Insert Diagnostic Code Using Pipes and PRX (Regular Expression) Functions
Brian Fairfield-Carter, ICON Clinical Research, Redwood City, CA

ABSTRACT
Have you ever trudged through a program inserting PROC PRINTs after each data step, trying to figure out where a particular data point went astray? Wished you had a quick and easy way of displaying records for a focal subject at each step boundary? This paper uses a simple ‘pre-processor’ concept to show how diagnostic code can be inserted at run time, by piping SAS statements to a second SAS session and using Perl Regular Expression (PRX) functions to identify target break-points such as step boundaries.

INTRODUCTION
Clinical studies, particularly prior to database lock when the bulk of analysis program development takes place, tend to be plagued with data quality issues. Data handling rules often need to be developed and revised as the study progresses, and require frequent examination and re-examination of the raw data. This often leads to cluttered analysis programs, with commented-out ‘Proc Print’ and ‘Proc Freq’ steps scattered throughout the code. It would be nice to be able to view these snapshots of the data at the various stages of data transformation without having to repeatedly add and remove, or un-comment and re-comment, the diagnostic code.

One approach is to make use of ‘dynamically-commented’ code:

```
%let usubjid=196;
%let cmt=.;

&cmt proc print data=labs;
&cmt WHERE ALSO USUBJID="&USUBJID";
&cmt run;
```

The question is, what if you also wanted to be able to vary the procedure being used, or throw in additional statements (for example, to subset on additional criteria besides subject identifier)?

THE PRE-PROCESSOR CONCEPT
A ‘pre-processor’ is a program that produces output that serves as input to another program. In some cases pre-processors modify and augment source code prior to compilation and execution, using syntax and rules that are themselves ‘program-like’. The SAS macro facility is effectively a pre-processor, since it takes the more compact macro statements as input, and produces the more verbose data step code as output, and this data step code is then used as input to the SAS session.

The approach taken here uses ‘pipes’ (a rudimentary mechanism for communicating between processes, in this instance SAS sessions) to direct the flow of SAS statements, and uses regular expression (PRX) functions to identify ‘insertion points’ for diagnostic code. This approach holds a certain appeal since the inserted program statements do not persist, they’re simply added to the program at compilation; no modified copy of the program is actually created.

REGULAR EXPRESSIONS AND PRX FUNCTIONS
A ‘regular expression’ can be thought of as an ‘abstract string’, or a generalized description of a particular type of text string. A regular expression describes a pattern of characters that can be matched to a certain set of text strings. Regular expressions are constructed using elements describing categories of characters, for example:

```
\d digits [0 - 9]
\D non-digits
\s white-space characters
\S non-white-space characters
```

In Perl, the regular expression

```
\s(s+|$)
```
describes an abstract string containing the text `.sas`, either terminating the line of text (`$`), or followed by at least one blank space (" +"); so this expression would match the string `myprogram.sas`, but would not match `mydata.sas7bdat`.

With SAS 9, Perl regular expressions are supported via the `PRX` functions. For the sake of this paper, all we need to know is that the PRXPARSE function tells SAS to treat the argument passed to the function as a Perl regular expression, and the PRXSUBSTR function matches the regular expression against whatever string you pass to the function (and gives the position within the string where the matching text pattern exists, if any match is found).

THE PRE-PROCESSOR MACRO

The macro described here can be thought of as a pre-processor since it takes a SAS program as input, and passes modified and augmented source code as input to a second SAS session.

The first step is to set up a pipe to a new SAS session, which will receive the SAS statements for the ‘augmented’ program:

```sas
%macro x_preproc(progfile=,regex=,insert=);
    filename test_pipe 'sas9 -ysin /dev/stdin';

The `source` program is then read line-by-line using functions FOPEN and FREAD:

```sas
%let filrf=myfile;
%let rc=%sysfunc(filename(filrf,"&progfile"));
%let fid=%sysfunc(fopen(&filrf));

%if &fid > 0 %then %do;
    data _null_;
        attrib str_ length=200.;
        file test_;
        step_=0;
        %do %while(%sysfunc(fread(&fid)) = 0);
            %let rc=%sysfunc(fget(&fid,c,200));
        %end;

    Lines are masked using the macro quoting function `superq`, and then passed through the pipe to be executed in the second SAS session:

```sas
%let c_=%superq(c);
    put "&c_";

    Insertion points are identified using PRXPARSE and PRXSUBSTR, and program statements are added:

```sas
    pattern = "&regex";
    patternID = prxparse(pattern);
    call prxsubstr(patternID,"&c_",position,length);
    if position>0 then do;
        step_+1;
        str_="**--**-INSERTED CODE, STEP " ||trim(left(put(step_,best.))) ||" **-- **";
        put str_;
        put "&insert";
        str_="/--END INSERTED CODE, STEP " ||trim(left(put(step_,best.))) ||"--++";
        put str_;
        end;
%
end;
run;
@end;
```
Lastly, the input program is closed:

```sas
%let rc=%sysfunc(fclose(&fid));
%let rc=%sysfunc(filename(filrf));
%mend x_preproc;
```

The ‘PRXSUBSTR’ finds the starting position of the specified text pattern within the given line of SAS code, and code is inserted depending on whether or not the target string exists (if it exists, ‘position’ will be >0). This macro isn’t very sophisticated, since code insertion only takes place between lines of input code (in other words, no use is actually made of the ‘position’ and ‘length’ parameters of PRXSUBSTR); however, a fancier example will be presented later on that makes use of these parameters.

When the augmented code is piped to the second SAS session and executed, this SAS session by default writes to ‘standard output’ (the command or console window), so log and listing output are directed to files ‘stdin.log’ and ‘stdin.lst’.

**A SIMPLE EXAMPLE**

Let’s say you have a data derivation program (called ‘xtemp.sas’) that looks like this:

```sas
proc sort data=sasdata.patinfo out=patinfo;
   by subjid;
run;
proc sort data=sasdata.medhx out=medhx;
   by subjid;
run;

data patinfo;
   merge patinfo(in=_1) medhx(in=_2);
   by subjid;
   if _1 & _2;
run;
```

At some point, after completing the program, you decide you want to be able to view data for a specific patient after each data step. Simply use a regular expression that identifies the ‘run’ statements in the program, and specify a ‘proc print’ as the inserted code:

```sas
%x_preproc(progfile=%str(xtemp.sas),
   regex=%str(/run;/i),
   insert=%str(proc print;where input(subjid,best.)=1;run;));
```

The regular expression ‘/run;/i’ specifies a case-insensitive search for the string ‘run;’. The default log file ‘stdin.log’ shows where the ‘proc print’ step is inserted:

```
125 proc sort data=sasdata.patinfo out=patinfo;
126   by subjid;
127 run;

NOTE: There were 262 observations read from the data set SASDATA.PATINFO.
NOTE: The data set WORK.PATINFO has 262 observations and 51 variables.
NOTE: PROCEDURE SORT used (Total process time):
   real time   0.28 seconds
   cpu time    0.06 seconds
128 *** -----INSERTED CODE, STEP 1----- ***;
129 proc print;where input(subjid,best.)=1;run;

NOTE: There were 1 observations read from the data set WORK.PATINFO.
   WHERE INPUT(subjid, BEST12.)=1;
NOTE: The PROCEDURE PRINT printed page 1.
NOTE: PROCEDURE PRINT used (Total process time):
   real time   0.57 seconds
   cpu time    0.06 seconds
130 *** -----END INSERTED CODE, STEP 1----- ***;
...(etc.)...
```
The default output file `stdin.lst` contains the output from the three inserted `proc print` steps. Note that a step counter gives the number of the preceding data step; the same approach can be used to add a title statement for each inserted `proc print`, so that the output can also be matched to the original data step.

**A PERMUTATION**

Inserting code between source code lines may be all you really need, but in some cases what you may actually want is to add code in the middle of an input line (for example, to add a ‘where’ clause to subset every input dataset on a particular set of values). For example, what if at each datastep you wanted to be able to do this:

```sas
proc sort data=sasdata.patinfo(where=(input(subjid,best12.)=1)) out=patinfo;
   by subjid;
run;
```

In this case you’ll want to make use of the ‘position’ and ‘length’ values returned by `PRXSUBSTR`. To illustrate, lets drop an extra chunk of code into the macro, conditionally executed according to a new parameter ‘breakline’:

First, we add a macro parameter (arbitrarily called ‘breakline’) to differentiate between within- and between-line code insertions:

```sas
%macro x_preproc(progfile=,regex=,insert=,breakline=NO);
```

Code to create the pipe and read the input program is identical to the previous example:

```sas
filename test_pipe 'sas9 -sysin /dev/stdin';
%let filrf=myfile;
%let rc=%sysfunc(filename(filrf,"&progfile"));
%let fid=%sysfunc(fopen(&filrf));

%if &fid > 0 %then %do;
   data _null_;
   attrib str_length=$200.;
   file test_;
   step_="_";
   %do %while(%qsysfunc(fread(&fid)) = 0);
   %if &BREAKLINE=NO %THEN %DO;
   <-- (SAME AS THE PREVIOUS EXAMPLE) -->
   %END;

   On this branch, `PRXSUBSTR` captures the position and length of any matching character string...

   %ELSE %DO;
   %let rc=%qsysfunc(fget(&fid,c,200));
   %let c=%superq(c);
   pattern = "&regex";
   patternID = prxparse(pattern);
   call prxsubstr(patternID,"&c",position,length);
   if position>0 then do;
   ...
```

...and the line of code is split apart at the end of the target text string (position+length-1), with additional code inserted at the breakpoint:

```sas
str_=substr("&c_",1,position+length-1);
put str_ @;
put "&insert " @;
str_=substr("&c_",position+length,length("&c_")-(position+length)+1);
```
To use this in order to insert a 'where' clause at each data step, we'd use a macro call something like this:

```sas
%let rc=%sysfunc(fclose(&fid));
%let rc=%sysfunc(filename(filrf));
%mend x_preproc;
```

In this case, our regular expression (/data=\S+\S+/i) performs a case-independent search for any string consisting of "data=", followed by any number of non-whitespace characters, followed by at least one white-space character (this is probably not the most ingenious expression, but it serves to illustrate the point). When we run this against our test program 'xtemp.sas', the log (written to 'stdn.log') shows that the 'where' clause has been added to each data step:

```
130     proc sort data=sasdata.patinfo (where=(input(subjid,best.)=1))
131         out=patinfo;
132         by subjid;
133         run;

NOTE: There were 1 observations read from the data set SASDATA.PATINFO.

WHERE INPUT(subjid, BEST12.)=1;

NOTE: The data set WORK.PATINFO has 1 observations and 51 variables.
NOTE: PROCEDURE SORT used (Total process time):
          real time         0.14 seconds
          cpu time         0.05 seconds

133
134     proc sort data=sasdata.medhx (where=(input(subjid,best.)=1)) out=medhx;
135         by subjid;
136         run;

NOTE: There were 20 observations read from the data set SASDATA.MEDHX.

WHERE INPUT(subjid, BEST12.)=1;

NOTE: The data set WORK.MEDHX has 20 observations and 23 variables.
NOTE: PROCEDURE SORT used (Total process time):
          real time         0.12 seconds
          cpu time         0.02 seconds
```
CONCLUSION

Adapting a 'pre-processor' concept can add fun and enjoyment to the otherwise tedious task of creating and manipulating 'diagnostic code'. The simple illustrations given in this paper will hopefully serve as a starting point for further creative uses (try displaying a record count at each data step, and terminating the SAS session if specified criteria are met).

ACKNOWLEDGMENTS

I would like to thank ICON Clinical Research for consistently encouraging and supporting conference participation, and all my friends at ICON for their great ideas, enthusiasm and support, in particular Syamala Schoemperlen and Jackie Lane.

CONTACT INFORMATION

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

    Name: Brian Fairfield-Carter
    Enterprise: ICON Clinical Research
    City, State ZIP: Redwood City, CA
    E-mail: fairfieldcarterbrian@gmail.com

SAS and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. © indicates USA registration.

Other brand and product names are trademarks of their respective companies.