Paper CT13

Being a Perfect Programmer: Code Smarter, Review Sharper, and Embrace the Future

Akhil Vijayan, Catalyst Clinical Research, Manchester, United Kingdom
Limna Salim, Catalyst Clinical Research, Trivandrum, India

ABSTRACT

What does it take to be the "perfect" programmer in today's dynamic world of clinical data programming? It's more than technical proficiency, it's about cultivating smart habits, leveraging automation, and embracing the power of AI. This paper presents practical tips, coding strategies, and real-world examples that help clinical programmers work more efficiently, code more robustly, and review more effectively.

INTRODUCTION

In today's fast-paced and evolving world of clinical data programming, technical skill alone is no longer enough. The modern clinical programmer is expected to be efficient, adaptable, and forward-thinking, balancing quality, speed, and compliance under growing demands. So, what does it take to be the "perfect" programmer in this dynamic environment?

This paper explores the habits, tools, and smart strategies that elevate good programmers into great ones. Drawing from real-world experience, we present practical tips, coding best practices, and examples that can help clinical programmers work smarter, not harder. Whether it's leveraging purpose-built SAS/STAT® Software macros for final review efficiency, adopting reusable coding structures for consistency across studies, or using AI tools to learn, review, and automate, the goal remains the same: to write more robust code, reduce rework, and make review processes seamless.

Each section of this paper focuses on one core area from final deliverable automation and numeric derivation safety to code reusability and AI-enhanced learning equipping programmers with practical knowledge that can be immediately applied in daily clinical programming tasks.

MACROS IN REVIEW PROCESS

Macros are widely used in many areas of programming to simplify processes, improve quality, and support data checking. They are especially valuable during lead reviews, as they help reduce repetitive tasks and improve accuracy. While not every activity can be automated through macros, there are several areas where they can significantly ease the workload.

One key example is the responsibility of a lead to ensure that validation datasets are generated after the corresponding production datasets during SDTM, ADaM, and TFL development. When double programming is scoped for a project, there should always be a validation dataset in the library corresponding to each production dataset. SAS macros can help automate this check, ensuring that both production and validation datasets are consistently created and updated following any programming changes.

% VALIDATION_CHECKS

This paper introduces a validation checks macro designed to support lead reviews by automating critical quality steps. The macro performs a series of checks including verifying that a validation dataset exists for each development dataset, ensuring validation datasets are generated after development, confirming that the number of observations and variables match between development and validation datasets, and checking that domain labels are consistent. It also ensures that both production and validation datasets are generated after the program's last update date. This macro can be applied across all programming scenarios where double programming is in scope, with input parameters defined as prodlib for the production library name, valdlib for the validation library name, and studyloc for the study location, which helps identify the production and validation programming paths. By integrating these checks, leads can improve consistency, reduce manual review effort, and enhance overall quality in SDTM, ADaM, and TFL development.

Macro working: The macro works by first accessing the datasets in the production library. It begins by retrieving production library dataset details from SASHELP.VTABLE, then identifies the corresponding datasets in the validation library using the production dataset names. Based on company standards, the macro renames production datasets into a V_XX format to match the naming convention for validation datasets. For example, if the production dataset is

DM, the corresponding validation dataset in the library will be stored as V_DM. Once the production and validation datasets are identified, the next step is to extract program details from the study folder. The macro is designed to work with the company's standard folder structure, identifying the production and validation folders for both SDTM and ADaM. For each program, the last modified date of the .sas file is obtained and compared against the dataset timestamp to ensure alignment as part of the validation checks. The corresponding SAS code implementing this logic is provided below.

```
%macro validation_checks(prodlib=SDTM,valdlib=valdata,studyloc=);
*Validation datastet name based on Catalyst standards;
%if %upcase(&prodlib)=SDTM %then %do;
      %let mac_valdata=cats("V_",memname);
      %let mac_mem=scan(memname,2,"_");
%end;
%if %upcase(&prodlib)=ADAM %then %do;
      %let mac_valdata=cats("V_A_",memname);
      %let mac_mem=scan(memname,3,"_");
%end;

     proc sql noprint;
       %*Retrieve dataset information from the provided library;
   create table &prodlib._prod as
   select distinct *
   from sashelp.vtable
   where libname=%upcase("&prodlib");

       %*Assumption: For every production dataset, a corresponding validation dataset
is expected;
   select distinct quote(&mac_valdata) into: vald_data separated by " "
   from &prodlib._prod;

       %*Extract main domain information;
   select count(distinct memname) into: &prodlib.cnt
   from &prodlib._prod
       where substr(memname,1,4) ne "SUPP";
quit;

     proc sql noprint;
   %* Modify variable names in the validation library to support macro programming;
   select distinct cats(name,'=V_',name) into: mac_vars separated by ' '
   from sashelp.vcolumn
       where libname="WORK" and memname=upcase("&prodlib._prod") ;

       %*Retrieve dataset information from the provided library;
   create table &prodlib._vald as
   select distinct *,&mac_mem as mem2
   from sashelp.vtable
   where libname=%upcase("&valdlib") and memname in (&vald_data);
    quit;

    %*Get program modified date to ensure datasets are created after the program
update;
    filename dirlist pipe "dir ""&studyloc.\&prodlib.prog\*.sas"" /T:C /T:W /O:D /-C";
    data &prodlib._p_prod;
      keep memname program moddt_pp modtm_pp modate_pp;
      length program $10;
      format moddt_pp date9. modate_pp datetime16. modtm_pp time5.;
      infile dirlist truncover;
      input line $400.;

      %*Parse only lines with file info;
```

```
      if prxmatch('/^\d{2}\/\d{2}\/\d{4}/', line) then do;
        moddt_pp= input(scan(line,1," "),mmddyy10.);
        modtm_pp= input(scan(line,2," "),time5.);
        modate_pp = dhms(moddt_pp,0,0,modtm_pp);
        program =upcase(scan(line,-1," "));
        memname=scan(program,1,".");
        output;
      end;
    run;

    %*Get program modified date to ensure datasets are created after the program
update;
    filename dirlist pipe "dir ""&studyloc.\validation\*.sas"" /T:C /T:W /O:D /-C";
    data &prodlib._p_vald;
      keep memname program moddt_vp modtm_vp modate_vp;
      length program $10;
      format moddt_vp date9. modate_vp datetime16. modtm_vp time5.;
      infile dirlist truncover;
      input line $400.;

      %*Parse only lines with file info;
      if prxmatch('/^\d{2}\/\d{2}\/\d{4}/', line) then do;
        moddt_vp= input(scan(line,1," "),mmddyy10.);
        modtm_vp= input(scan(line,2," "),time5.);
        modate_vp = dhms(moddt_vp,0,0,modtm_vp);
        program =upcase(scan(line,-1," "));
        memname=scan(scan(program,1,"."),2);
        output;
      end;
    run;

proc sort data=&prodlib._p_prod;
  by memname;
    run;

proc sort data=&prodlib._p_vald;
  by memname;
    run;

data &prodlib._comp;
  keep memname  &prodlib._:;
  merge &prodlib._prod(in=a ) &prodlib._vald(in=b rename=(&mac_vars. mem2=memname))
&prodlib._p_prod(in=c) &prodlib._p_vald;
  by memname;
  if a;
      %*1.* Checking if validation exists corresponding to development;
  if cmiss(v_modate) then &prodlib._1=cat("Validation ",strip(memname), " dataset
missing");
      %*2. Checking validation datasets are generated after development;
  if ~cmiss(modate,v_modate) and modate gt v_modate then &prodlib._2=cat("Production
",strip(memname), " was generated after Validation.");
      %*3. Checking No. of obs are equal in development & validation ;
  if ~cmiss(modate,v_modate) and nobs ne v_nobs then &prodlib._3=cat(strip(memname),"
datasets have an unequal number of variables:",nvar," in Production and ",v_nvar," in
Validation.");
      %*4. Checking No. of variables are equal in development & validation ;
  if ~cmiss(modate,v_modate) and nvar ne v_nvar then &prodlib._4=cat(strip(memname),"
datasets have an unequal number of variables:",nvar," in Production and ",v_nvar," in
Validation.");
```

```
      %*5. Checking domain labels in development & validation ;
  if ~cmiss(modate,v_modate) and memlabel ne v_memlabel then
&prodlib._5=cat(strip(memname)," dataset label is different between Production and
Validation. '",strip(memlabel),"' in Production and '",strip(v_memlabel),"' in
Validation.");
      %*6.Check that production datasets were generated after the programs last update
date;
  if modate lt modate_pp then &prodlib._6=cat("Production ",strip(memname), " was
generated before the program update.");
      %*7.Check that validation datasets were generated after the programs last update
date;
  if v_modate lt modate_vp then &prodlib._7=cat("Validation ",strip(memname), " was
generated before the program update.");

  call symput(cats("p&prodlib.",_n_),strip(memname));
run;
  %mend checklist;
%checklist;
%mend validation_checks;
```

Additional checks can also be incorporated into the macro to further strengthen lead review. For example, program completion dates from the tracker can be compared with program update dates or dataset timestamps to ensure consistency with company standards. Beyond SAS macros, AI tools can be leveraged to generate supporting scripts, such as PowerShell programs, that verify ownership details by comparing the OwnerName attribute in the production and validation libraries, ensuring that the same individual has not worked on both, thereby maintaining independence in double programming.

```
"Generate a PowerShell script that compares the OwnerName field from datasets in the
production and validation libraries and highlights cases where the same owner is
assigned to both."
```

```powershell
# Compare OwnerName between production and validation datasets
$prodPath = "D:\Study\Prod"
$valdPath = "D:\Study\Val"

$prodFiles = Get-ChildItem $prodPath -Recurse | Select-Object Name, @{Name="Owner";Expression={(Ge
$valdFiles = Get-ChildItem $valdPath -Recurse | Select-Object Name, @{Name="Owner";Expression={(Ge

$comparison = Compare-Object -ReferenceObject $prodFiles.Owner -DifferenceObject $valdFiles.Owner

$comparison | ForEach-Object {
    if ($_.SideIndicator -eq "==") {
        Write-Output "Same Owner found in both Prod and Val: $($_.InputObject)"
    }
}
```

CODE SMARTER

As we work in an area where even the smallest decimal difference can significantly impact output, it is the programmer's responsibility to ensure the highest level of quality. A critical first step is to verify that SAS logs are clean, especially when rerunning programs on updated extracts, since such cases may introduce discrepancies in the outputs. Ignoring SAS notes in the log can lead to undetected issues.

CASE 1: HANDLING NUMERIC VARIABLES

Derivations using numeric variables are a core part of programming activities, starting from simple day calculations to complex efficacy derivations. In this example, we consider two datasets from different extracts containing four variables: SUBJID, TEST, TESTDT, and COMPDT. Among these, TESTDT and COMPDT are numeric date variables, highlighting the need for careful handling to ensure consistent and correct derivations.

Extract1

SUBJID	TEST	TESTDT	COMPDT
XX-01	Dummy Test1	2025-07-24	2025-08-13
XX-01	Dummy Test2	2025-07-30	2025-08-13
XX-01	Dummy Test3	2025-08-05	2025-08-13
XX-02	Dummy Test1	2025-07-30	2025-08-18
XX-02	Dummy Test2	2025-08-11	2025-08-18
XX-02	Dummy Test3	2025-08-23	2025-08-18
XX-03	Dummy Test1	2025-08-05	2025-08-23
XX-03	Dummy Test2	2025-08-23	2025-08-23
XX-03	Dummy Test3	2025-09-10	2025-08-23
XX-04	Dummy Test1	2025-08-11	2025-08-28
XX-04	Dummy Test2	2025-09-04	2025-08-28
XX-04	Dummy Test3	2025-09-28	2025-08-28

Extract2

SUBJID	TEST	TESTDT	COMPDT
XX-01	Dummy Test1	2025-07-24	2025-08-13
XX-01	Dummy Test2	2025-07-30	2025-08-13
XX-01	Dummy Test3	.	2025-08-13
XX-02	Dummy Test1	2025-07-30	2025-08-18
XX-02	Dummy Test2	2025-08-11	2025-08-18
XX-02	Dummy Test3	.	2025-08-18
XX-03	Dummy Test1	2025-08-05	2025-08-23
XX-03	Dummy Test2	.	2025-08-23
XX-03	Dummy Test3	2025-09-10	2025-08-23
XX-04	Dummy Test1	2025-08-11	2025-08-28
XX-04	Dummy Test2	2025-09-04	2025-08-28
XX-04	Dummy Test3	2025-09-28	2025-08-28

The following SAS code is used as part of a derivation based on these datasets.

```
data case1;
  set input;
  if compdt gt testdt then flag="Y";
    else flag="N";
run;
```

In the first extract, values for the TESTDT variable were populated for all observations, and the programmer missed including any condition while deriving the flag variable. However, in the second extract, some values in TESTDT were missing, which led to issues in the derived flag variable due to the incomplete programming logic. The outputs from the first and second extracts are shown below.

Extract1

SUBJID	TEST	TESTDT	COMPDT	FLAG
XX-01	Dummy Test1	2025-07-24	2025-08-13	Y
XX-01	Dummy Test2	2025-07-30	2025-08-13	Y
XX-01	Dummy Test3	2025-08-05	2025-08-13	Y
XX-02	Dummy Test1	2025-07-30	2025-08-18	Y
XX-02	Dummy Test2	2025-08-11	2025-08-18	Y
XX-02	Dummy Test3	2025-08-23	2025-08-18	N
XX-03	Dummy Test1	2025-08-05	2025-08-23	Y
XX-03	Dummy Test2	2025-08-23	2025-08-23	N
XX-03	Dummy Test3	2025-09-10	2025-08-23	N
XX-04	Dummy Test1	2025-08-11	2025-08-28	Y
XX-04	Dummy Test2	2025-09-04	2025-08-28	N
XX-04	Dummy Test3	2025-09-28	2025-08-28	N

Extract2

SUBJID	TEST	TESTDT	COMPDT	FLAG
XX-01	Dummy Test1	2025-07-24	2025-08-13	Y
XX-01	Dummy Test2	2025-07-30	2025-08-13	Y
XX-01	Dummy Test3	.	2025-08-13	Y
XX-02	Dummy Test1	2025-07-30	2025-08-18	Y
XX-02	Dummy Test2	2025-08-11	2025-08-18	Y
XX-02	Dummy Test3	.	2025-08-18	Y
XX-03	Dummy Test1	2025-08-05	2025-08-23	Y
XX-03	Dummy Test2	.	2025-08-23	Y
XX-03	Dummy Test3	2025-09-10	2025-08-23	N
XX-04	Dummy Test1	2025-08-11	2025-08-28	Y
XX-04	Dummy Test2	2025-09-04	2025-08-28	N
XX-04	Dummy Test3	2025-09-28	2025-08-28	N

The screenshot below shows the results after rerunning with the second extract, where SAS did not display any note or warning for this issue.

```
22   data case1;
23     set input;
24     if compdt gt testdt then flag="Y";
25       else flag="N";
26   run;

NOTE: There were 12 observations read from the data set WORK.INPUT.
NOTE: The data set WORK.CASE1 has 12 observations and 5 variables.
NOTE: DATA statement used (Total process time):
      real time           0.01 seconds
      cpu time            0.01 seconds
```

A simple condition to check whether the input variables are missing could have prevented the problem. The recommended example code for such cases are:

```
if ~cmiss(compdt, testdt) and compdt > testdt then flag = "Y";

if cmiss(compdt, testdt) = 0 and compdt > testdt then flag = "Y";

if nmiss(compdt, testdt) = 0 and compdt > testdt then flag = "Y";
```

CASE 2: IMPORTANCE OF ADDING CHECKS IN USER-DEFINED FORMATS

Programmers often use user-defined formats for the generation of SDTM or ADaM datasets. This approach supports reusability of programs for example, in the case of extension studies. But, there is a potential risk: new values may be introduced in later studies that were not present in earlier studies and, therefore, not included in the original formats. If these new values are not accounted for, records can be missed during programming.

Example: Consider a case where formatted value is used in the derivation of a flag for a study 101 and reused the same program for another study 102 which is a similar to study 101 with minor changes. The result values of the collected containing the values like “Result 1”, “Result 2”, “Result 3” etc. and these results are categories as to “E1” “E2” “E3” etc. to the variable in ADaM and deriving another variable as apart of the analysis used the values to for a flag when

Study 1

SUBJID	RESULT
XX-01	Result 1
XX-01	Result 2
XX-01	Result 3
XX-01	Result 4
XX-01	Result 5
XX-02	Result 1
XX-02	Result 2
XX-02	Result 3
XX-02	Result 4
XX-02	Result 5

Study 2

SUBJID	RESULT
XX-01	Result 1
XX-01	Result 2
XX-01	Result 3
XX-01	Result 4
XX-01	Result 5
XX-01	Result 6
XX-02	Result 1
XX-02	Result 2
XX-02	Result 3
XX-02	Result 4
XX-02	Result 5
XX-02	Result 6

Suppose the value "Result 6" appears in Study 2, but it was not included in the formats because no such value existed in Study 1. In this scenario, there is a high chance of missing these records in the and the recommended approach to avoid such scenario is noted below.

```
proc format;
  value $resfmt
  "Result 1"="E1"
  "Result 2"="E1"
  "Result 3"="E2"
  "Result 4"="E2"
  "Result 5"="E3"
  other= "CHECK_VALUE";
run;


data case2;
  set result;
  result_cat=put(result,resfmt.);
  if result_cat in ("E2" "E3") then anlfl="Y";
  if result_cat="CHECK_VALUE" then put "WAR" "NING: Check the result value for
subject=" subjid " and result=" result;
run;
```

An example of the log generated when the code is run in the second study is provided below.

```
119  data case2;
120    set result;
121    result_cat=put(result,resfmt.);
122    if result_cat in ("E2" "E3") then anlfl="Y";
123    if result_cat="CHECK_VALUE" then put "WAR" "NING: Check the result value for subject=" subjid "
123!  and result=" result;
124  run;

WARNING: Check the result value for subject=XX-01  and result=Result 6
WARNING: Check the result value for subject=XX-02  and result=Result 6
NOTE: There were 12 observations read from the data set WORK.RESULT.
NOTE: The data set WORK.CASE2 has 12 observations and 4 variables.
NOTE: DATA statement used (Total process time):
      real time           0.01 seconds
      cpu time            0.00 seconds
```

CASE 3: CONCATENATION PRACTICES AND AVOIDING TRUNCATION

Concatenating different variables is a common coding practice frequently used by programmers in their day-to-day activities. But, there is a high risk of truncation when the code is reused in another study or rerun on a different data extract if concatenation is done using delimiters. In the screenshot below, the first and second datasets generate the same output, but in the first dataset, concatenation was done using the || delimiter, while in the second dataset, the cat() function was used. The cat() function generates a warning in the log when the length of the resulting variable does not accommodate the data values, helping to prevent unnoticed truncation.

```
125  data not_rec;
126    length conc $10;
127    set sashelp.cars (obs=2);
128    conc=strip(model)||"-"||strip(type);
129  run;

NOTE: There were 2 observations read from the data set SASHELP.CARS.
NOTE: The data set WORK.NOT_REC has 2 observations and 16 variables.
NOTE: DATA statement used (Total process time):
      real time           0.02 seconds
      cpu time            0.01 seconds


130
131  data recmnd;
132    length conc $10;
133    set sashelp.cars (obs=2);
134    conc=catx("-",of model,type);
135  run;

WARNING: In a call to the CATX function, the buffer allocated for the result was not long enough to
         contain the concatenation of all the arguments. The correct result would contain 20
         characters, but the actual result might either be truncated to 10 character(s) or be
         completely blank, depending on the calling environment. The following note indicates the
         left-most argument that caused truncation.
NOTE: Argument 2 to function CATX('-',' RSX Type S '[12 of 40 characters shown],'Sedan    ') at line
      134 column 8 is invalid.
CONC=  MAKE=Acura MODEL=RSX Type S 2dr TYPE=Sedan ORIGIN=Asia DRIVETRAIN=Front MSRP=$23,820
INVOICE=$21,761 ENGINESIZE=2 CYLINDERS=4 HORSEPOWER=200 MPG_CITY=24 MPG_HIGHWAY=31 WEIGHT=2778
WHEELBASE=101 LENGTH=172 _ERROR_=1 _N_=2
NOTE: There were 2 observations read from the data set SASHELP.CARS.
NOTE: The data set WORK.RECMND has 2 observations and 16 variables.
NOTE: DATA statement used (Total process time):
      real time           0.01 seconds
      cpu time            0.01 seconds
```

CASE 4: USING THE SCAN() FUNCTION EFFECTIVELY

Similar to concatenation, programmers often use the scan() function to handle different programming approaches and implement various logics. The scan() function can be applied based on the current data or in anticipation of a concatenated variable, allowing the code to handle values more flexibly. The recommended approach below provides details on using the scan() function effectively, which helps improve programming quality and ensures that the outputs match the expected results.

The dataset below is used as an example to illustrate scenarios containing the variables SUBJID and RESCAT1 to RESCAT5. The code provided in the section below demonstrates two example cases: Example 1 shows a programmer concatenating variables to implement a logic in the program, while Example 2 shows a programmer subsetting the values into different variables.

SUBJID	RESCAT1	RESCAT2	RESCAT3	RESCAT4	RESCAT5
SUB01	Medium	Unknown,	Unknown,	Medium	High
SUB02	Unknown,	High	Unknown,	Unknown,	Unknown,
SUB03	Low	High	High	Medium	Low
SUB04	Unknown,	Medium	High	High	Medium
SUB05	Medium	Low	Medium	Unknown,	Unknown,
SUB06	Medium	Medium	High	Unknown,	Medium
SUB07	Unknown,	High	Unknown,	Unknown,	Medium
SUB08	Medium	Low	High	Unknown,	High

The screenshot below illustrates a case where the log generated a warning with the code shown, as the programmer attempted to concatenate variables while the delimiter already existed in the data. This warning helps the programmer identify the issue and modify the delimiter to avoid potential problems in the program.

```
613  %let rescatdel=,;
614  data example1;
615    drop i;
616    set case4;
617    *Array for checking the existence of a delimiter before concatenation ;
618    array resvar {*} rescat: ;
619    do i=1 to dim(resvar);
620      if find(resvar(i),"&rescatdel") then put "WAR" "NING: Delimiter found in " SUBJID= "
620! Variable=RESCAT" i " Value=" resvar(i);;
621    end;
622    catval=catx("&rescatdel", of rescat:);
623  run;

WARNING: Delimiter found in SUBJID=SUB01  Variable=RESCAT2  Value=Unknown,
WARNING: Delimiter found in SUBJID=SUB01  Variable=RESCAT3  Value=Unknown,
WARNING: Delimiter found in SUBJID=SUB02  Variable=RESCAT1  Value=Unknown,
WARNING: Delimiter found in SUBJID=SUB02  Variable=RESCAT3  Value=Unknown,
WARNING: Delimiter found in SUBJID=SUB02  Variable=RESCAT4  Value=Unknown,
WARNING: Delimiter found in SUBJID=SUB02  Variable=RESCAT5  Value=Unknown,
WARNING: Delimiter found in SUBJID=SUB04  Variable=RESCAT1  Value=Unknown,
WARNING: Delimiter found in SUBJID=SUB05  Variable=RESCAT4  Value=Unknown,
WARNING: Delimiter found in SUBJID=SUB05  Variable=RESCAT5  Value=Unknown,
WARNING: Delimiter found in SUBJID=SUB06  Variable=RESCAT4  Value=Unknown,
WARNING: Delimiter found in SUBJID=SUB07  Variable=RESCAT1  Value=Unknown,
WARNING: Delimiter found in SUBJID=SUB07  Variable=RESCAT3  Value=Unknown,
WARNING: Delimiter found in SUBJID=SUB07  Variable=RESCAT4  Value=Unknown,
WARNING: Delimiter found in SUBJID=SUB08  Variable=RESCAT4  Value=Unknown,
NOTE: There were 8 observations read from the data set WORK.CASE4.
NOTE: The data set WORK.EXAMPLE1 has 8 observations and 7 variables.
NOTE: DATA statement used (Total process time):
      real time           0.03 seconds
      cpu time            0.03 seconds
```

Code:

```
%let rescatdel=,;

data example1;

  drop i;

  set case4;

  *Array for checking the existence of a delimiter before concatenation ;

  array resvar {*} rescat: ;

  do i=1 to dim(resvar);

    if find(resvar(i),"&rescatdel") then put "WAR" "NING: Delimiter found in " SUBJID=
" Variable=RESCAT" i " Value=" resvar[i];;

  end;

  catval=catx("&rescatdel", of rescat:);

run;


*Checking the number of variables for concatenation;

proc sql noprint;

  select cats(length(catval) - length(compress(catval, '&rescatdel'))+1) into:
cntresdel

  from example1;

quit;


data example2;

  drop i;

  set example1;

  *Array for creating sub-variables based on the values in CATVAL ;

  array subvar {*}$ subvar1-subvar&cntresdel.;

  do i=1 to dim(subvar);

    subvar(i)=scan(catval,i,"&rescatdel");

  end;

run;
```

The working of the code is straightforward. rescatdel is a macro parameter used to pass the delimiter value when concatenating variables. In Example 1, the programmer creates a variable CATVAL that contains the concatenated values from the RESCAT variables. The array resvar ensures that the delimiter value does not already exist in the data; if it does, a warning is displayed, similar to the screenshot above.

In the next example, the programmer subsets the values of variables into separate sub-variables. The first step is to check how many times the delimiter appears in the data, and +1 is used to pass this count to the macro parameter cntresdel. This macro parameter ensures that all values are properly subsetted, even if a new value, such as RESULTCAT6, appears in the dataset.

CASE 5: EFFICIENT DATA HANDLING WITH IDVAR IN PROC TRANSPOSE

Transpose is an area that most programmers (around 90%) use daily, especially in SDTM/ADaM programming. The example below illustrates how a simple ID statement can help identify issues, allowing programmers to update their code or approach accordingly and ensure that all values are handled as expected.

The dataset below comes from a different extract. The first and second transpose codes provided were created based on the initial data (Extract 1). The screenshot below shows the differences in the log after running both codes on the new data extract.

Extract 1

SUBJID	VISIT	PARAM1	PARAM2	PARAM3
XX-01	Visit 1	36.292445351	74.5194713	83.105867302
XX-01	Visit 2	27.627717297	18.382375417	72.88826889
XX-01	Visit 3	7.7892529814	73.431820596	70.725353794
XX-01	Visit 4	76.407951245	48.775378731	18.158753458
XX-01	Visit 5	15.276748228	39.360054647	45.952215905
XX-01	Visit 6	85.948743618	39.253602754	18.129381406

Extract 2

SUBJI	VISIT	PARAM1	PARAM2	PARAM3
XX-01	Visit 1	36.292445351	74.5194713	83.105867302
XX-01	Visit 2	27.627717297	18.382375417	72.88826889
XX-01	Visit 3	7.7892529814	73.431820596	70.725353794
XX-01	Visit 3	.	.	76.407951245
XX-01	Visit 4	48.775378731	18.158753458	15.276748228
XX-01	Visit 5	39.360054647	45.952215905	85.948743618
XX-01	Visit 6	39.253602754	18.129381406	16.222651078

```
proc transpose data=case5 out=case5_t1;
  var param1 param2 param3;
  by subjid visit;
run;


data case5a;
  set case5;
  idvar="result";
run;


proc transpose data=case5a out=case5_t2;
  var param1 param2 param3;
  by subjid visit;
  id idvar;
run;
```

```
351  proc transpose data=case5 out=case5_t1;
352    var param1 param2 param3;
353    by subjid visit;
354  run;

NOTE: There were 7 observations read from the data set WORK.CASE5.
NOTE: The data set WORK.CASE5_T1 has 18 observations and 5 variables.
NOTE: PROCEDURE TRANSPOSE used (Total process time):
      real time           0.04 seconds
      cpu time            0.01 seconds


355
356  data case5a;
357    set case5;
358    idvar="result";
359  run;

NOTE: There were 7 observations read from the data set WORK.CASE5.
NOTE: The data set WORK.CASE5A has 7 observations and 6 variables.
NOTE: DATA statement used (Total process time):
      real time           0.03 seconds
      cpu time            0.01 seconds


360
361  proc transpose data=case5a out=case5_t2;
362    var param1 param2 param3;
363    by subjid visit;
364    id idvar;
365  run;

ERROR: The ID value "RESULT" occurs twice in the same BY group.
NOTE: The above message was for the following BY group:
      SUBJID=XX-01 VISIT=Visit 3
WARNING: 1 BY groups omitted due to earlier errors.
NOTE: There were 7 observations read from the data set WORK.CASE5A.
NOTE: The data set WORK.CASE5_T2 has 15 observations and 4 variables.
NOTE: PROCEDURE TRANSPOSE used (Total process time):
```

REDEFINING EFFICIENCY WITH AI TOOLS

The applications of AI tools extend across many aspects of clinical trial programming, from the creation of Excel trackers to more advanced R dashboards. Traditionally, most organizations have relied on Excel-based trackers to plan and monitor study progress, as well as to record clarifications or data issues identified during programming.

HOW TO CREATE A TRACKER

Some organizations may already have trackers developed in the past. But, if the tracker is outdated, it may be time to review and update it with the help of AI tools such as ChatGPT or GitHub Copilot. For example, one can provide a query like:

"Create a template for a tracker that can be used to monitor SDTM or ADaM development progress."

Below is a screenshot of a sample response generated for the above query.

SDTM / ADaM Development Progress Tracker

Domain / Dataset	Specification Status	Programming Status	QC / Validation Status	Issues Found	Issues Resolved	Final Status	Pinnacle 21 Validation (SDTM only)	Target Completion Date	Responsible Programmer	Responsible QC	Comments / Nc
DM (Demographics)	Not Started / In Progress / Completed	Not Started / In Progress / Completed	Not Started / In Progress / Completed	0	0	Pending / Completed	Pass / Fail / NA	2025-09-05	Akhil	Reviewer 1	Example comm
AE (Adverse Events)	In Progress	Not Started	Not Started	1	0	Pending	NA	2025-09-10	Programmer 2	Reviewer 2	Waiting for CRF updates

Once a base template is available, it can be further refined and customized according to specific requirements. For example, an updated command could be:

"Provide only one Status column with the options: Not Started, In Progress, QC to Review, Prod to Review, and Completed. Programming Status is not required. Include the names of the Production and Validation Programmers."

SDTM / ADaM Development Progress Tracker

Domain / Dataset	Specification Status	Production Target Date	Production Completion Date	Production Programmer	QC Target Date	QC Completion Date	QC / Validation Programmer	Status	Issues Found	Issues Resolved	Pinnacle 21 Validation (SDTM only)	Comments / Notes
DM (Demographics)	Completed	2025-09-05	2025-09-04	Programmer 1	2025-09-07	2025-09-06	Reviewer 1	Completed	2	2	Pass	Spec updated to V2
AE (Adverse Events)	In Progress	2025-09-10	-	Programmer 2	2025-09-12	-	Reviewer 2	Not Started	1	0	NA	Waiting on CRF updates
VS (Vital Signs)	Completed	2025-09-01	2025-08-30	Programmer 3	2025-09-03	-	Reviewer 3	QC to Review	1	0	Pass	Validation ongoing

AI tools can also help apply Excel rules to highlight rows or cells with different color codes, or generate graphs based on study progress. These visualizations allow leads and managers to quickly assess updates and track overall progress more efficiently.

SYNTHETIC DATA CREATION: A SMARTER WAY TO TEST CODE

Programmers can easily leverage AI tools to improve programming quality and ensure that code runs dynamically, even when data changes, while also generating custom errors as expected. A common practice is for programmers to manually alter datasets with different values to test code robustness. However, a more efficient and scalable approach is to use AI tools to generate varied data values and pass them into test datasets. This method allows programmers to validate that their programs handle unexpected or edge-case scenarios effectively.

It is important to note that programmers should never provide study-level or confidential information to AI tools. Instead, they should use generic requests. For example, a safe query might be:

"Create a raw demography dataset format in SAS with the following variables: SUBJECT, SEX, RACE, COUNTRY, and AGE.",

Users can further enhance the output by providing additional specifications, such as:

"For the RACE variable, create RACE1 to RACE5, with 1 indicating 'Yes' and 0 indicating 'No' for different races. Include a few intentional data issues to test whether they can be identified during programming."

Response for **1st Query:**

```
sas

data demography;
    length SUBJECT $10 SEX $1 RACE $20 COUNTRY $20 AGE 8;
    infile datalines dsd truncover;
    input SUBJECT $ SEX $ RACE $ COUNTRY $ AGE;
    datalines;
SUBJ001,M,Mixed,USA,45
SUBJ002,F,Asian,India,33
SUBJ003,M,White,UK,52
SUBJ004,F,Black,South Africa,29
SUBJ005,M,Hispanic,Brazil,40
SUBJ006,F,Asian,China,37
SUBJ007,M,White,Australia,61
SUBJ008,F,Black,Nigeria,48
;
run;
```

2nd Query

```
data raw_demography;
    /* Define variables */
    length SUBJECT $10 SEX $1 COUNTRY $20;
    array RACE[5] RACE1-RACE5;

    /* Input sample data with a race data issue (RACE3=2 for subject 003) */
    input SUBJECT $ SEX $ RACE1-RACE5 COUNTRY $ AGE;

    datalines;
001 M 1 0 0 0 0 USA 25
002 F 0 1 0 0 0 CAN 30
003 M 0 0 2 0 0 UK 28  /* Data issue: RACE3=2, invalid value */
004 F 0 0 0 1 0 AUS 35
005 M 0 0 0 0 1 IND 40
006 F 1 0 0 0 0 USA 27
007 M 0 1 0 0 0 CAN 32
008 F 0 0 1 0 0 UK 29
009 M 0 0 0 1 0 AUS 38
010 F 0 0 0 0 1 IND 42
;
run;
```

CONCLUSION

Efficient and accurate programming is critical in clinical trial development, where even minor errors can impact results. This paper highlights how macros and AI tools can support review processes by reducing repetitive tasks, improving consistency, and ensuring quality. The validation checks macro demonstrates the value of automation in aligning production and validation outputs, while coding practices such as handling missing values, designing robust formats, and applying effective concatenation, scan, and transpose techniques reinforce the importance of building flexibility into programs.

Beyond macros, AI tools provide additional opportunities to streamline workflows, from generating trackers and visualizations to creating synthetic datasets for robust testing. While these tools enhance efficiency, programmer oversight remains essential to maintain accuracy and integrity. By integrating sound programming practices with

automation and AI, organizations can achieve a more reliable, efficient, and standardized review process, ultimately strengthening the quality of SDTM, ADaM, and TFL deliverables.

ACKNOWLEDGMENTS

We would like to thank our colleagues, reviewers, and mentors for their valuable guidance and support in preparing this paper. We also acknowledge the contributions of the programming and statistical teams for sharing practical insights that shaped the examples presented. Additionally, we would like to recognize the role of AI tools, including ChatGPT, in assisting with idea structuring and content refinement, which streamlined the writing process.

CONTACT INFORMATION

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

Author Name: Akhil Vijayan
Company: Catalyst Clinical Research
Address: Alderley Park, Block 11, Alderley Edge, SK10 4ZF, UK
Work Phone: +44 1625 810 241
Email: akhil.vijayan@catalystcr.com
Website: https://catalystcr.com/

Co-Author Name: Limna Salim
Company: Catalyst Clinical Research
Address: Second Floor, Nila Building, Technopark Campus, Thiruvananthapuram , Kerala, 695581
Work Phone:
Email: limna.salim@catalystcr.com
Website: https://catalystcr.com/