SESUG Paper 231-2019

Getting Started with PROC DS2

James Blum, University of North Carolina Wilmington;
Jonathan Duggins, North Carolina State University

ABSTRACT

This workshop is designed to give DATA step programmers foundational information to develop programs in PROC DS2. Starting with several common tasks given as DATA step program examples, the workshop goes through transitioning the code examples to PROC DS2 code step-by-step. As part of the process, various similarities and differences between the two steps are noted, and pros and cons of using each are discussed. Suggested topics for study for building on the PROC DS2 concepts presented are also provided, along with reference material to aid in further study.

INTRODUCTION

Introduced with SAS® 9.4, the DS2 procedure expands on the power of the DATA step, adding in SQL support including ANSI SQL data types, user-defined methods and packages, and other language extensions. Of course, these modifications come at the expense of learning new syntax rules. In the examples that follow, common tasks from DATA step programming are transitioned to PROC DS2 to illustrate some similarities and differences. The goal here is to build some foundations and intuition for adapting DATA step programming skills to the DS2 language.

BASIC PROGRAM STRUCTURE IN PROC DS2

The DS2 procedure has a few basic features that differentiate it from other procedures. It is one example of a procedure that ends with a QUIT statement. It also supports RUN-group processing, allowing for the submission of multiple RUN-groups as long as PROC DS2 is running—the PROC DS2 statement need not be resubmitted. In this behavior, it is similar to PROC SQL, which allows SQL statements to be submitted individually while PROC SQL is running. There are three fundamental RUN-groups permitted in PROC DS2: DATA, PACKAGE, and THREAD. Most of this paper focuses on the DATA RUN-group, starting with the next section.

A (VERY) SIMPLE DS2 PROCEDURE PROGRAM

To get started with the DS2 procedure, consider the “Hello World”-style code given below:

```
data _null_;
  Say='I am from the DATA step';
  put say;
run;

proc ds2;
  data _null_;
    method run();
      Say='I am from DS2';
      put say;
    end;
  enddata;
  run;
quit;
```

As expected, the DATA step puts a single line into the SAS log with the value of the variable Say (along with notes on execution). The DS2 program, in addition to execution notes, also writes the value for its version of Say but, perhaps surprisingly, also generates the following warning:

```
WARNING: Line ####: No DECLARE for assigned-to variable say; creating it as a
      global variable of type char(13).
```

Given the ability of PROC DS2 to work with multiple platforms and data types, variable declaration and scope is much more particular than in the DATA step. Before exploring this further, consider the following example:

```
proc ds2;
  data _null_;
    method run();
      Say='I am from DS2';
      put say;
    end;
  enddata;
  data _null_;
    method run();
      Say='Me too!';
      put say;
    end;
  enddata;
  run;
quit;
```

This code generates the following collection of errors:

```
ERROR: Compilation error.
ERROR: Parse encountered DATA when expecting end of input.
ERROR: Line ####: Parse failed:  >>> data <<<  _null_;
```

The RUN statement is not an optional step boundary in this setting and the attempt to invoke a second DATA RUN-group fails (place a RUN statement after the first ENDDATA statement and the code functions). The ENDDATA statements are not required here, but including them is a good programming practice.

COMPUTING A NEW VARIABLE WITH PROC DS2

To begin with DATA step operations we are all familiar with, compute the EPA combined MPG for the Cars data distributed with SAS in the Sashelp library. Typical DATA step code would look something like this:

```
data Combo;
   set sashelp.cars;
   mpg_combo=0.55*mpg_city+0.45*mpg_combo;
run;
```

The RUN method, used in a simple form in the previous examples, performs much like the implicit loop, implicit output structure of the typical DATA step. It also supports common DATA step statements like SET—any execution-time statement from the DATA step that is permitted with PROC DS2 must occur inside a method. A reasonable attempt at completing the task could be formulated as follows:

```
proc ds2;
  data cars;
    method run();
     set sashelp.cars;
     mpg_combo=0.55*mpg_city+0.45*mpg_combo;
    end;
  enddata;
```

```
  run;
quit;
```

Unfortunately, this seemingly simple program produces the following set of errors:

```
ERROR: Compilation error.
ERROR: BASE driver, schema name SASHELP was not found for this connection
ERROR: Table "SASHELP.CARS" does not exist or cannot be accessed
ERROR: Line ####: Unable to prepare SELECT statement for table cars
```

This error set seems to imply that the Sashelp.Cars data set is not present—while it may not be present in certain circumstances, it most certainly was when this code was initially executed. Sashelp is a composite library, made up of several directories, and PROC DS2 does not support connections to these types of libraries. Fortunately, a copy of the cars dataset has been supplied with the data for this workshop; the next attempt at this assumes the library DS2HOW has been assigned to the folder where these data sets are stored:

```
proc ds2;
  data cars;
    method run();
     set DS2HOW.cars;
     mpg_combo=0.55*mpg_city+0.45*mpg_highway;
    end;
  enddata;
  run;
quit;
```

While this still gives a type declaration (and scope) warning message in the log, it does create the data set expected. The next example attempts to use the DECLARE statement to define MPG_Combo as a double-precision variable:

```
proc ds2;
  data cars;
    method run();
     declare double mpg_combo;
     set DS2HOW.cars;
     mpg_combo=0.55*mpg_city+0.45*mpg_highway;
    end;
  enddata;
  run;
quit;
```

This code produces an error set, shown below, but these errors are not related to the attempt to use the DECLARE statement.

```
ERROR: Compilation error.
ERROR: Base table or view already exists CARS
ERROR: Unable to execute CREATE TABLE statement for table work.cars.
```

The default behavior for table creation is to not overwrite tables that already exist. The behavior can be changed to overwrite with the dataset (or table) option OVERWRITE=YES, as shown below:

```
proc ds2;
  data cars(overwrite=yes);
    method run();
     declare double mpg_combo;
     set DS2HOW.cars;
     mpg_combo=0.55*mpg_city+0.45*mpg_highway;
    end;
  enddata;
  run;
```

```
quit;
```

Now, open the Cars data set from the Work library, or run PROC CONTENTS on it—either shows that the MPG_Combo variable is not present. PROC DS2 allows variables to have either global and local scope—local variables being local to the method in which they are defined. Variables derived from tables given in a SET statement (and MERGE and others) are global in scope. Looking back at the warnings generated for undeclared variables, it is clear that these were given a global scope as well. Since the declaration for MPG_Combo appears inside the RUN method, it is local to that method and is temporary, meaning it is not part of the PDV. The next example fixes this declaration, by moving it to the opening statement of the DATA RUN-group, and the process completes with no warnings or errors:

```
proc ds2;
  data cars(overwrite=yes);
    declare double mpg_combo;
    method run();
     set DS2HOW.cars;
     mpg_combo=0.55*mpg_city+0.45*mpg_combo;
    end;
  enddata;
  run;
quit;
```

Since PROC DS2 is designed to work directly with more platforms and data types than the DATA step is, it is much more particular about variable declaration. Therefore, since variable declaration is much more sensitive in PROC DS2, options are available to control how undeclared variable references are handled. In the PROC DS2 statement, the SCOND option can be set to NONE, NOTE, WARNING (the default), or ERROR. For each of the first three, the program compiles and executes normally, with the corresponding information type for messages on undeclared variables sent to the log. ERROR causes compilation to fail and the program does not execute. The SAS system option DS2SCOND has the same possible settings and produces the same behavior. The program that follows revisits and earlier attempt without declarations; however, this time the SCOND setting of ERROR results in PROC DS2 not executing:

```
proc ds2 scond=error;
  data carsB(overwrite=yes);
    method run();
     set DS2HOW.cars;
     mpg_combo=0.55*mpg_city+0.45*mpg_combo;
    end;
  enddata;
  run;
quit;
```

The error messages are perhaps a bit deceiving:

```
ERROR: Compilation error.
ERROR: Line ####: No DECLARE for assigned-to variable mpg_combo; creating it
       as a global variable of type double.
```

The MPG_Combo variable is not really created as no data set is created by this program at all.

OTHER METHODS AVAILABLE IN THE DATA RUN-GROUP

In addition to the RUN method, the DATA RUN-group supports the INIT and TERM methods—the following code provides a simple illustration of each:

```
proc ds2;
  data carsC(overwrite=yes);
    declare double mpg_combo;
    method init();
      put 'New variable initialized, processing of data to commence';
    end;
    method run();
     set sasuser.cars;
     mpg_combo=0.55*mpg_city+0.45*mpg_highway;
     put _n_;
    end;
    method term();
      put 'Data is ready';
    end;
  enddata;
  run;
quit;
```

Here, the INIT method operates in a manner similar to conditioning on _N_ = 1 in a DATA step, and the TERM method is akin to conditioning on an END= variable. Note the position of the DECLARE statement for the MPG_Combo variable, and contrast it with the following:

```
proc ds2;
  data carsD(overwrite=yes);
    method init();
      declare double mpg_combo;
      put 'New variable initialized, processing of data to commence';
    end;
    method run();
     set DS2HOW.cars;
     mpg_combo=0.55*mpg_city+0.45*mpg_highway;
     put _n_;
    end;
    method term();
      put 'Data is ready';
    end;
  enddata;
  run;
quit;
```

In the end, the CarsC and CarsD data sets are the same, but the process is different and, if SCOND is set to ERROR, CarsD will not be created as MPG_Combo is only declared as local to the INIT method.

DATA STEP METHODS FOR COMBINING DATA SETS APPLIED IN PROC DS2

The data provided with this paper and workshop contains some splits of the cars data sets, one set of those splits being across the Origin variable: AsiaCars, EurCars, and USCars. Consider the following PROC DS2 code:

```
proc ds2;
  data carsBuild;
  method run();
    set DS2HOW.AsiaCars DS2HOW.EurCars DS2HOW.USCars;
  end;
  enddata;
  run;
quit;
```

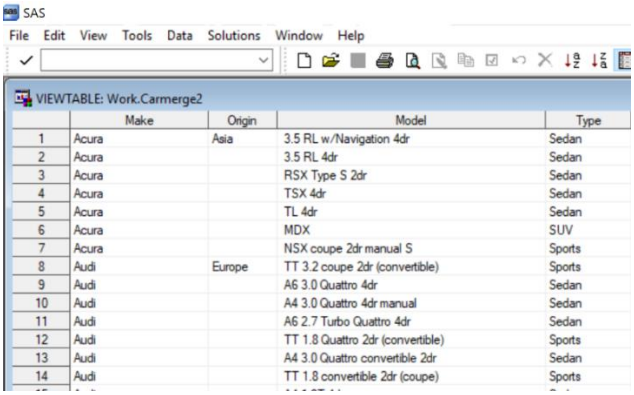
The concatenation of these data sets occurs much like that of the DATA step. The cars data set is also split across its variable set: CarDims (which includes dimensions along with make and model information), CarPrices (price info with make and model also), and CarOrigins (including only make and origin). Assuming the proper sorting on all data sets, the one-to-one merge of the price and dimension information in PROC DS2 looks and performs like that of the DATA step:

```
proc ds2;
  data carMerge;
    method run();
      merge DS2HOW.CarDims DS2HOW.CarPrices;
      by make model drivetrain;
    end;
    enddata;
  run;
quit;
```

Joining the origin data to either the prices or the dimensions is a one-to-many merge, the code you would expect to run is likely something like this:

```
proc ds2;
  data carMerge2(overwrite=yes);
    method run();
      merge DS2HOW.CarOrigins DS2HOW.CarDims;
      by make;
    end;
    enddata;
  run;
quit;
```

Inspecting the data shows that the execution of this merge does not perform the same way in PROC DS2 as it does in the DATA step. The Origin variable, which is present only in the CarOrigins table, does not have its value retained for all matches on the Make variable, as it would in a DATA step merge.



	Make	Origin	Model	Type
1	Acura	Asia	3.5 RL w/Navigation 4dr	Sedan
2	Acura		3.5 RL 4dr	Sedan
3	Acura		RSX Type S 2dr	Sedan
4	Acura		TSX 4dr	Sedan
5	Acura		TL 4dr	Sedan
6	Acura		MDX	SUV
7	Acura		NSX coupe 2dr manual S	Sports
8	Audi	Europe	TT 3.2 coupe 2dr (convertible)	Sports
9	Audi		A6 3.0 Quattro 4dr	Sedan
10	Audi		A4 3.0 Quattro 4dr manual	Sedan
11	Audi		A6 2.7 Turbo Quattro 4dr	Sedan
12	Audi		TT 1.8 Quattro 2dr (convertible)	Sports
13	Audi		A4 3.0 Quattro convertible 2dr	Sedan
14	Audi		TT 1.8 convertible 2dr (coupe)	Sports

There is a direct work-around for this problem—the SET statement supports embedded SQL, as shown in the next example:

```
proc ds2;
  data carMerge3(overwrite=yes);
    method run();
    set {select Origin, Dim.*
          from DS2HOW.CarOrigins as Orig, DS2HOW.CarDims as Dim
          where orig.make = dim.make};
    end;
    enddata;
  run;
quit;
```

The PROC DS2 MERGE statement (and other such statements like MODIFY) does not support imbedded SQL. Also, the SQL queries do not support mnemonics for comparison operators such as EQ (try it in the WHERE clause above).

MORE COMPLEX COMPUTATIONS AND CONDITIONAL LOGIC

A data set named Employees is also provided with the files given for this workshop, and for this data the following modifications are to be made:

1. Create a retirement eligibility flag for any employees at least 65 years of age, or at least 60 years of age with at least 30 years of service.
2. Compute an updated salary base on a 2% raise for level 1 employees, 1.5% for level 2 employees, 1% for level 3, and 1.75% for all others. The level is stored as the third character of the JobCode variable

A first attempt at this based on DATA step principles might look like the following:

```
proc ds2;
  data emps(overwrite=yes);
    method run();
      set DS2HOW.employees;
      Age=yrdif(DateOfBirth,Today());
      Service=yrdif(DateOfHire,Today());
      Level=input(substr(JobCode,3,1),1.);
      if age ge 65 or (age ge 60 and Service ge 30) then RetEligible='Y';
        else RetEligible='N';
      select(level);
        when(1) salary=1.02*salary;
        when(2) salary=1.015*salary;
        when(3) salary=1.01*salary;
        otherwise salary=1.0175*salary;
      end;
    end;
  enddata;
 run;
quit;
```

Right away, of course, a set of errors are generated:

```
ERROR: Compilation error.
ERROR: Parse encountered INPUT when expecting one of: identifier constant
       expression.
ERROR: Line ####: Parse failed: Level= >>> input <<< (substr(JobCode,3,1),
```

The INPUT function is not available in PROC DS2—not all DATA step functions have direct analogs to the DS2 procedure. The INPUTN function can be used here, as is attempted in the following:

```
proc ds2;
  data emps(overwrite=yes);
    method run();
      set DS2HOW.employees;
      Age=yrdif(DateOfBirth,Today());
      Service=yrdif(DateOfHire,Today());
      Level=inputn(substr(JobCode,3,1),1.);
      if age ge 65 or (age ge 60 and Service ge 30) then RetEligible='Y';
        else RetEligible='N';
      select(level);
        when(1) salary=1.02*salary;
```

```
        when(2) salary=1.015*salary;
        when(3) salary=1.01*salary;
        otherwise salary=1.0175*salary;
      end;
    end;
  enddata;
 run;
quit;
```

Unfortunately, while the replacement of INPUT with INPUTN repairs one problem, another is exposed:

```
ERROR: Compilation error.
ERROR: Line ####: Invalid conversion for date or time type.
ERROR: Line ####: Invalid conversion for date or time type.
```

As PROC DS2 is able to work with many data types, implicit type conversion occurs often (and there are options to change any messages generated by such conversions). Date and time-related data types are not converted even when they might otherwise need to be (they are considered non-coercible). DateOfBirth and DateOfHire are considered to be of the DATE type, but the YRDIF function expects doubles as input. The subsequent code uses the TO_DOUBLE function to correct this:

```
proc ds2;
  data emps(overwrite=yes);
    method run();
      set DS2HOW.employees;
      Age=yrdif(to_double(DateOfBirth),Today());
      Service=yrdif(to_double(DateOfHire),Today());
      Level=inputn(substr(JobCode,3,1),1.);
      if age ge 65 or (age ge 60 and Service ge 30) then RetEligible='Y';
        else RetEligible='N';
      select(level);
        when(1) salary=1.02*salary;
        when(2) salary=1.015*salary;
        when(3) salary=1.01*salary;
        otherwise salary=1.0175*salary;
      end;
    end;
  enddata;
 run;
quit;
```

This program achieves the desired result, though with some ugliness—several type declaration warnings and several error messages generated by the INPUTN function (for those cases where the value cannot be converted to a number). The following code applies previously learned principles to clean up these issues:

```
proc ds2;
  data emps(overwrite=yes);
    declare char(1) RetEligible; /**1**/
    method run();
      declare double age; /**2**/
      declare double service;
      declare integer level;
      set DS2HOW.employees;
      Age=yrdif(to_double(DateOfBirth),Today());
      Service=yrdif(to_double(DateOfHire),Today());
```

```
      if anydigit(substr(JobCode,3,1)) gt 0 then
            Level=inputn(substr(JobCode,3,1),1.);
        else Level=.; /**3**/

      if age ge 65 or (age ge 60 and Service ge 30) then RetEligible='Y';
        else RetEligible='N';

      select(level);
        when(1) salary=1.02*salary;
        when(2) salary=1.015*salary;
        when(3) salary=1.01*salary;
        otherwise salary=1.0175*salary;
      end;
    end;
  enddata;
 run;
quit;
```

Note the effects of changes at each of the commented positions:

1. RetEligible is declared as global so this flag variable is properly declared and placed into the final data set.
2. The Age, Service, and Level variables are explicitly declared; however, since they are not wanted in the final data set, they are set up to be local to the RUN method.
3. To control the conversion more robustly when using the INPUTN function, some conditioning is employed.

An attempt at a variation on the process above is done in the next example, adding a Raise and NewSalary variable included in the final data set, with formats and labels:

```
proc ds2;
  data empsB(overwrite=yes);
    declare char(1) RetEligible;
    declare double raise;
    declare double NewSalary;
    method run();
      declare double age;
      declare double service;
      declare integer level;
      set DS2HOW.employees;
      Age=yrdif(to_double(DateOfBirth),Today());
      Service=yrdif(to_double(DateOfHire),Today());
      if anydigit(substr(JobCode,3,1)) gt 0
        then Level=inputn(substr(JobCode,3,1),1.);
        else Level=.;
      if age ge 65 or (age ge 60 and Service ge 30) then RetEligible='Y';
        else RetEligible='N';
      select(level);
        when(1) raise=1.02*salary;
        when(2) raise=1.015*salary;
        when(3) raise=1.01*salary;
        otherwise raise=1.0175*salary;
      end;
      NewSalary=Salary+Raise;
      format NewSalary Raise dollar12.2;
      label NewSalary='Updated Salary';
    end;
```

```
  enddata;
 run;
quit;
```

Unfortunately, this generates errors (not shown) because the LABEL and FORMAT statements are not supported in PROC DS2 as they are in the DATA step (you might consider adding them to the INIT or TERM methods, but they are not supported anywhere in PROC DS2). Label and format definitions are actually available as part of the variable declaration, using the HAVING clause, as shown in the following code:

```
proc ds2;
  data empsB(overwrite=yes);
    declare char(1) RetEligible;
    declare double Raise having format dollar12.2;
    declare double NewSalary having format dollar12.2 label 'Updated Salary';
    method run();
      declare double age;
      declare double service;
      declare integer level;
      set DS2HOW.employees;
      Age=yrdif(to_double(DateOfBirth),Today());
      Service=yrdif(to_double(DateOfHire),Today());
      if anydigit(substr(JobCode,3,1)) gt 0
        then Level=inputn(substr(JobCode,3,1),1.);
        else Level=.;
      if age ge 65 or (age ge 60 and Service ge 30) then RetEligible='Y';
        else RetEligible='N';
      select(level);
        when(1) Raise=.02*salary;
        when(2) Raise=.015*salary;
        when(3) Raise=.01*salary;
        otherwise Raise=.0175*salary;
      end;
      NewSalary=Salary+Raise;
    end;
  enddata;
 run;
quit;
```

USER-DEFINED METHODS

Some of you may be users of PROC FCMP for defining functions and/or call routines, and PROC DS2 can use these (though it is not covered here). However, it is also possible to define methods and packages within DS2 itself to provide such functionality. The following example revisits the previous one, defining some of the computations via methods:

```
proc ds2;
  data empsC(overwrite=yes);
    declare char(1) RetEligible;
    declare double Raise having format dollar12.2;
    declare double NewSalary having format dollar12.2 label 'Updated Salary';

    method retire(double age, double serve) returns char;
      declare char(1) Retire;
      if age ge 65 or (age ge 60 and serve ge 30) then Retire='Y';
        else Retire='N';
      return Retire;
    end;
```

```
    method Raise(double salary, integer group, double rate1, double rate2,
                 double rate3, double rate0) returns double;
      select(group);
        when(1) Raise=rate1*salary;
        when(2) Raise=rate2*salary;
        when(3) Raise=rate3*salary;
        otherwise Raise=rate0*salary;
      end;
      return Raise;
    end;

    method run();
      declare double age;
      declare double service;
      declare integer level;
      set DS2HOW.employees;
      Age=yrdif(to_double(DateOfBirth),Today());
      Service=yrdif(to_double(DateOfHire),Today());
      RetEligible=Retire(Age,Service);
      if anydigit(substr(JobCode,3,1)) gt 0
        then Level=inputn(substr(JobCode,3,1),1.);
        else Level=.;
      Raise=Raise(Salary,Level,0.02,0.015,0.01,0.0175);
      NewSalary=Salary+Raise;
    end;
  enddata;
 run;
quit;
```

Each method defined, Retire and Raise, uses the general form of the opening statement:

```
method method-name(parameter-list) returns type;
```

This names the method, gives a list of parameters to pass with their types, and the type of value the method returns. The method contains the necessary programming statements to complete the process, and a return statement for the value the function produces. Note that the Retire method has a declaration for the variable it returns, Retire, but the same is not true for the Raise method. If the declaration for the Retire variable is removed, a warning is generated. Based on the concepts discussed thus far, what is the reason for this seemingly contradictory behavior?

The next program considers reverting to the case where only the updated salary is desired. While there are several ways to achieve this, methods can be written to update parameters passed to them by setting those as IN_OUT parameters:

```
proc ds2;
  data empsD(overwrite=yes);
    declare char(1) RetEligible;

    method retire(double age, double serve) returns char;
      declare char(1) Retire;
      if age ge 65 or (age ge 60 and serve ge 30) then Retire='Y';
        else Retire='N';
      return Retire;
    end;

    method Raise(in_out double salary, integer group, double rate1,
                 double rate2, double rate3, double rate0);
      select(group);
```

```
      when(1) Salary=(1+rate1)*salary;
      when(2) Salary=(1+rate2)*salary;
      when(3) Salary=(1+rate3)*salary;
      otherwise Salary=(1+rate0)*salary;
    end;
  end;

  method run();
    declare double age;
    declare double service;
    declare integer level;
    set DS2HOW.employees;
    Age=yrdif(to_double(DateOfBirth),Today());
    Service=yrdif(to_double(DateOfHire),Today());
    RetEligible=Retire(Age,Service);
    if anydigit(substr(JobCode,3,1)) gt 0
      then Level=inputn(substr(JobCode,3,1),1.);
      else Level=.;
    Raise(Salary,Level,0.02,0.015,0.01,0.0175);
  end;
 enddata;
run;
quit;
```

Note there is no RETURN statement in the Raise method at this point since the IN_OUT defines the returned value and, for the same reason, the use of the Raise method in the RUN method does not require an assignment statement. Note that it may also be helpful to update the format on the Salary variable via a global DECLARE statement.

Suppose it is desired to send the records for those eligible for retirement to one data set, and those not to another. The DATA RUN-group supports multiple data sets, so the following code attempts to achieve this:

```
proc ds2;
  data Retire Not/overwrite=yes;
    declare char(1) RetEligible;
    method retire(double age, double serve) returns char;
      declare char(1) Retire;
      if age ge 65 or (age ge 60 and serve ge 30) then Retire='Y';
        else Retire='N';
      return Retire;
    end;

    method Raise(in_out double salary, integer group, double rate1,
                 double rate2, double rate3, double rate0);
      select(group);
        when(1) Salary=(1+rate1)*salary;
        when(2) Salary=(1+rate2)*salary;
        when(3) Salary=(1+rate3)*salary;
        otherwise Salary=(1+rate0)*salary;
      end;
    end;

    method run();
      declare double age;
      declare double service;
      declare integer level;
```

```
      set DS2HOW.employees;
      Age=yrdif(to_double(DateOfBirth),Today());
      Service=yrdif(to_double(DateOfHire),Today());
      if anydigit(substr(JobCode,3,1)) gt 0
        then Level=inputn(substr(JobCode,3,1),1.);
        else Level=.;
      Raise(Salary,Level,0.02,0.015,0.01,0.0175);
      if retire(age,service) eq 'Y' then output retire;
        else output NonRetire;
    end;
  enddata;
 run;
quit;
```

Of course, there is some pitfall here judging by this extraction from the log:

```
ERROR: Compilation error.
ERROR: Parse encountered expression when expecting ';'.
ERROR: Line ####: Parse failed: data Retire  >>> Not <<< /overwrite=yes;
```

This is probably one of the more tricky notions when going from the DATA step to PROC DS2—in PROC DS2 keywords are reserved words. Not, being a keyword for comparison operations, cannot be used as a table name in this context. Change that data set name to something that is not reserved and everything works fine.

USER-DEFINED PACKAGES

These final examples are somewhat complex, but show the power available when defining a package within PROC DS2 (and, of course, there is much more power that DS2 offers that is not explored here). The complete code for this package is given in the appendix, but is broken into smaller pieces here. First, PROC DS2 is invoked and the PACKAGE statement defines a name and location for the package (an ENDPACKAGE statement will appear before the step boundary):

```
proc ds2;
package sasuser.EmployeeStuff/overwrite=yes;
```

Package replacement follows similar rules to table replacement (indeed package information is stored in a table), so the overwrite option may be necessary, but it cannot be given as a table option and must be given after the /. Other components of this package are methods, several of which use the same name but with distinct parameter lists—a concept known as method overloading. The first two methods have no parameters:

```
method EmployeeStuff();
  put;
  put 'Methods Available in EmployeeStuff:';
  put '  Raise';
  put '  Retire';
  put;
end;
method raise();
  put;
  put 'RAISE Method';
  put 'General Syntax: Raise(Salary,Group,List of Rate Parameter) or Raise(Salary,rate)';
  put '  Salary is double for input and is updated for output';
  put '  Group is integer';
  put '  Rates are double, number of parameters is one more than levels of group available--';
  put '  Levels of group can be 1, 1 and 2, or 1 and 2 and 3';
  put '  Rates are given in sequence for each level plus another for all other categories';
  put 'Without a group, a single rate is given to apply to all records';
  put ;
end;
```

Defining methods without parameters allows for calls to documentations of them to be made. Giving one the same name as the package automatically sends that information to the log when the package is declared in subsequent DS2 programs. Other methods look more like functions defined previously, here are the other methods associated with the name Raise:

```
method Raise(in_out double salary, integer group, double rate1, double rate2,
             double rate3, double rate0);
  select(group);
    when(1) Salary=(1+rate1)*salary;
    when(2) Salary=(1+rate2)*salary;
    when(3) Salary=(1+rate3)*salary;
    otherwise Salary=(1+rate0)*salary;
  end;
end;
method Raise(in_out double salary, integer group, double rate1, double rate2,
             double rate0);
  select(group);
    when(1) Salary=(1+rate1)*salary;
    when(2) Salary=(1+rate2)*salary;
    otherwise Salary=(1+rate0)*salary;
  end;
end;
method Raise(in_out double salary, integer group, double rate1,
             double rate0);
  select(group);
    when(1) Salary=(1+rate1)*salary;
    otherwise Salary=(1+rate0)*salary;
  end;
end;
method Raise(in_out double salary, double rate0);
  Salary=(1+rate0)*salary;
end;
```

It is important that the parameter sets are sufficiently distinct when overloading so that there is no possibility of ambiguity among the method definitions with the same name. A set of Retire methods are also defined—a bit better than the setup of this method in the previous section:

```
method retire();
  put 'RETIRE Method';
  put 'General Syntax: Retire(Age, AgeLim <,Serve, AgeServLim, ServLim>)';
  put '  Age is double, employee age';
  put '  AgeLim is double, cutoff for age-based retirement eligibility';
  put '  Serve is optional years of service parameter, also requires:';
  put '    AgeServLim: age cutoff when service is included together with...';
  put '    ServLim: Years of service required in conjuction with age';
end;
method retire(double age, double agelim, double serve, double ageservlim,
                  double servlim) returns char;
  declare char(1) Retire;
  if age ge agelim or (age ge ageservlim and serve ge servlim)
      then Retire='Y';  else Retire='N';
  return Retire;
end;
method retire(double age, double agelim) returns char;
  declare char(1) Retire;
  if age ge agelim then Retire='Y'; else Retire='N';
  return Retire;
```

```
end;
```

Again, a null version of the method exists to provide information the user can request, and two other versions of the method are stacked, one including a service time component, the other not. The package definition concludes with some housekeeping:

```
endpackage;
run;
quit;
```

To use the package in another PROC DS2 program, it must be declared (including a name) much like a variable is, and references to its methods are given in two levels—*package-name.method.* The following program reproduces previous results using the package definitions:

```
proc ds2;
  data Retire2(overwrite=yes) NonRetire2(overwrite=yes);
    declare package sasuser.EmployeeStuff ES();
    method init();
      ES.Raise();
      ES.Retire();
    end;

    method run();
      declare double age;
      declare double service;
      declare integer level;
      set DS2HOW.employees;
      Age=yrdif(to_double(DateOfBirth),Today());
      Service=yrdif(to_double(DateOfHire),Today());
      if anydigit(substr(JobCode,3,1)) gt 0
         then Level=inputn(substr(JobCode,3,1),1.);
        else Level=.;
      ES.Raise(Salary,Level,0.02,0.015,0.01,0.0175);
      if ES.retire(age,65,service,60,30) eq 'Y' then output retire2;
        else output NonRetire2;
    end;
  enddata;
 run;
quit;
```

Check the log to see that the documentation of the package is shown via the use of the empty method calls in the INIT method (these can be placed in the RUN method, but they will appear many times…).

Of course, any of the methods can be used as defined, as the final example illustrates:

```
proc ds2;
  data Retire3(overwrite=yes) NonRetire3(overwrite=yes);
    declare char(1) RetEligible;
    declare package sasuser.EmployeeStuff ES();

    method run();
      declare double age;
      declare double service;
      declare integer level;
      set DS2HOW.employees;
      Age=yrdif(to_double(DateOfBirth),Today());
      Service=yrdif(to_double(DateOfHire),Today());
```

```
      if anydigit(substr(JobCode,3,1)) gt 0
        then Level=inputn(substr(JobCode,3,1),1.);
        else Level=.;
      ES.Raise(Salary,0.01);
      if ES.retire(age,62) eq 'Y' then output retire3;
        else output NonRetire3;
    end;
  enddata;
 run;
quit;
```

CONCLUSION

Transitioning from the DATA step to PROC DS2 is not simple, and seemingly minor issues can frustrate you along the way. But the value and power of PROC DS2 is high if you can exploit it, so hopefully some of the concepts discussed here will help you make the transition. Blogs and a book (see below) by Mark Jordan are highly recommended if you plan to go down this path.

RECOMMENDED READING

- *Mastering the SAS® DS2 Procedure,* Mark Jordan, SAS Institute, Cary, NC, 2018

CONTACT INFORMATION

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

James E. Blum
University of North Carolina Wilmington
blumj@uncw.edu
http://people.uncw.edu/blumj

APPENDIX—EMPLOYEESTUFF PACKAGE CODE

```
proc ds2;
package sasuser.EmployeeStuff/overwrite=yes;
method EmployeeStuff();
  put;
  put 'Methods Available in EmployeeStuff:';
  put '  Raise';
  put '  Retire';
  put;
end;
method raise();
  put;
  put 'RAISE Method';
  put 'General Syntax: Raise(Salary,Group,List of Rate Parameter) or Raise(Salary,rate)';
  put '  Salary is double for input and is updated for output';
  put '  Group is integer';
  put '  Rates are double, number of parameters is one more than levels of group available--';
  put '  Levels of group can be 1, 1 and 2, or 1 and 2 and 3';
  put '  Rates are given in sequence for each level plus another for all other categories';
  put 'Without a group, a single rate is given to apply to all records';
  put ;
end;
method Raise(in_out double salary, integer group, double rate1, double rate2, double rate3,
double rate0);
  select(group);
    when(1) Salary=(1+rate1)*salary;
```

```
    when(2) Salary=(1+rate2)*salary;
    when(3) Salary=(1+rate3)*salary;
    otherwise Salary=(1+rate0)*salary;
  end;
end;
method Raise(in_out double salary, integer group, double rate1, double rate2, double rate0);
  select(group);
    when(1) Salary=(1+rate1)*salary;
    when(2) Salary=(1+rate2)*salary;
    otherwise Salary=(1+rate0)*salary;
  end;
end;
method Raise(in_out double salary, integer group, double rate1, double rate0);
  select(group);
    when(1) Salary=(1+rate1)*salary;
    otherwise Salary=(1+rate0)*salary;
  end;
end;
method Raise(in_out double salary, double rate0);
  Salary=(1+rate0)*salary;
end;

method retire();
  put;
  put 'RETIRE Method';
  put 'General Syntax: Retire(Age, AgeLim <,Serve, AgeServLim, ServLim>)';
  put '  Age is double, employee age';
  put '  AgeLim is double, cutoff for age-based retirement eligibility';
  put '  Serve is optional years of service parameter, also requires:';
  put '     AgeServLim: age cutoff when service is included together with...';
  put '     ServLim: Years of service required in conjuction with age';
  put ;
end;
method retire(double age, double agelim, double serve, double ageservlim, double servlim) returns
char;
  declare char(1) Retire;
  if age ge agelim or (age ge ageservlim and serve ge servlim) then Retire='Y';
    else Retire='N';
  return Retire;
end;
method retire(double age, double agelim) returns char;
  declare char(1) Retire;
  if age ge agelim then Retire='Y';
    else Retire='N';
  return Retire;
end;
endpackage;
run;
quit;
```