SAS variable names where the native database names are not usable, and gracefully handle data type conversions to fit the SAS model of character and numeric variables. If conversions are not possible, as with memo fields or binary large objects, it is necessary to write messages to the log or to error files documenting the loss of data. When reading binary files, there is no substitute for accurate documentation of the general file type. If the file type is proprietary and subject to change, consideration must be given to scanning for a version or revision level identifier within the file and acting on it appropriately. You may find it necessary to periodically upgrade your code to support new features. But if it is necessary to regularly convert binary files to SAS data sets, the inconvenience of maintaining a code generator will probably outweigh the difficulty of writing and testing custom programs manually.

## Testing and Debugging

Code generators have a general structure that looks like the following.

```
do some processing to gather info;

data _null_;
   put lines to a file plugging in
   gathered info;

%include generated code;
```

Testing, therefore, begins with inspecting the output of each part of the model. If SAS procedures are used to gather information, the output of those procedures must be examined and verified before coding the DATA _NULL_ that actually writes code. If a DATA _NULL_ step is used to gather information, you will need to write the values of macro and data step variables it creates to the log for inspection. Obviously, well designed test data can be quite useful at this stage.

Debugging a DATA _NULL_ step that writes code is probably the most difficult task of all. The reason is twofold. First, the step must be robust enough to respond to error conditions without writing incorrect code. Thus, it must be stressed in ways that may not be immediately obvious. Second, it must write syntactically correct code that may possess its own complex logic. In other words, a code generator may be required to execute its own complex logic in order to write complex code. A thorough understanding of what the code generator is supposed to do is essential. If you can't write error-free code yourself, you will not be able to write a code generator to do the same job. Desk checking generated code is a good way to identify problems that are common to any kind of program. Quoting problems, missing or misplaced semicolons, misplaced blanks and simple typographic errors are things to look for before submitting generated code.

The next stage of testing involves executing your generated code and reviewing all

output that it generates. For example, the DATA step generator example described earlier creates data step views or ordinary SAS data sets. To assure ourselves that it is functioning properly, we would have to create both types of output and use a procedure such as PRINT or FSVIEW to check for correctness. We would also inspect the log and make sure that the correct number of records were read, the correct number of observations were created and no invalid values were unexpectedly generated. We would need to prove that we were reading data at the correct column locations and that our SAS informats and formats were correct.

In a complex application, the testing and debugging process will be iterative. Once the code generator executes without apparent errors, running its output will probably reveal errors in logic or in the files on which it depends. In a dictionary-based system, incorrect entries will cause faulty code to be written. If you wrote a system that looked up file names from a data set to write INFILE statements, you might discover errors in that data set at run time. It would be wise to consider pre-processing that data set to ensure that it contained valid file names. As a general rule, the more complex the system, the more checking you should do outside of the actual code generator to guarantee trouble-free operation.

## Macro Considerations

Values can be passed to SAS programs by using %LET statements to set global macro variables or by packaging the entire program in a SAS macro program that is invoked with appropriate arguments. The first method is best used where the macro variables are used only for substitution of simple parameters such a variable or data set names. Providing a macro wrapper for a code generator is much more flexible, as it allows not only for simple substitution, but also for complex flow control through conditional code generation. Placing the macro in an AUTOCALL library simplifies its use by making it look much like a function call and may improve efficiency as well.

For simple applications it is easy to write a macro program in one pass. For more complex situations, you may find it easier to stage the implementation as follows.

- Determine the general structure of the program.
- Write ordinary SAS code.
- Debug and refine until you are satisfied.
- Modify code that requires macro substitution.
- Modify code that requires flow control based on macro variables.
- Debug, debug, debug.

Consider implementing a debugging flag in your code that is turned off by default.

```
%macro mymac(arg1=, arg2=,
  debug=OFF);
  %if %upcase(&debug=ON) %then %do;
    options mprint symbolgen;
  %if %upcase(&debug=NO) %then %do;
    options noprint nosymbolgen;
  %end;

  perform necessary processing;

%mend mymac;
```

It isn't necessary to include the debugging flag in the argument list on invocation.

## Conclusion

Using code generators to automate repetitive or difficult but well defined programming tasks can result in more productive use of a programmer's time. Code generators are like other programs in that there are many ways of reaching the same goals. You may know other ways of implementing the concepts set forth in this paper or perhaps have requirements that haven't been addressed. For an excellent discussion of programs that modify themselves at run-time, refer to chapter 12 of *Professional SAS Programming Secrets* by Rick Aster and Rhena Seidman,

published by Windcrest/McGraw-Hill.

## References

1. Harris, Michael. "Using the SAS® Data Step Language to Read dBASE III Files": NESUG 92 Conference Proceedings (1992): 217-223.

## Acknowledgements

SAS is a registered trademark of SAS Institute Inc.

dBASE is a registered trademark of Borland/Ashton-Tate.