Meta-Programming in SAS© with Data Step

Paulo Tanimoto
KEMA Inc., Madison, WI

October, 2010

Abstract

It is a common pattern in SAS to pass information from a dataset to macro variables, perform a number of operations, and iterate as necessary. Besides inefficient, this method can be prone to mistakes and vulnerabilities. We present an alternative approach that eliminates the need for macros and macro variables altogether, running directly from a Data Step via `call execute`. Arguments can be validated with regular character functions. We illustrate the technique with examples that require meta-programming: renaming a large number of variables, applying labels and formats from a list, and dropping blank variables. These turn out to be generalizations of basic operations on datasets. A brief discussion of security issues is offered at the end.

1 Introduction

The well-known slogan is, "code is data, data is code".

Along the years, the SAS language has been extended with a number of strategies for processing datasets. A few examples include different forms of iteration, processing observations in groups, manipulating arrays of variables, and, more recently, hash tables. When a certain feature is not available, a programmer might be interested in implementing it herself. Regrettably, there is limited support for modifying, let alone introducing new functionality to the language. To make matters worse, meta-programming is frequently the only resource at hand for combining existing operations.

Programmers usually work by writing programs in a language, which performs some computation when executed. Meta-programming adds another layer of indirection: one writes a meta-program in the meta-language, which when executed can generate a program in the base language. As the *de facto* meta-language in SAS, macros are extremely empowered: they sport a special syntax and enjoy many features that only recently became available in the base language, most notably the ability to define custom functions.

Table 1: Comparison of Approaches in Meta-Programming

Macro Loop 1

```
%macro loop;
/* Number of Obs */
data _null_;
  set foobar nobs=n;
  call symput('n', n);
  stop;
run;

%do i=1 %to &n;
  /* Get ith Variable */
  data _null_; i = &i;
    set foobar point=i;
    call symput('var', var);
    stop;
  run;

  /*
  Code goes here
  Curr Value: &var
  */
%end;
%mend;

%loop;
```

Macro Loop 2

```
%macro loop;
data _null_;
  set foobar nobs=n;
  /* Number of Obs */
  if _n_ = 1 then
  call symput('n', n);

  /* Get all Variables */
  call symput
  (cats('var',_n_), var);
run;

%do i=1 %to &n;
  /*
  Code goes here
  Curr Value: &&var&i
  */
%end;
%mend;

%loop;
```

Data Step

```
data _null_;
  set foobar;
  call execute
  ('/* Code goes here */');
run;
```

Macros are not, however, *the only* way of carrying out meta-programming in SAS. An oft-overlooked idea is to generate code from a Data Step using `call execute`, an operation that is analogous to `eval` in many dynamically-typed languages. This insight is certainly not new and previous discussions can be found in Whitlock (1997); First & Ronk (2006). Our contribution lies in exploring the advantages and generalizations of the technique. Recall that a rectangular dataset, the basic data structure in SAS, is typically processed sequentially, that is, row by row in a simple loop. Because `call execute` takes in a string and submits commands during run-time, we have a mechanism for transforming data into code and thus accomplishing meta-programming. This apparent inversion of control is especially useful when we are required to program according to a given list, as it dispenses with the need for passing information to macro variables and then resolving them.

To appreciate how useful this approach can be, consider the common task of performing certain operations on each observation of a dataset. For generality, we do not determine in advance what the task is, but it could range from creating datasets to running different model specifications. Table 1 compares our technique with two that are usually

found in the literature and even in SAS documentation. The first approach retrieves the i^{th} observation at the start of the loop and saves it into a macro variable. The second approach saves all observations into macro variables beforehand, then proceeds to loop. If we let n be the number of observations in the dataset, in the worst case the first code runs in $O(n^2)$ time and $O(1)$ space. The second is faster, $O(n)$, but consumes much more memory, $O(n)$. In contrast, our approach is concise and combines the advantages of the other two: it runs in $O(n)$ time and takes $O(1)$ space. An extra point in avoiding macro variables is that there is no risk of overwriting existing ones — composability benefits.

At this point a few words about the technique are in order. First, a minor detail is that although code submitted via `call execute` is resolved right away, it is only executed at the end of the Data Step. In other words, the current Data Step must terminate before the submitted code can run. Second, it is important to remember that `call execute` expects a string argument. This means that functions for manipulating text, such as `cat`, `strip`, and `quote` can be helpful if not indispensable. Third, whereas SAS provides system options for troubleshooting macros, the same is not true here. Fortunately, the fact that code is executed from a Data Step means that one can rely on familiar strategies for debugging, such as printing to the log or inspecting a variable. Finally, just as in any form of meta-programming, proper care must be taken to prevent vulnerability problems. We address this point in a separate section.

For the sake of completeness, we show in Table 2 the relationship between meta-programming with macros and with Data Step. Because one technique can be written in terms of the other, it is easy to see that they are equally powerful in most situations. More importantly, the table shows that it is possible to mix different approaches whenever convenient. This we will do in the next section.

Exercise 1. A clear omission from Table 2, due to space constraints, are macro variables. It is indeed possible to write and execute macros and arbitrary Data Steps using only macro variables. Complete the table with those cases.

2 Applications

In lieu of prolonging the discussion on meta-programming, we immediately turn to applications of the approach advanced in the previous section. We present a few cases in which generating code directly from a given dataset can be fruitful. At the end it will hopefully be evident that our programs are nothing more than generalizations of common operations on a dataset.

2.1 Renaming Variables with Regular Expressions

> *Some people, when confronted with a problem, think "I know, I'll use regular expressions". Now they have two problems.*
>
> *Jamie Zawinski*

Table 2: Matrix of Meta-Programming with Macro and Data Step

	Macro	**Data Step**

Macro	`/* Macro in a Macro */`
	`%macro macro1;`
	`  %macro macro2;`
	`  %put Hello World!;`
	`  %mend;`
	`%mend;`
	
	`%macro1;`
	`%macro2;`
		`/* Data Step in a Macro */`
		`%macro macro1;`
		`  data test;`
		`  x = 42;`
		`  run;`
		`%mend;`
		
		`%macro1;`

Data Step	`/* Macro in a Data Step */`
	`data _null_;`
	`  call execute`
	`  ( '%macro macro1;`
	`  %put Hello World!;`
	`  %mend;'  );`
	`run;`
	
	`%macro1;`
		`/* Data Step in a Data Step */`
		`data _null_;`
		`  call execute`
		`  ( 'data test;`
		`  x = 42;`
		`  run;'  );`
		`run;`

Changing the name of a large number of variables is a surprisingly laborious task in SAS. Because renaming is declarative, it is necessary to rely on meta-programming for iteration. Unfortunately, most solutions found in the literature are specialized to a single operation: adding a suffix, prefix, or simple alterations. See Ravi (2003) and Weng & Feng (2009) for recent examples. Our solution uses regular expressions. Regular expressions provide a powerful yet succinct mechanism of performing text substitution. Moreover, it can be shown that many published implementations are a special case of ours. *Indeed, by treating code as data, we reduce the problem to one for which a general solution has already been invented.*[1]

In Listing 1 a simplified version of our macro is shown. Since a list of a dataset's variables can be extracted from `sashelp.vcolumn`, we generate our code directly from it. At the first observation we start a new dataset. For each variable in the list we try to match the regular expression and, for the positive cases, we run the substitution. The next step generates a rename statement, whose syntax is given by:

```
rename <old name> = <new name>;
```

Finally, after the last variable in the dataset is processed, we submit a `run` statement in order to have the generated code executed.

[1] In this case, by the mathematician Stephen Cole Kleene.

Listing 1: Macro for Renaming Variables with Regular Expressions

```
%macro rename_vars_regex(data, out, regex, lib=work);
  data _null_;
    set sashelp.vcolumn end=end;

    /* create a new dataset */
    if _n_ = 1 then do;
      call execute("data &out;");
      call execute("set &data;");
    end;

    /* match and apply substitution */
    match = prxmatch(&regex, strip(name));
    if match > 0 then do;
      subst = prxchange(&regex, -1, strip(name));
      call execute
        (catx(' ', 'rename', name, '=', subst, ';'));
    end;

    /* run commands */
    if end then do;
      call execute('run;');
    end;

    where compare(libname, "&lib" , 'il') = 0
      and compare(memname, "&data", 'il') = 0;
  run;
%mend;
```

In the following code, we present a few examples of usage of our macro, along with resulting changes in comments. The first example illustrates the simplest case of search and replace, in which the substring `"eight"` is changed to `"gt"`. For instance, `"weight"` becomes `"wgt"`. In the next two cases, we deal with the common task of adding a prefix or a suffix to all variables in a dataset. Recall that in a regular expression the special character `'^'` denotes the start of a string, whereas `'$'` refers to the end of it. In practice, we place a given text into the beginning or ending position of the variable name. The fourth example is more involved and shows the expressive power of our approach. The part in parentheses captures the entire variable name and allows us to refer to it with `'\1'`. Adding the prefix and suffix is just a matter of surrounding the backreference with the desired text.

```
/* Sample Dataset */
data class;
```

```
  set sashelp.class;
run;

/* Simple Substitution */              /* Before    After           */
%rename_vars_regex                     /* name   -> name            */
( data  = class                        /* sex    -> sex             */
, out   = class_abbr                   /* age    -> age             */
, regex = 's/eight/gt/i'               /* height -> hgt             */
);                                     /* weight -> wgt             */

/* Adding a Prefix */                  /* Before    After           */
%rename_vars_regex                     /* name   -> pre_name        */
( data  = class                        /* sex    -> pre_sex         */
, out   = class_pre                    /* age    -> pre_age         */
, regex = 's/^/pre_/i'                 /* height -> pre_height      */
);                                     /* weight -> pre_weight      */

/* Adding a Suffix */                  /* Before    After           */
%rename_vars_regex                     /* name   -> name_post       */
( data  = class                        /* sex    -> sex_post        */
, out   = class_post                   /* age    -> age_post        */
, regex = 's/$/_post/i'                /* height -> height_post     */
);                                     /* weight -> weight_post     */

/* Adding a Prefix and Suffix */       /* Before    After           */
%rename_vars_regex                     /* name   -> pre_name_post   */
( data  = class                        /* sex    -> pre_sex_post    */
, out   = class_pre_post               /* age    -> pre_age_post    */
, regex = 's/(\w+)/pre_\1_post/i'      /* height -> pre_height_post */
);                                     /* weight -> pre_weight_post */
```

Note that we implicitly assumed that the regular expression is not only well-formed, but also that it will output a valid variable name. In reality, that assumption is both naive and dangerous. At a minimum, the macro should check if that is indeed the case and sanitize undesirable cases. We come back to this point in a later section.

Exercise 2. If the user's intent is to rename variables in-place, it is possible to optimize our macro so that it avoids creating a new dataset. How would you do that?

Exercise 3. Modify our macro so that instead of renaming variables, it renames datasets using regular expressions. The same idea can be used for libraries and catalogs.

2.2 Applying Labels and Formats from a List

Although there are functions to extract labels and formats from variables, there are no routines to apply them. To do that it is necessary to write declarative statements, which

Listing 2: Macro for Applying Labels and Formats from a List

```
%macro apply_attrib(data, list, out);
  data _null_;
    set &list end=end;

    /* create a new dataset */
    if _n_ = 1 then do;
      call execute("data &out;");
      call execute("set &data;");
    end;

    /* apply labels */
    if not missing(strip(label)) then do;
      call execute
        (catx(' ', 'label', variable, '=', quote(label), ';'));
    end;

    /* apply formats */
    if not missing(strip(format)) then do;
      call execute
        (catx(' ', 'format', variable, format, ';'));
    end;

    /* run commands */
    if end then do;
      call execute('run;');
    end;
  run;
%mend;
```

can only be iterated with some form of meta-programming. This can be particularly frustrating when working with a dataset with many variables. However, in the context of a survey or a database, a list of attributes may be readily available. In that case, we can dispatch the work directly from the dataset, resulting in code that combines simplicity and efficiency. Contrast this approach with the practice of placing the labels and formats in the code, a method that is not only prone to error but also difficult to maintain. *Because we can automate the transformation of data into code, we prefer to keep data in its original form.*

Listing 2 contains a sample implementation. The macro takes an input dataset and a list with labels and formats to create a new dataset with those attributes applied. It assumes that the auxiliary dataset contains variables called `variable`, `label`, and `format`. The idea is to generate code from that dataset which, when executed, will achieve the desired goal. As before, the first and last parts of the code are used to start a new Data Step and run it. The middle part generates the statements for applying the attributes

when the corresponding variable is not missing.

Below we present an example using our macro to attach custom attributes to the dataset `class`. In practice, the macro would be useful when many, possibly hundreds, of labels and formats have to be applied. The only requirement is that the information be stored in variables with the correct names. Running the macro is then a simple matter of passing the appropriate dataset names.

```
/* List of Attributes */
data attrib;
  input variable $15. label $25. format $5.;
/*-----------------------------------------
variable        label                 format
-----------------------------------------*/
  datalines4;
name            First Name              $60.
sex             Sex (M or F)            $1.
age             Age (Years)             8.0
height          Height (Inches)         8.0
weight          Weight (Pounds)         8.0
;;;;
run;

/* Apply Attributes */
%apply_attrib
( data = class
, list = attrib
, out  = class_attrib
);
```

As before, our macro assumes that all data is correct or, at least, safe. For instance, no checks are made to ensure that the variables exist, or that the given labels and formats are valid. The last section provides further discussion of this point.

Exercise 4. One source of inconvenience is that our macro requires the variables in the list to be named exactly `variable`, `label`, and `format`. Adjust the macro so that those names can be passed as optional arguments.

2.3 Dropping Blank Variables

Another application of our technique is the task of dropping blank variables from a dataset. A numeric or character variable is said to be blank if it is missing across all observations. The problem is to efficiently identify variables satisfying that condition and create a new dataset without them. As the reader may have guessed, meta-programming comes into play because keeping and dropping are declarative statements in SAS.

Once again, different solutions can be found in the literature. A recent publication is Sridharma (2010). That paper in turn improves on a solution offered in a SAS Support

page. Unfortunately, both methods suffer from issues identified in Section 1, namely, a dependence on (global) macro variables to hold all the information. Not only that, but because arrays are used for iteration, all variables will be repeatedly checked for missing values even after a counter-example has been found.

Indeed, our solution takes advantage of this observation: it starts with a hash table holding the variables and iterates over them. As soon as a non-missing value is detected, the corresponding variable is discarded from the list, and the process continues with the remaining ones. At the end, the surviving entries are dropped from the dataset. If the hash table becomes empty at any point, the algorithm terminates early.

The complete code is given in Listing 3, in the Appendix. Besides being asymptotically faster, our program is also safer from overflowing and name collisions. Since there are limitations to the sizes of macro variables, previous solutions may not work with extremely large datasets. They also pollute the namespace with global variables, potentially overwriting existing variables. In comparison, these are not issues in our code due to the fact that arguments are submitted directly with `call execute`.

Below we provide an example of usage of our macro with a large dataset: it contains 2,000 numeric and 2,000 character variables, half of which are blank. In our experiments, the other solutions referenced above did not process this dataset correctly.

```
/* Sample Dataset */
data blank;
  length    Numeric____0001-Numeric____2000  3;
  length    Character__0001-Character__2000 $1;
  array num Numeric____0001-Numeric____2000;
  array chr Character__0001-Character__2000;
  do i=1 to 100;
    do j=1 to 1000;
      num[j] = 0;
      chr[j] = "-";
    end;
    output;
  end;
run;

%drop_blank_vars
( data = blank
, out  = blank_clean
);
```

Exercise 5. One useful feature offered by Sridharma's solution is the ability to specify which variables should not be dropped. Implement that for our macro.

Exercise 6. A closely related problem is to remove blank observations, that is, observations for which all variables have missing values. Fortunately, that is much simpler to do. Show how. Why the asymmetry between dropping variables and observations?

3 Security Issues

> *I think computer viruses should count as life. I think it says something about human nature that the only form of life we have created so far is purely destructive. We've created life in our own image.*
>
> *Stephen Hawking*

As hinted in a number of instances, the power of meta-programming comes at the price of constant vigilance for vulnerabilities. This lesson is quite general and is in no manner exclusive to the techniques presented in this paper. When turning data into code, it is the programmer's responsibility to ensure that the generated code does no more than it is supposed to do. In particular, she should be suspicious of data that is obtained from external sources, because even when it is free of malicious intent, an unexpected sequence may lead to an undesired side-effect.

The following discussion provides a hypothetical yet telling example of how seemingly reasonable assumptions can lead to serious vulnerability in a program. In this case, Alice and Bob are working together to import and analyze a large number of `xml` files. Alice receives a listing of all files from Bob, who explains that the names were automatically created by a software. Upon consulting the manual, they find that all names should start with `LX` and be followed by 10 digits. A few entries are shown below.

```
/* List of Files */
data files;
  input file $20.;
  datalines4;
LX2319743788.xml
LX9632097123.xml
LX1438869336.xml
LX1132729300.xml
LX5681552801.xml
LX2887165738.xml
LX1461344487.xml
LX3264443107.xml
LX;ENDSAS;13.xml
LX6944228125.xml
LX1017054301.xml
;;;;
run;
```

Our programmers hurriedly write the following code for importing the files. For convenience, they decide to create the datasets using the same name as the files. Unfortunately, as can be seen in the list, Eve had planted a file with a malicious name, `"LX;ENDSAS;13.xml"`, that went undetected. Note in passing that many special characters in the SAS language are perfectly valid in file names, including `'%'`, `'&'`, and `';'`. Luckily,

the only unexpected effect in this situation is that SAS will terminate in the middle of the session, possibly confusing the programmers. A more advanced exploit could remove files or steal information, and still remain unnoticed. *More than anything, it is important to understand how to ensure that code is kept as code and data as data.*

```
/* Import XML Files */
data _null_;
  set files;
  /* Use the file name as dataset */
  /* Very unsafe! */
  name = scan(file, 1, '.');

  /* Open the library */
  call execute
    (catx(' ', 'libname xml xml', quote(file), ';'));

  /* Import to a dataset */
  call execute
    (catx(' ', 'data', name, '; set xml.data; run;'));

  /* Close the library */
  call execute
    ('libname xml clear;');
run;
```

To solve this specific problem, the code should at least first check if the variable contains a valid dataset name. It is unfortunate that SAS lacks an official function for validating or sanitizing names. We present our own implementation that works for common cases. It verifies that a given string starts with a letter or underscore, and is followed by zero or up to 31 characters that can be a letter, a digit, or an underscore. Note that the function does not take into account special names, such as `_null_` for datasets, or `_n_` for variables.

```
/* Function for Checking Valid Names */
options cmplib = work.functions;
proc fcmp outlib = work.functions.checks;
  function valid_name(name $);
    match = prxmatch('/^[_a-z]\w{0,31}?$/io', strip(name));
    return(match);
  endsub;
quit;

data _null_;
  set files;
  name = scan(file, 1, '.');

  if valid_name(name) then do;
```

```
      call execute
        (catx(' ', 'data', name, '; set xml.data; run;'));
    end; else do;
      put "ERROR: Invalid Name: " name;
    end;
  run;
```

Exercise 7. Each of our previous macros have security holes than can be easily exploited. Fix at least one of them and discuss others.

4 Conclusion

Meta-programming provides a powerful mechanism for accomplishing complex tasks in a language with limited support for abstraction and no higher-order functions. We have discussed a particular form of meta-programming in SAS that has received less attention in the literature, despite its strengths for solving certain problems. We attempted to rectify the situation by offering solutions to problems that appear often in practice, namely, the generalization of operations for renaming, labeling, formatting, and dropping variables. Our discussion ended with a cautionary account of some security risks that come with meta-programming.

5 Acknowledgments

The author is grateful to a number of people who helped reduce the flaws in this paper: Andrew Stryker, Kelley Weston, and Jim Woods. Ken Agnew suggested submitting the abstract in the first place, and the macro for renaming variables with regular expressions was written for Nicole Buccitelli. Special thanks go to Liang Li for unconditional encouragement throughout the writing process.

6 Contact Information

Paulo Tanimoto
paulo.tanimoto@kema.com
KEMA Inc.
122 W Washington Ave Ste 1000
Madison, WI 53703

References

First, Steven, & Ronk, Katie. (2006). Intermediate and Advanced SAS® Macros. *Proceedings of the Thirty-First Annual SAS® Users Group International Conference*. 107-31. Cary, NC: SAS Institute Inc.

Ravi, Prasad. (2003). Renaming All Variables in a SAS® Data Set Using the Information from PROC SQL's Dictionary Tables. *Proceedings of the Twenty Eigth Annual SAS® Users Group International Conference*. 118-28. Cary, NC: SAS Institute Inc.

Sridharma, Selvaratnam. (2010). Dropping Automatically Variables with Only Missing Values. *Proceedings of the SAS® Global Forum 2010 Conference*. 048-2010. Cary, NC: SAS Institute Inc.

Weng, Vincent, & Feng, Ying. (2009). Renaming in Batches. *Proceedings of the SAS® Global Forum 2009 Conference*. 075-2009. Cary, NC: SAS Institute Inc.

Whitlock, H. Ian. (1997). CALL EXECUTE: How and Why. *Proceedings of the Twenty Second Annual SAS® Users Group International Conference*. 70. Cary, NC: SAS Institute Inc.

Appendix

Listing 3: Macro for Dropping Blank Variables

```
%macro drop_blank_vars(data, out);
  data _null_;
    set &data end=_end_;

    /* define arrays with all variables */
    array _num_ _numeric_;
    array _chr_ _character_;

    if _n_ = 1 then do;
      /* define hash table and iterator */
      declare hash  _ht_ (ordered:'y');
      declare hiter _it_ ('_ht_');
      _rc_ = _ht_.defineKey ('_key_');
      _rc_ = _ht_.defineData('_key_');
      _rc_ = _ht_.defineDone();
      length _key_ _lag_ $32;
      call missing(of _key_ _lag_);

      /* load variables into the hash table */
      do _i_=1 to dim(_num_);
        if missing(_num_[_i_]) then do;
          _key_ = vname(_num_[_i_]);
          _rc_ = _ht_.add();
        end;
      end;
      do _i_=1 to dim(_chr_);
        if missing(_chr_[_i_]) then do;
          _key_ = vname(_chr_[_i_]);
          _rc_ = _ht_.add();
        end;
      end;
    end;

    /* iterate over the hash table */
    _rc_ = _it_.first();
    do while (_rc_ = 0);
      if _del_ then _rc_ = _ht_.remove(key:_lag_);
      if not missing(vvaluex(_key_)) then do;
        _lag_ = _key_;
        _del_ = 1;
      end;
      _rc_ = _it_.next();
```

```
    end;
    if _del_ then _rc_ = _ht_.remove(key:_lag_);

    /* execute commands */
    if _end_ or _ht_.num_items=0 then do;
      call execute("data &out; set &data (drop=");
      _rc_ = _it_.first();
      do while (_rc_ = 0);
        call execute(_key_);
        _rc_ = _it_.next();
      end;
      call execute('); run;');
    end;
  run;
%mend;
```