A common scenario for many programmers is to face a database full of data that needs to be validated. The programmer, therefore, must find both a means of validating the data and reporting the findings of that validation. In such a scenario, I feel the most efficient approach to SAS validation lies in the use of individual, self-sufficient checks grouped together to produce a common output. While this does increase system usage, the advantages of this approach easily outweigh any loss of time and system efficiency. Using self-sustained programs for consistency checks (CCs) allows the checks to be easily copied from one protocol to another, requiring only minor modifications to the path. This also allows the checks to be run individually (if desired) as well as tested individually.

However, if consistency checks are not grouped in some manner during output, the Medical Data Administrator (MDA) can encounter significant headaches. If the checks aren't grouped, preferably by patient, the MDA is left bouncing back and forth between individual reports attempting to determine the number of problems flagged for a particular patient. And, while a patient may only be flagged for one error, there is no way to determine this until all of the CC reports have been examined. It is this dilemma that led me to develop the macros I am discussing today.

Collecting the data into one data set

To group the data we begin by creating a separate program that calls each of the individual checks and runs the macro to group them. This is all done within the macro call:

```sas
%error_in(filename, dataset, program description);
```

The first parameter in the macro call refers to the individual check, which is run from a `%include` statement within the macro. The second parameter is the data set within that individual check, which contains the desired output. The final parameter (as is probably self-evident) is the description of the individual check that is used in our report. So a file in which each CC successively calls the input macro allows all of the checks to run within one batch.

The first obstacle in creating this macro is the means by which we group the data. Certainly, we wouldn't want to spend time renaming all of the variables so that the CCs share a common name. The most obvious solution to this problem is Proc Transpose. Yet that leaves us either running Proc Transpose in each data set (an unacceptable approach) or finding some way the variable name can be passed to the Proc in a consistent manner throughout all of the CCs.

The way to circumvent this problem is to run Proc Contents, storing the output in a data set. By running a DATA step we can then transfer the variable names into a single macro variable using the `symget` and `symput` functions (see Fig.1). In this example we use `symget` as a parameter within the `symput` function; the existing contents of `&varlist` and the next variable name from the Proc Contents output are joined and retrieved by `symget`, putting both back into `&varlist`. The `trim(left(name))` argument is used here because Proc Contents automatically outputs a right justified variable name, and this must be reversed for output in our report.

```sas
%macro errorin(chknm,dset,chktype);
%include "&chknm";
%let varlist=;
data datin;
  set &dset; /*input data set parameter from macro call*/
run;

/*"CREATE VARIABLE SUBOCC TO RESOLVE DIFFICULTIES THAT MIGHT ARISE FROM MULTIPLE RECORDS WITH A SINGLE SUBJECT"*/
data nodups;
  set datin;
  retain subocc;
  by subject;
  if first.subject then
    subocc=0;
    subocc=subocc+1;
  run;
proc contents data=nodups out=cont noprint;
run;

data _NULL_; set cont;
  if upcase(name) ^= "SUBJECT";
  if upcase(name) ^= "SUBOCC";
  call symput("varlist", symget("varlist") || " " ||trim(left(name)));
run;
```

Once the variable list has been created, all that remains of grouping the data is to run Proc Transpose to provide the desired horizontal data structure and to convert the wide variety of variable names into one (see Fig. 2). We now come to the means of accumulating the data. While this could be done with a standard DATA step, it would require far more effort than it is worth to
work properly. It could also be done with Proc Datasets, but that too would require significantly more code than the two lines needed by my preference, Proc Append.

```
proc transpose data=nodups out=trans name=varname
  prefix=var;
by subject subocc;
var &varlist;
run;
/* THIS DATASTEP CONVERTS THE TRANSPOSED DATA FROM RIGHT JUSTIFIED INTO LEFT JUSTIFIED AND ASCERTAINS THAT THERE ARE NO MISSING SUBJECTS***/
data one (drop-vert):
  length descript $60;
  set trans;
  Result=left(var1);
  if subject ^=.;
  descript=&chktype;
run;
proc append base-tot data-one force;
run;
```

Fig. 2

This procedure is repeated, this time creating a one-observation data set in which the description is the only variable kept. It is then appended to a second master data set, which will be used to report the number of errors flagged from each check (Fig. 3).

```
data datnone(keep=descript);
  set contall(obs=1);
  length descript $60;
  set trans;
  Result=left(var1);
  if subject ^=.;
  descript=&chktype;
run;
proc append base=tot data=one force;
run;
```

Fig. 3

Creating the report

Having once gotten the data into one large data set, the actual report creation is simple. The data must, of course, be sorted by patient. In my particular case I added the date of the patient's first visit in the trial. The output was then created in a standard data_NULL_ statement (see Fig. 4).

```
data _NULL_
  format curdate WORDDATE18.;
  length tempvar $8;
  curdate=TODAY(); "GET CURRENT DATE"
  set total;
  by subject descript;
  retain tempvar cnt;
  file print notitle header=top_pg linesleft=chklines;
  if subject ^=.;
  if first.descript then
    tempvar=varname;
  if chklines < 5 then put _PAGE_;
  if first.subject then do;
    put @2 id @12 subject @22 stdate @;
    end;
  if first.descript then do;
    if "first.subject then
      put @95 25*'-';
    put @30 descript @95 vamame @11 0 Result
    end;
  if vamarne-ternpvar then
    put @95 25*'-';
  put @95 varname @11 0 Result;
  end;
  if last.subject then
    put @1 132*'-';
  return;
top_pg:
  pgnum+1;
  put @2 curdate @115 'Page: 'pgnum /
    @55 &protocol' Validation Report' ///
    @1 132*'=';
  return;
run;
```

Fig. 4

We now turn to a means of reporting the number of validation failures from each check. First, we need to limit the appearances of subject to 1 for each failure. To do this we use a method similar to that used above to separate multiple observations under the same subject, but now we keep only the first one (see Fig. 5).

```
data _NULL_
  format curdate WORDDATE18.;
  length tempvar $8;
  curdate=TODAY(); "GET CURRENT DATE"
  set total;
  by subject descript;
  retain tempvar cnt;
  file print notitle header=top_pg linesleft=chklines;
  if subject ^=.;
  if first.descript then
    tempvar=varname;
  if chklines < 5 then put _PAGE_;
  if first.subject then do;
    put @2 id @12 subject @22 stdate @;
    end;
  if first.descript then do;
    if *first.subject then
      put @95 25*'-';
    put @30 descript @95 varname @110 Result
    end;
  if varname=tempvar then
    put @95 25*'-';
  put @95 varname @110 Result;
  end;
  if last.subject then
    put @1 132*'-';
  return;
top_pg:
  pgnum+1;
  put @2 curdate @115 'Page: 'pgnum /
    @55 &protocol' Validation Report' ///
    @1 132*'=';
  return;
run;
```

Fig. 5
data tot;
length tempvar $8;
set tot;
by subject descript;
retain tempvar;
if first.descript then
    tempvar=varname;
if tempvar=varname;
run;

proc sort data=tot;
    by descript;
run;

proc means data=tot noprint;
    by descript;
    var subject;
    output out=tot_flag n=numsubj;
run;

data tot_flag;
merge tot_flag empty;
    by descript;
if numsubj=. then
    numsubj=0;
run;

Next, having sorted the data by description, we run Proc Means to determine the number of observations in the output from each check. The data is then merged with the data set empty, created in the input macro, which contains only the descript variable. This allows us to determine which checks had 0 patients flagged. Information of this sort can be particularly useful. If the output from any particular check is perpetually empty, one might question the necessity of running it. A report very similar to the one created above presents the data, but in a much simpler way (see Fig. 6).

data _NULL_
format curdate WORDDATE18.
curdate=TODAY();
set tot_flag;
by descript;
file print notitle header-top_pg linesleft=chklines;
if chklines < 5 then put _PAGE_
    put @2 descript @96 numsubj;
    put @1 132*'='
    return;
top_pg;
pnum+1;
put @2 curdate @115 'Page: 'pgnum /
    @55 &protocol 'Number of Flags' //
    @1 132''=' /
    @96 'Number' /
    @2 'Description of check' @96 'Flagged' /
    @1 132''=';
return;
r
run;

Conclusion

It may be expected that variations of this approach are already in use elsewhere. Indeed, when I created these macros there were already macros that produced similar reports, but grouped all the checks together at the outset rather than creating them individually and grouping them afterwards. I readily acknowledge the system efficiency advantage of such an approach. My approach, however, requires far less code in the long run because the separate checks can be copied as needed from one protocol to another. The only modifications needed would be designating the path to the data (which can be further limited if a third macro establishes the options and libname statement). Also, once the macros are created, the only things necessary to get a consistent report are to run the macro call (seen at the beginning of this paper) for each check and then run the report macro at the end of the batch. And, the ability to accomplish the desired outcome quickly and easily, as this does, is always the goal of programming.

\footnotetext[1]{Much of the code throughout this paper draws on suggestions and contributions by my co-worker Karl Konrad, whose assistance in this project has been invaluable.}