

Program, Code Thyself: Techniques for Writing Programs that Write Programs.

Renato G. Villacorte, Fairbank, Maslin, Maullin & Associates, Santa Monica, CA

Abstract

"Meta-programming" makes repetitive procedures and routine coding more efficient. With simple Base SAS tools, one can effectively write programs that determine unknown data file names, variable names, or other parameters and construct tailor made programs to process or analyze those objects. One can also use these tools to develop subroutines that require minimal user input or guidance. Examples of these coding techniques include how to read an MS Windows folder's filenames into a SAS dataset, how to extract Proc results into datasets, and how to make SAS write programs based on these results. Note: All examples and expertise offered is based on the SAS for Windows product operating in a stand-alone PC environment.

Introduction

Senior programmers are often faced with the challenge of training novice programmers while maintaining the continuous productivity of the team. An example of this problem was the introduction of SAS software to a data processing team with no formal programming training. A meta-programming approach produced data processing files that were used to continue productivity and to serve as training tools. Programs that gathered an individual project's parameters and wrote project-specific syntax eliminated the need for the experienced SAS programmer to write individual project programs. Instead, the individual programs were written automatically and then inspected and used by the novice programmers.

Description of the Example

The following describes the techniques used to develop a simple meta-program that writes a data importing program. The tasks for the meta-program include:

- Identifying the project data and dictionary file names.
- Converting data dictionary file into a readable text file.
- Reading the data dictionary information, variables and column assignments, into a SAS dataset.
- Writing a Data step routine to read the project data file according to the data dictionary information.

The only parameter required in the macro is the project number, i.e. 991-9901, which is also the name of the data folder. For clarity, the macro variable that represents the project number is resolved in the sample code.

Parameter Gathering Techniques

DOS Commands

Since the overall goal is to import data into SAS, efforts must be made to obtain information from non-SAS objects. In the Windows environment this often requires the use of DOS command line entries that generate the necessary information. In the example, a DOS command is also necessary to translate a coded dictionary file into a plain text file.

Get File Names

The first task is to read the contents of the project folder and identify the names of the project data and dictionary file names. According to the project template, data files have a *.dat file extension and dictionary files have a *.dic extension. We can easily find these names with a simple DOS directory command.

```
OPTIONS noxwait;
%SYSEXEC CD F:\991-9901;
%SYSEXEC DIR *.dic > F:\991-9901\dicname.txt /b;
%SYSEXEC DIR *.dat > F:\991-9901\datname.txt /b;
```

Here, SYSEXEC is used to start a DOS command window, change the current directory to the project folder and perform directory searches for files with the desired extensions. The additional parameters on the DIR command save the results of the directory search to a text file. Once in the text file, it is a simple matter to import the names into a dataset.

```
PROC IMPORT OUT=DICNAME
  DATAFILE= "F:\991-9901\dicname.txt"
  DBMS=DLM REPLACE; DELIMITER='00'x;
  GETNAMES=NO; DATAROW=1;
PROC IMPORT OUT=DATNAME
  DATAFILE= "F:\991-9901\datname.txt"
  DBMS=DLM REPLACE; DELIMITER='00'x;
  GETNAMES=NO; DATAROW=1;
RUN;
```

These import statements result in two datasets with a single record and observation equal to the file name we require. To complete our task of getting the file names, we need to assign the values to macro variables for later use.

```
DATA _NULL_ ; SET DICNAME;
  NAME= SUBSTR(VAR1, 1, (LENGTH(VAR1)-4));
  CALL SYMPUT ("DICFILE", NAME);
DATA _NULL_ ; SET DATNAME;
  NAME= SUBSTR(VAR1, 1, (LENGTH(VAR1)-4));
  CALL SYMPUT ("DATFILE", NAME);
RUN;
```

Now, we have the names of each file ready to use as a macro variable in further programming.

Convert Dictionary

Since the data dictionary is delivered as a proprietary coded file, we need to execute a DOS command to convert it to text. This is easily accomplished by sending another SYSEXEC command through the system.

```
%SYSEXEC CONVERTDIC &DICFILE.;
```

The DOS based CONVERTDIC program processes the dictionary file and saves the text file under same dictionary name but with a *.map extension. The contents of the example dictionary file are displayed below.

VER	7	1
Q1	8	1
Q2	9	1
Q3	10	1
Q4	11	1
AGE	12	2
SEX	36	1
PARTY	37	1
ZIP	38	5
PRECINCT	43	8

Read Dictionary File

Fortunately, the *.map file is formatted in a predictable manner and can be read with a simple data step.

```
DATA MAPFILE;
  INFILE "F:\991-9901\&DICFILE..MAP" MISSEVER;
  INPUT      @1  NAME      $10.0
             @12 COL      3.0
             @18 WIDTH    2.0;
  DUMMY=1;
RUN;
```

At this point, we have all the necessary parameters for writing a data importing program for this specific project.

A Program That Writes A Program

Write Directly To A File

We used the DATA_NULL_ step to write our program directly into a file and used the MAPFILE dataset to provide the necessary input parameters.

```
DATA _NULL_ ; SET MAPFILE;
  BY DUMMY;
  FILE "F:\991-9901\READ_991-9901.SAS";
```

The dummy variable is equal to 1 for all records in the MAPFILE and is used to mark the beginning and ending records. This is helpful for placing header and footer programming within the file.

```
IF FIRST.DUMMY THEN DO;
PUT "* PROGRAM FOR READING PROJECT 991-9901;";
PUT "FILENAME DATAFILE =
'F:\991-9901\&DATFILE..DAT';";
PUT "LIBNAME DATAFOLDER 'F:\991-9901';";
PUT "DATA DATAFOLDER.RAWDATA;";
PUT "INFILE DATAFILE MISSEVER LRECL=800;";
PUT "INPUT";
END;
```

Within this first DO section, we initiate the program by assigning the data file reference and establishing a permanent data library within the project folder. Then we initiate the import routine by declaring a permanent dataset and providing the necessary INFILE parameters. Finally, the INPUT statement marks the beginning of the column input instructions.

```
IF WIDTH <3 THEN DO;
PUT "@ " COL " " NAME " " WIDTH +(-1) ".0";
END;
ELSE DO;
PUT "@ " COL " " NAME " $CHAR" WIDTH +(-1) ".0";
END;
IF LAST.DUMMY THEN PUT "; RUN;";
```

According to the project template, all survey questions are limited to 99 responses and are thus less than 3 digits long. Project data that are 3 or more digits long are expected to be imported as character data. The run statement marks the end of the data step and the READ_991-9901.SAS program.

Writing To A Dataset

Notice that the technique described above is most suitable for writing a program with a single data step. If called for, the program can be altered to enter the lines of syntax into the rows of a dataset. Then, you could stack datasets together to make a program dataset containing several data steps or procedures. The final step in this approach would be to write out the dataset into a file with code similar to that shown above.

The Finished Program

Now that the READ_991-9901 has been written to import the project data, the novice programmer can read the syntax and how it behaves with the current project.

```
* PROGRAM FOR READING PROJECT 991-9901;
FILENAME DATAFILE = 'F:\991-9901\9919901F.DAT';
LIBNAME DATAFOLDER 'F:\991-9901';
DATA DATAFOLDER.RAWDATA;
INFILE DATAFILE MISSEVER LRECL=800;
INPUT
  @7 VER 1.0
  @8 Q1 1.0
  @9 Q2 1.0
  @10 Q3 1.0
  @11 Q4 1.0
  @12 AGE 2.0
  @36 SEX 1.0
  @37 PARTY 1.0
  @38 ZIP $CHAR5.0
  @43 PRECINCT $CHAR8.0
; RUN;
```

After some experience and training, the novice can edit the default program according to the special needs of an individual project. For example, they may need to change the format and width of a variable. Additionally, even when staffed with experienced programmers, the method of meta-programming allows for advanced procedure automation and saves the time and effort associated with making programs from scratch.

Additional Techniques

In addition to those used in this example, there are numerous techniques that enhance the capabilities of meta-programming. Using PROC CONTENTS is just an example of how one can extract parameter information from SAS object files and libraries. In fact, any PROC that generates a report can be the source of parameter gathering; even those without an OUTPUT= option. For these procedures, consider using the TRACE and OUTPUT features found in ODS. The combination of %SYSEXEC, PROC CONTENTS and ODS provides enormous versatility when designing a meta-programming system.

Conclusion

The flexibility of the meta-programming approach offers many viable solutions to a wide variety of simple and complex processing problems. Almost certainly, the techniques discussed in this paper do not comprise all the possible approaches to this level of programming. Hopefully, more authors will contribute on this topic so that the meta-programming toolbox will grow.

Acknowledgements

I would like to thank my colleagues and the partners of Fairbank, Maslin, Maullin & Associates for supporting this paper and my professional development. I would also like to thank Eliot Roth for his continued mentorship and for introducing me to SAS. The Roth %READDLM() macro was the inspiration for this paper. Foremost, all my love and thanks goes to my wife and son, Estrelita and RJ.

Contact Information

Your comments and questions are truly welcome and encouraged. Please contact the author at:

Renato@FMMA.com or
Fairbank, Maslin, Maullin & Associates
2425 Colorado Avenue, Santa Monica, CA 90404
(310) 828-1183