Paper TU01

Tools for supporting the development of robust SAS macros

Martin Gregory, Merck Serono, Darmstadt, Germany

1 Abstract

The PhUSE 2009 paper *Techniques for writing robust SAS macros* [4] presented suggestions for building macros which are robust in the sense that they validate their inputs, handle failure in a controlled and elegant way, appropriately notify the calling programmer or program of the outcome of execution and clean up after themselves.

Application of these techniques does, however, involve additional effort on the part of the developer. This paper describes extensions to the Emacs editor which support the developer in applying the techniques. One extension is the Emacs Speaks Statistics (ESS) package, designed specifically as a development environment for statistical software. Along with a further set of editor functions written by the author to explicitly support application of the techniques, Emacs and ESS may significantly reduce the additional effort required to achieve robustness.

2 Introduction

In any endeavour where standards or standard approaches are desirable, the ease with which these may be applied will, to a large extent, determine the extent to which the standards will be put into practice. In the case of software development in general there are many tools which support programming or development standards. In the case of SAS software, there is not as much choice as for other software. The GNU Emacs editor [1] is a long established and comprehensive tool for editing text of any sort, including program code. One of the most interesting features of Emacs is the ease with which it may be extended. Extensions range from simple customisations such as assignment of commands to keys, through definition of keyboard macros, to the creation of additional commands or applications. One such extension of particular interest to statisticians and statistical programmers alike is the package Emacs Speaks Statistics [6] which provides functionality for working with a number of statistical sofware packages including S-Plus, R, SAS and Stata. In addition to support for the language syntax, ESS is able, with some restrictions in the case of SAS, to control the execution of code in these software packages [5].

3 Emacs

While Emacs is basically a text editor, the large range of available extensions effectively allow it to be used as a so-called integrated development environment. That is to say, in addition to editing programs in Emacs, they may be tested, the outputs examined, files checked into and out of revision control systems, programs, outputs or versions thereof compared. In addition, Emacs has very useful interface to the file system allowing reviewing and managing directories and their contents. For example, if the structure of a dataset is to be changed it is a simple matter to find all the files in a given directory tree which reference this dataset and successively open each file, positioning the cursor at the corresponding location in the file. Searching for and replacing text may make use of regular expressions which allow searching for patterns in context and therefore reduce the number of matches which do not need to be replaced[1].

Emacs provides so-called editing modes which are specific to particular types of text. Of special interest to programmers is the support for many different programming languages. The mode for a language is a collection of functions for manipulating text in the language and includes a syntax table allowing different types of keywords, comments or text strings to be recognized and acted upon. Highlighting keywords is one such action, moving the cursor by units of language constructs is another. In addition to passive support, the modes help a programmer to write readable code by providing features such as automatic indentation or keyword and file name completion. In all these cases, the rules may be customized by the user. Depending on the language, the mode may also provide commands for compiling or executing the code.

A further useful feature of Emacs is the fact that documentation for Emacs functions, both built-in and user-written, is part of the function definition providing instant access to the documentation. We shall see in section 6 below how this is done.

[1] For a comprehensive description of regular expressions see *Programming Perl* [7], for Emacs specifics see *Learning GNU Emacs* [1]

4 Emacs Speaks Statistics

From the ESS manual [6]:

> Statistical packages are powerful software systems for manipulating and analyzing data, but their user interfaces often leave something to be desired: they offer weak editor functionality and they differ among themselves so markedly that you have to re-learn how to do those things for each package. ESS is a package which is designed to make editing and interacting with statistical packages more uniform, user-friendly and give you the power of emacs as well.

ESS provides Emacs modes for a number of statistical software packages. The best support is for R, S-Plus and SAS, while Stata and SPSS are also supported. ESS provides the standard features for programming modes:

- syntax highlighting
- syntactically appropriate indentation
- movement of the cursor according to syntactically meaningful constructs such as function, statement, step
- auto-completion of file names and language keywords
- formatting of comment sections

In addition, ESS can control execution of statistical programs on a local or remote computer. The code to be executed can be typed in directly, selected as part of a program being edited or as an entire program being edited. Execution control is fully supported for R and S-Plus on any platform and for SAS on UNIX systems. Since SAS on Microsoft Windows cannot interact with standard input and output ESS cannot control an interactive SAS session on Microsoft Windows.

ESS is implemented using a standard feature of Emacs for controlling other programs. As a result it inherits a number of useful features such as recalling previous commands, clearing unwanted output and maintaining command history across sessions.

5 Additional extensions for robust macro techniques

In this section we describe a number of extensions which automate the implementation of the techniques described in [4]. With the exception of the function *sas-macro-find-mvar-use*, which checks an existing macro, the functions all generate code from a template, replacing placeholders with values supplied by the user. While the functions themselves are written in a dialect of Lisp known variously as Emacs Lisp or eLisp, it is fairly easy to adapt the actual inserted text for local standards with only a basic understanding of Lisp programming. In order to demonstrate this, we describe in detail, in section 6 below, the code for one of these functions.

Using the functions is simple as Emacs allows binding of functions to key sequences or menu items so that the functions may be called with one or two key strokes. Additionally all of the functions are designed to prompt for required input.

5.1 Setting up a macro skeleton

The *sas-macro-skeleton* function inserts a template for a macro at the cursor. It prompts for the name of the macro and inserts the following template which includes the set-up for a global return code macro variable and a label for premature termination of execution:

```
%macro macname(
   rc=rc_macname) ;
   %if %quote(&rc) eq %str() %then %do ;
      %put NOTE: &sysmacroname: Using %upcase(&sysmacroname) for the RC parameter ;
      %let rc=rc_macname;
      %end ;
   %global &rc ;
   %let &rc=9999 ;

   %* Set RC=0: if we arrive here all should be ok;
   %let &rc=0;
 %DONE:
%mend;
```

5.2 SAS macro parameter validation

The *sas-macro-validate-parms* function builds a *%do %while* group to check that a supplied list of macro variables are non-missing. The most common use is to check that required parameters have been supplied a value. The function takes two parameters, both optional:

Parameter List a list of the parameters to test, separated by periods; and

Return Code the return code to set if any parameter is missing.

If no parameter list is specified, the function attempts to build it automatically by parsing the parameters in the *%macro* statement of the macro in which the cursor is currently located. If the cursor is not located within a macro definition, the function returns without carrying out any actions.

If no return code is specified, a macro comment, stating that no return code is being set, is inserted in place of an assignment.

Note that the return code can be a SAS macro expression containing the loop variable (_pi) used in the *%while* condition to identify the (first) missing parameter.

As an example, consider the following macro definition:

```
%macro finddups(data=_last_,/* input dataset */
                out=_dups,      /* output dataset */
                out2=,          /* optional nodup output dataset */
                by=,            /* by variables */
                where=,         /* where clause */
                maxdups=,       /* global macro var for max number of dups */
                rc=rc_dups      /* global macro var for return code */
                );
```

Calling *sas-macro-validate-parms* with default arguments causes the following code to be inserted:

```
%* parameter validation ;
%local _pi _params _param ;
%let _params=.data.out.out2.by.where.maxdups.rc;
%let _pi=1;
%do %while(%scan(&_params,&_pi,.)^=%str()) ;
   %let _param=%scan(&_params,&_pi,.);
   %if %quote(&&&_param) eq %then %do ;
      %put ERROR: &sysmacroname: Parameter %upcase(&_param) is required.;
      %* Note: no return code is being assigned ;
      %goto DONE;
      %end;
   %let _pi=%eval(&_pi+1) ;
   %end;
```

This template aids the developer in writing robust code by taking care of details such as declaring the temporary macro variables as local, automatically generating the list of parameters and constructing the while loop, allowing the programmer to concentrate on the problem at hand.

5.3 Finding undeclared SAS macro variables

The function *sas-macro-find-mvar-use* searches a macro for variables either assigned or used in the macro but not declared. Such variables can lead to problems if the macro is called by another macro which uses the same variable names. The function does an interactive search through the macro, allowing the user to specify whether an undeclared macro variable name should be saved for later insertion into a *%global* or *%local* statement, or whether it should be ignored. Macro parameters and system macro variables are considered as declared.

The scope of the search is determined by the location of the cursor:

- the cursor must be located within a macro definition;
- /* */ comments within the macro definition are excluded from the search;
- macro definitions embedded within the macro containing the cursor are excluded from the search on the basis that they most likely require their own, independent scoping;

- the entire macro definition is searched, i.e. the search does not start at the cursor position.

Having collected the lists the functions *sas-macro-declare-global*, *sas-macro-declare-local* or *sas-macro-declare-skip* may be used to insert the corresponding list at the cursor. The lists are saved in buffer local variables with the same names as the functions for inserting them. Since these variables are local to a buffer, seaches may be made independently in different buffers. Note, however, that the variables for a buffer are initialised each time the function is called.

Inserting the skip list into a macro comment with the word *skip* immediately following the comment:

```
%*skip ... ;
```

will cause the listed variables to be ignored in future calls. This feature should only be rarely used, as failure to declare macro variables can lead to problems. One case where it is recommended is when using macro constructs such as *&&var&num* to simulate arrays. It is quite likely that the variables may be declared and referenced in ways such that the reference cannot be matched to the declaration. For example, defining a series of local variables could be done explicitly:

```
%local n1 n2 n3 n4 ;
```

or inside a loop of variable length:

```
%let i=1 ;
%do %while (%scan(&list,&i,%str( )) eq %str()) ;
   %local n&i ;
   %let i=%eval(&i+1) ;
   %end ;
```

The variables could be referred to using a different macro variable as index:

```
%do k=1 %to &x ;
   %put &&n&k ;
   %end ;
```

and *n&k* would be identified as an undeclared variable. So it would be useful in this case to add *n&k* to the skip list to avoid having to consider it again on future calls.

The user interface behaves in a similar way to that of the built-in Emacs function *query-replace-regexp*. The function places the cursor at the next undeclared variable and waits for user input which can be one of

g to add the variable to the globals list,

l to add it to the locals list,

s or n to skip to next match, saving this macro variable for a *%*skip* statement,

RETURN or q to exit

h or ? for help

In addition to displaying the list of actions, the help option also displays all variables which were declared when the function was called, and the lists of variables collected so far in the current call.

5.4 Highlighting the current macro

The *sas-macro-highlight-this-macro* function highlights the code of the macro containing point. It excludes /* */ comments and embedded macros from the highlighting. The text highlighted is exactly the text that is searched by the *sas-macro-find-mvar-use*.

5.5 Checking return codes

The function *sas-macro-syserr-block* inserts a conditional *%do* group testing the *syserr* automatic macro variable. Within the group, a message is written to the log, a return code is set and execution is transferred to the macro label *DONE* as set up by the *sas-macro-skeleton*. The function takes two parameters, prompting as necessary:

Message is the text of the message to write to the log. The prefix *ERROR: &sysmacroname:* is automatically inserted.

Return Code is the corresponding return code to assign to the global macro variable defined by the macro parameter RC.

As an example, when called with the values *Failure calculating spline expansion.* for the message and *2002* for the return code, the following code is inserted:

```
%if &syserr > 4 %then %do ;
 %put ERROR: REGSPLINE: Failure calculating spline expansion. ;
 %let &rc=2002 ;
 %goto DONE ;
 %end ;
```

This template is usually inserted after a procedure or data step to check the outcome of the step and to transfer execution to an error handling label.

6 Description of the *sas-macro-syserr-block* function

In order to demonstrate the relatively simple nature of the extensions, we now describe the Lisp code for the function *sas-macro-syserr-block*. Lisp syntax is basically simple, consisting only of function calls. These are written as lists bound by parentheses with the first element of the list being the function and the remaining elements the arguments to the function. Any element of a list may itself be a list. The function *defun* is used to define a function and has the following form:

```
(defun name (argument-list) "Documentation" function-code)
```

The definition of *sas-macro-syserr-block* is:

```
01 (defun sas-macro-syserr-block (the-message the-rc)
02   "Insert a condition testing the SYSERR automatic macro variable, setting
03 a return code and exiting the macro. The function takes two parameters:
04
05  Message: is the text of the message to write to the log. The prefix
06  ERROR: &sysmacroname: is automatically inserted.
07
08  Return Code: is the corresponding return code to assign to the
09  global macro variable defined by the macro parameter RC.
10
11 If major mode is ess-mode or sas-mode, lines are automatically indented.
12
13 sas-macro-syserr-block is bound to \\[sas-macro-syserr-block]
14 "
15   (interactive "sMessage: \nsReturn Code: ")
16   (save-excursion
17     (setq my-start (point-marker))
18     (insert (concat " %if &syserr > 4 %then %do ;\n"
19      "   %put ERROR: &sysmacroname: " the-message ";\n"
20      "   %let &rc=" the-rc ";\n"
21      "   %goto DONE ;\n"
22      "   %end ;\n"))
23     (setq my-end (point-marker))
24     (sas-macro-indent my-start my-end)
25     (set-marker my-start nil)
26     (set-marker my-end nil)
27     )
28   )
```

The line numbers are for reference purposes only and are not part of the lisp program. Line 01 defines the function name and parameters. Lines 02-14 contain the documentation for the function which can be display interactively in Emacs by calling *describe-function* with the name of the function as argument. The syntax

```
\\[function]
```

in line 13 has the effect of displaying the key sequence to which the function is bound instead of the function name, if the function is bound to a key sequence. The *interactive* function call, line 15, specifies how Emacs is to prompt for arguments if they are not supplied. The *save-excursion* function tells Emacs to save various environment settings, most importantly the location of the cursor, and restore them after its call. The *insert* function call on lines 18-22 is the code that actually inserts the text, substituting the arguments as appropriate. Lines 22-26 ensure that the inserted code is properly indented with respect to the context. Lines 27 and 28 terminate the *save-excursion* and *defun* calls, respectively.

The function could be easily extended to parametrize the name of the label used. Apart from modifying the the documentation, only three small changes are required. First, an additional parameter must be added:

```
01 (defun sas-macro-syserr-block (the-message the-rc the-label)
```

The *interactive* function must be modified to prompt for the new parameter:

```
15   (interactive "sMessage: \nsReturn Code: \nsLabel: ")
```

Finally, the value of parameter must be used in the *insert* function:

```
21      "   %goto " the-label " ;\n"
```

With the exception of the *sas-macro-find-mvar-use* and a function called by *sas-macro-validate-parms* to parse the *%macro* statement, all the functions described in this paper are similar in structure and complexity to *sas-macro-syserr-block*.

For the reader who wishes to write simple Lisp programs, *An Introduction to Programming in Emacs Lisp* [2], which is available on-line, is an excellent starting point. For those who wish to write more complex Lisp programs Glickstein's *Writing GNU Emacs Extensions* [3] is an invaluable resource combining a theoretical introduction to the language with substantive case studies.

7 Conclusion

While our description of both Emacs and ESS has been of necessity brief, we have shown in some detail how some simple extensions to Emacs can automate and thereby reduce the effort required for the implementation of the coding techniques for writing robust SAS macros. Furthermore we have shown that the implementation of all but one of these extensions involves relatively simple Lisp programs. This, in turn, means that it is easy to add further extensions as required for other techniques or coding styles.

The facilities available in Emacs, combined with the ESS package and these user-written extensions provide an environment which can significantly improve the productivity of the programmer while at the same time supporting him or her in producing readable and robust code.

References

[1] D. Cameron, B. Rosenblatt, and E. Raymond. *Learning GNU Emacs*. O'Reilly & Associates, second edition, 1996.

[2] Robert J. Chassell. *An Introduction to Programming in Emacs Lisp*. GNU Press, http://www.gnu.org/software/emacs/emacs-lisp-intro/emacs-lisp-intro.html, third edition, 2009.

[3] Bob Glickstein. *Writing GNU Emacs Extensions*. O'Reilly & Associates, 1997.

[4] Martin Gregory. Techniques for writing robust sas macros. *Pharmaceutical Programming*, 2(1):5–13, 2009.

[5] A. J. Rossini, M. Maechler, K. Hornik, R. M. Heiberger, and R. A. Sparapani. Emacs speaks statistics: A universal interface for statistical analysis. Biostatistics Working Paper Series 173, University of Washington, 2001.

[6] A. J. Rossini, M. Maechler, K. Hornik, R. M. Heiberger, and R. A. Sparapani. Emacs speaks statistics. http://ess.r-project.org/, 2008.

[7] L. Wall, T. Christiansen, and R. L. Schwartz. *Programming Perl*. O'Reilly & Associates, second edition, 1996.

Acknowledgements

Thanks to Antonella Santoli and Manuel Cornes for providing feedback on and testing the Emacs Lisp functions presented in this paper.

Contact Information

Martin Gregory
Head of Statistical Programming Germany
Merck KGaA
Frankfurterstr 250
64293 Darmstadt
Germany
Martin.Gregory@merck.de