Modernizing Legacy SAS® Applications and Program Code

Kirk Paul Lafler, Software Intelligence Corporation
Clark Roberts, Decision Analytics

Abstract
Whether you are a novice or experienced SAS® programmer with responsibility for the support of your organization’s legacy applications, programs and code, assistance is available to help modernize and streamline code for the 21st century and beyond. This paper and presentation explores the available constructs, statements, functions, algorithms, operators, methods, expressions, programming techniques and approaches users have to update and modernize legacy applications, programs and code first introduced as far back as the 1970’s. Attendees learn how to use the SCAPROC procedure – the SAS code analyzer – to analyze metadata about the contents of SAS code, and to streamline, scale and modernize code constructs, statements, functions, algorithms, and legacy applications and program code.

Table of Contents

Abstract .................................................................................................................................................. 1
Table of Contents .................................................................................................................................. 1
Introduction ........................................................................................................................................ 3
What are Legacy Applications and Program Code? ................................................................. 3
Signs that a Legacy Application May Need Modernizing .......................................................... 3
Conditional Logic Scenarios ............................................................................................................ 4
  Conditional Logic with IF-THEN / ELSE......................................................................................... 4
  Conditional Logic with SELECT-WHEN / OTHERWISE .............................................................. 5
  Conditional Logic with CASE Expressions ...................................................................................... 6
Subsetting with WHERE Expressions in a PROCedure .............................................................. 8
Using the IN Operator for Comparisons ....................................................................................... 9
Concatenating Strings and Variables with CAT Functions ........................................................ 9
Concatenating (or Appending) Data Sets ....................................................................................... 11
  Concatenating with the DATA-SET-RUN Construct .................................................................. 11
  Concatenating with a PROC SQL Outer Union CORR ............................................................. 11
  Concatenating with PROC APPEND (or PROC DATASETS – APPEND Statement) .......... 12
Processing Multiple TABLE Statements with PROC FREQ .................................................... 12
List of Procedures Supporting a CLASS Statement ................................................................. 13
Producing Page Numbers with ODS RTF Pagination Functions ........................................... 13
  Page Counters with ODS RTF Functions ...................................................................................... 14
Automating the Process of Creating Multiple HTML Files ....................................................... 15
Automating the Process of Creating Multiple Excel Files ........................................................ 16
Discovering the Number of Occurrences of a Value in a Data Set ........................................... 17
  Discovering the Number of Occurrences of a Value in a DATA Step ...................................... 17
  Discovering the Number of Occurrences of a Value with the PROC FREQ NLEVELS Option .............................................................................................................................. 18
Discovering the Number of Occurrences of a Value with PROC SQL ..................................................18

**Using Metadata to Determine the Number of Observations in a Data Set** ...........................................19
Older Methods of Determining the Number of Observations in a Data Set ............................................19
Using DICTIONARY.TABLES Metadata to Determine the Number of Observations in a Data Set .................20
Using SASHELP.VTABLE Metadata to Determine the Number of Observations in a Data Set ....................21

**Using PROC PRINT with Style** .............................................................................................................21

**Using Available Memory with Hash Object Programming** .................................................................23

**User-developed Macros** ......................................................................................................................24

**PC Windows Environment** ..................................................................................................................25

**User-developed Functions with the FCMP and PROTO Procedures** .....................................................28

**PROC DS2** ........................................................................................................................................29
  Methods................................................................................................................................................33
  Packaging............................................................................................................................................33
  Built-in Packages (refer to Jordan 2016 for detailed coverage of built-in packages):.................................33
  User Defined Packages........................................................................................................................36

**Parallel Processing** .............................................................................................................................38

**SAS Grid** ..........................................................................................................................................40

**Exploring PROC SCAPROC – The SAS Code Analyzer** .................................................................40

**Conclusion** ..........................................................................................................................................42

**References** ..........................................................................................................................................42
  **DS2 References and Suggested Reading** ..........................................................................................42
  **Efficiency and Performance Tuning References and Suggested Reading** ........................................43
  **General References and Suggested Reading** ...................................................................................43
  **Hash Object References and Suggested Reading** ...........................................................................44
  **Macro References and Suggested Reading** .....................................................................................44
  **Operations Research (OR) References and Suggested Reading** .....................................................44
  **PROC FCMP References and Suggested Reading** .......................................................................44
  **PROC SCAPROC References and Suggested Reading** ...............................................................45
  **SAS Grid References and Suggested Reading** .................................................................................45
  **SAS Programming Techniques References and Suggested Reading** ............................................45
  **Text Analytics References and Suggested Reading** ........................................................................46
  **WIN32API References and Suggested Reading** ..............................................................................46

**Acknowledgments** ...............................................................................................................................47

**Trademark Citations** ...........................................................................................................................47

**Data Sets Used in Examples** ...............................................................................................................47

**Author Bios** ....................................................................................................................................50
Introduction
As SAS users around the world celebrate a milestone of more than 40-years using SAS software, organizations want, and need, to look at ways to modernize their inventories of legacy SAS applications and program code to operate in the 21\textsuperscript{st} century and beyond. This means that IT personnel, systems and data analysts, SAS\textsuperscript{®} programmers, end-users, management, and stakeholders everywhere must assume the responsibility of identifying, and modernizing, mission-critical legacy applications and program code, with newer, and more scalable and efficient, statements, functions, options, code constructs, algorithms, and programming techniques. This paper explores many areas for consideration, and provides tips, techniques and examples to help guide SAS users through the modernization process. We suggest and illustrate a foundation of technologies, techniques, and approaches to consider while modernizing applications and program code. We also provide an example on the use of the SCAPROC procedure – the SAS code analyzer – to analyze metadata about the contents of SAS code, and to streamline, scale and modernize code constructs, algorithms, functions, and legacy application program code.

What are Legacy Applications and Program Code?
Programmers and application developers have different interpretations and meanings for what legacy code means. For some legacy applications and program code refers to code that someone else wrote a long time ago and as a result may not utilize or represent the latest technologies. Complicating matters further, often, the original developer(s) and/or programmer(s) is/are no longer available or affiliated with the organization. For others, legacy code represents a foundation of older, and often outdated, statements, functions, options, coding constructs, algorithms and other techniques causing applications and program code to become woefully behind 21\textsuperscript{st} century standards.

Other characteristics attributed to legacy code are the unlimited number of coding styles and modifications that occurs over time. This often translates into a legacy code base that may have been originally well-written but evolves into a complex, and less than maintainable, maze of spaghetti code. In these situations the legacy code base is no longer engineered but begins to take on the characteristics of a patched and tangled control structure. Making matters worse, this maze of patched, complex and confusing legacy code often survives without the existence of up-to-date and effective program documentation.

Signs that a Legacy Application May Need Modernizing
Green (2017) describes modernization as, “... newer, faster, sleeker and more useful – all qualities we want our applications to have.” Modernizing legacy applications and program code involves an incremental and structured approach. It consists of identifying the target application and program code; selecting potential solutions to use; and finally, implementing structured and scalable solutions to replace varied coding styles and conventions made over its functional life. So, how does an organization know when an application needs modernizing? Green shares five signs to answer this question.

1. Operation and maintenance costs are high.
2. It’s clunky or uses outdated technologies.
3. Your business processes have changed.
4. There’s no tight integration with future applications.
5. It’s not mobile-ready.

So, how should an organization proceed with the modernization of a legacy application and program code project? The best place to start is to get all stakeholders on-board and in agreement with the objectives and changes to be made. Next, obtain the necessary funding for performing the project work, modernizing a legacy application conjures concern from everyone involved. All too often, stakeholders develop a, “If it’s not broke then leave it alone!” attitude. To help alleviate the issues associated with modernizing legacy applications and program code, we recommend a five-step modernization approach.

1. Identify mission-critical applications and program code that is/are indispensable to the organization.
2. Review and understand the code associated with the user-interface, the data sources being accessed, the processing requirements, and finally the output and results.
3. Identify and modernize older technologies; hard to modify and inflexible code; and inefficient statements, functions, options and their settings, code constructs, algorithms, and programming techniques with newer and more efficient methods and techniques.

4. Test, Train and Deploy the modernized application and program code to bring all stakeholders on board.

5. Maintain and Support the modernized applications and program code to ensure their flexibility and adaptability to changing requirements, environments and technologies.

**Conditional Logic Scenarios**

A powerful and necessary programming technique in the SAS® software is its ability to perform different actions depending on whether a programmer-specified condition evaluates to true or false. The method for accomplishing this is to use one or more conditional statements, expressions, and constructs to build a level of intelligence in a program or application. Conditional logic scenarios in the DATA step are frequently implemented using IF-THEN / ELSE and SELECT statements. The SQL procedure also supports logic scenarios and is implemented with a coding technique known as a CASE expression.

**Conditional Logic with IF-THEN / ELSE**

The IF-THEN / ELSE construct in the DATA step enables a sequence of conditions to be assigned that when executed proceeds through the sequence of logic conditions until a match in an expression is found or until all conditions are exhausted. The example shows a character variable Movie_Length being assigned a value of either “Shorter Length”, “Average Length”, or “Longer Length” based on the mutually exclusive conditions specified in the IF-THEN and ELSE conditions. Although not required, an ELSE condition serves as an effective technique for continuing processing to the next specified condition when a match is not found. An ELSE condition can also be useful as a “catch-all” to prevent a missing value from being assigned.

```
LIBNAME MYDATA “E:/WORKSHOPS/WORKSHOP DATA”;
DATA IF_THEN_EXAMPLE;
   ATTRIB Movie_Length LENGTH=$14 LABEL='Movie Length';
   SET MYDATA.MOVIES;
   IF LENGTH < 120 THEN Movie_Length = ’Shorter Length’;
   ELSE IF LENGTH > 160 THEN Movie_Length = ’Longer Length’;
   ELSE Movie_Length = ’Average Length’;
RUN;
PROC PRINT DATA=IF_THEN_EXAMPLE NOOBS;
   VAR TITLE LENGTH Movie_Length;
RUN;
```
Conditional Logic with SELECT-WHEN / OTHERWISE
Another form of conditional logic available to users is a SELECT statement. Its purpose is to enable a sequence of logic conditions to be constructed in a DATA step by specifying one or more WHEN conditions and an optional OTHERWISE condition. When executed, processing continues through each WHEN condition until a match is found that satisfies the specified expression. Typically one or more WHEN conditions are specified in descending frequency order representing a series of conditions. The next example shows a value based on the mutually exclusive conditions specified in the sequence of logic conditions of “Shorter Length”, “Average Length”, or “Longer Length” being assigned to the character variable Movie_Length. Although not required, the OTHERWISE condition can be useful in the assignment of a specific value or as a “catch-all” to prevent a missing value from being assigned.

SELECT-WHEN / OTHERWISE Code:

LIBNAME MYDATA “E:/WORKSHOPS/WORKSHOP DATA” ;
DATA SELECT_EXAMPLE ;
  SET MYDATA.MOVIES ;
  SELECT ;
    WHEN (LENGTH < 120) Movie_Length = ‘Shorter Length’ ;
    WHEN (LENGTH > 160) Movie_Length = ‘Longer Length’ ;
  OTHERWISE Movie_Length = ‘Average Length’ ;
  END ;
RUN ;

PROC PRINT DATA=SELECT_EXAMPLE NOOBS ;
  VAR TITLE LENGTH Movie_Length ;
RUN ;
Conditional Logic with CASE Expressions
Another form of conditional logic available to users is a case expression. Its purpose is to provide a way of conditionally selecting result values from each row in a table (or view). Similar to an IF-THEN/ELSE or SELECT construct in the DATA step, a case expression can only be specified in the SQL procedure. It supports a WHEN-THEN clause to conditionally process some but not all the rows in a table. An optional ELSE expression can be specified to handle an alternative action should none of the expression(s) identified in the WHEN condition(s) not be satisfied. A case expression must be a valid SQL expression and conform to syntax rules similar to DATA step SELECT-WHEN statements. Even though this topic is best explained by example, a quick look at the syntax follows.

```
CASE <column-name>
   WHEN when-condition THEN result-expression
   <WHEN when-condition THEN result-expression> ...
   <ELSE result-expression>
END
```

A column-name can optionally be specified as part of the CASE-expression. If present, it is automatically made available to each when-condition, and is classified as a simple case expression. When it is not specified, the column-name must be coded in each when-condition, and is classified as a searched case expression. If a when-condition is satisfied by a row in a table (or view), then it is considered “true” and the result-expression following the THEN keyword is processed. The remaining WHEN conditions in the case expression are skipped. If a when-condition is “false”, the next when-condition is evaluated. SQL evaluates each when-condition until a “true” condition is found or in the event all when-conditions are “false”, it then executes the ELSE expression and assigns its value to the CASE expression’s result. A missing value is assigned to a case expression when an ELSE expression is not specified and each when-condition is “false”.

In the next example, a searched case expression is illustrated. A searched case expression in the SQL procedure provides users with the capability to perform more complex comparisons. Although the number of keystrokes can be more than with a simple case expression, the searched case expression offers the greatest flexibility and is the primary form used by SQL’ers. The
noticeable absence of a column name as part of the case expression permits any number of columns to be specified from the underlying table(s) in the WHEN-THEN/ELSE logic scenarios.

The next example shows a searched case expression being used to assign the character variable Movie_Length with the AS keyword. A value of “Shorter Length” for movie lengths less than 120 minutes, “Longer Length” for movie lengths greater than 160 minutes, or “Average Length” for all other movie lengths is assigned to the newly created column. Although not required, an ELSE condition can be useful in the assignment of a specific value or as a “catch-all” to prevent a missing value from being assigned.

```
LIBNAME MYDATA "E:/WORKSHOPS/WORKSHOP DATA" ;
PROC SQL;
  SELECT TITLE,
       LENGTH,
       CASE
         WHEN LENGTH < 120 THEN 'Shorter Length'
         WHEN LENGTH > 160 THEN 'Longer Length'
         ELSE 'Average Length'
       END AS Movie_Length
  FROM MYDATA.MOVIES ;
QUIT ;
```

As previously mentioned, searched case expressions provide users with the capability to perform more complex logic comparisons. Combined with logical and comparison operators, searched case expressions along with their WHERE clause counterparts, provide the capabilities to construct complex logic scenarios. In the next example a listing of “Action” and “Comedy” movies are displayed. Using a searched case expression, a value of “Shorter Length” for movie lengths less than 120 minutes, “Longer Length” for movie lengths greater than 160 minutes, or “Average Length” for all other movie lengths is assigned to the newly created column. A column heading of Movie_Type is assigned to the new column with the AS keyword.

```
Searched CASE Expression Code:
LIBNAME MYDATA "E:/WORKSHOPS/WORKSHOP DATA" ;
PROC SQL;
  SELECT TITLE,
       LENGTH,
       CASE
         WHEN LENGTH < 120 THEN 'Shorter Length'
         WHEN LENGTH > 160 THEN 'Longer Length'
         ELSE 'Average Length'
       END AS Movie_Type
  FROM MYDATA.MOVIES
;QUIT ;

Searched CASE Expression Results:
```

As previously mentioned, searched case expressions provide users with the capability to perform more complex logic comparisons. Combined with logical and comparison operators, searched case expressions along with their WHERE clause counterparts, provide the capabilities to construct complex logic scenarios. In the next example a listing of “Action” and “Comedy” movies are displayed. Using a searched case expression, a value of “Shorter Length” for movie lengths less than 120 minutes, “Longer Length” for movie lengths greater than 160 minutes, or “Average Length” for all other movie lengths is assigned to the newly created column. A column heading of Movie_Type is assigned to the new column with the AS keyword.
**Searched CASE Expression Code:**

```
LIBNAME MYDATA "E:/WORKSHOPS/WORKSHOP DATA" ;
PROC SQL;
SELECT TITLE, RATING, LENGTH, CATEGORY, CASE WHEN UPCASE(CATEGORY) CONTAINS 'ACTION' AND LENGTH < 120 THEN 'Action Short' WHEN UPCASE(CATEGORY) CONTAINS 'ACTION' AND LENGTH > 160 THEN 'Action Long' WHEN UPCASE(CATEGORY) CONTAINS 'ACTION' AND LENGTH BETWEEN 120 AND 160 THEN 'Action Medium' WHEN UPCASE(CATEGORY) CONTAINS 'COMEDY' AND LENGTH < 120 THEN 'Comedy Short' WHEN UPCASE(CATEGORY) CONTAINS 'COMEDY' AND LENGTH > 160 THEN 'Comedy Long' WHEN UPCASE(CATEGORY) CONTAINS 'COMEDY' AND LENGTH BETWEEN 120 AND 160 THEN 'Comedy Medium' ELSE 'Not Interested' END AS MOVIE_TYPE FROM MYDATA.MOVIES WHERE UPCASE(CATEGORY) CONTAINS 'ACTION' OR 'COMEDY'; QUIT;
```

**Searched CASE Expression Results:**

<table>
<thead>
<tr>
<th>Title</th>
<th>Rating</th>
<th>Length</th>
<th>Category</th>
<th>MOVIE_TYPE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brave Heart</td>
<td>R</td>
<td>177</td>
<td>Action Adventure</td>
<td>Action Long</td>
</tr>
<tr>
<td>Casablanca</td>
<td>PG</td>
<td>103</td>
<td>Drama</td>
<td>Not Interested</td>
</tr>
<tr>
<td>Christmas Vacation</td>
<td>F6-13</td>
<td>97</td>
<td>Comedy</td>
<td>Comedy Short</td>
</tr>
<tr>
<td>Coming to America</td>
<td>R</td>
<td>115</td>
<td>Comedy</td>
<td>Comedy Short</td>
</tr>
<tr>
<td>Dracula</td>
<td>R</td>
<td>130</td>
<td>Horror</td>
<td>Not Interested</td>
</tr>
<tr>
<td>Dressed to Kill</td>
<td>R</td>
<td>105</td>
<td>Drama Mysteries</td>
<td>Not Interested</td>
</tr>
<tr>
<td>Forrest Gump</td>
<td>F6-13</td>
<td>142</td>
<td>Drama</td>
<td>Not Interested</td>
</tr>
<tr>
<td>Ghost</td>
<td>F6-13</td>
<td>127</td>
<td>Drama Romance</td>
<td>Not Interested</td>
</tr>
<tr>
<td>Jaws</td>
<td>F5</td>
<td>125</td>
<td>Action Adventure</td>
<td>Action Medium</td>
</tr>
<tr>
<td>Jurassic Park</td>
<td>F6-13</td>
<td>127</td>
<td>Action</td>
<td>Action Medium</td>
</tr>
<tr>
<td>Lethal Weapon</td>
<td>R</td>
<td>115</td>
<td>Action Cops &amp; Robber</td>
<td>Action Short</td>
</tr>
<tr>
<td>Michael</td>
<td>F6-13</td>
<td>100</td>
<td>Drama</td>
<td>Not Interested</td>
</tr>
<tr>
<td>National Lampoon's Vacation</td>
<td>F6-13</td>
<td>98</td>
<td>Comedy</td>
<td>Comedy Short</td>
</tr>
<tr>
<td>Poltergeist</td>
<td>F5</td>
<td>115</td>
<td>Horror</td>
<td>Not Interested</td>
</tr>
<tr>
<td>Rocky</td>
<td>F5</td>
<td>120</td>
<td>Action Adventure</td>
<td>Action Medium</td>
</tr>
<tr>
<td>Scarface</td>
<td>R</td>
<td>170</td>
<td>Action Cops &amp; Robber</td>
<td>Action Long</td>
</tr>
<tr>
<td>Silence of the Lambs</td>
<td>R</td>
<td>112</td>
<td>Drama Suspense</td>
<td>Not Interested</td>
</tr>
<tr>
<td>Star Wars</td>
<td>F5</td>
<td>124</td>
<td>Action Sci-Fi</td>
<td>Action Medium</td>
</tr>
<tr>
<td>The Hunt for Red October</td>
<td>F6-13</td>
<td>135</td>
<td>Action Adventure</td>
<td>Action Medium</td>
</tr>
<tr>
<td>The Terminator</td>
<td>R</td>
<td>108</td>
<td>Action Sci-Fi</td>
<td>Action Short</td>
</tr>
<tr>
<td>The Wizard of Oz</td>
<td>G</td>
<td>101</td>
<td>Adventure</td>
<td>Not Interested</td>
</tr>
<tr>
<td>Titanic</td>
<td>F6-13</td>
<td>194</td>
<td>Drama Romance</td>
<td>Not Interested</td>
</tr>
</tbody>
</table>

**Subsetting with WHERE Expressions in a PROCedure**

Gupta (2006) describes using a subsetting-IF versus a WHERE-statement or WHERE= data set option to subset observations. To avoid using a subsetting-IF statement in a DATA step, SAS users may be able to specify a WHERE= data set option for subsetting purposes directly in a procedure. This approach prevents the creation of a data set and, as a result, is more likely to scale better by reducing CPU and I/O resources. Gupta emphasizes an important detail that all SAS users should know when specifying a WHERE condition in a procedure, “Multiple WHERE conditions within SAS procedures are not cumulative as they are in a DATA step meaning the most recent WHERE condition replaces any, and all, previously specified WHERE condition(s).”
Using the IN Operator for Comparisons

Legacy SAS applications and program code often use one, or more, OR comparison operators to handle logic scenarios. Although syntactically correct, a series of individual comparisons separated by an OR comparison operator is generally less efficient than using an IN operator. The reason is due to the way an IN operator operates. When an IN operator is specified, SAS stops making comparisons as soon as it finds a match. This is not the case with an OR operator. In the next example, a number of individual comparisons are specified using an OR operator.

In the next example, an IN operator is specified to help modernize the process of handling a number of individual comparisons. The IN operator provides a convenient, and concise, way to specify scenarios with many OR comparisons. A similar example using DS2 SQLSTM1 package approach can be found here.

Concatenating Strings and Variables with CAT Functions

SAS functions serve an essential role in the Base SAS software. Representing a variety of built-in and callable routines, functions serve as the “work horses” in the SAS software providing users with “ready-to-use” tools designed to ease the burden of writing and testing often lengthy and complex code for a variety of programming tasks. The advantage of using SAS functions is evident by their relative ease of use, and their ability to provide a more efficient, robust and scalable approach to simplifying a process or programming task. In this example, we show how the TRIM and LEFT functions along with the concatenate operator to concatenate strings and variables together can be replaced with the CAT functions.
**CAT Function Code:**

```sas
data _null_;  
length NUM 3. A B C D E $ 8 BLANK $ 1 ;  
A = 'The';  
NUM = 5 ;  
B = ' Cats';  
C = ' in';  
D = ' the';  
E = ' Hat';  
BLANK = ' ';  

*Old concatenation approach with TRIM and LEFT functions and concatenation operator ;  
OLD=trim(left(A)) || BLANK || trim(left(NUM)) || BLANK || trim(left(B)) || BLANK || trim(left(C)) || BLANK || trim(left(D)) || BLANK || trim(left(E)) ;

* Using the CAT functions to concatenate character and numeric values together ;  
CAT = cat (A, NUM, B, C, D, E) ;  
CATQ = catq(BLANK, A, NUM, B, C, D, E) ;  
CATS = cats(A, NUM, B, C, D, E) ;  
CATT = catt(A, NUM, B, C, D, E) ;  
CATX = catx(BLANK, A, NUM, B, C, D, E) ;  
put OLD= / STRIP= / CAT= / CATQ= / CATS= / CATT= / CATX= / ;  
run ;
```

**CAT Function Results:**

OLD=The 5 Cats in the Hat  
CAT=The 5 Cats in the Hat  
CATQ="The      " 5 " Cats       " in       " the       " Hat       
CATS=The5CatsintheHat  
CATT=The5 Catsin theHat  
CATX=The 5 Cats in the Hat

**Analysis:**

In the example, above, a single numeric variable, NUM, and six character variables: A, B, C, D, E, and BLANK are defined with their respective values as: NUM=5, A='The', B=' Cats', C=' in', D=' the', E=' Hat' and BLANK=' '. The oldest way of concatenating two or more strings or variables together is then specified, using the TRIM and LEFT functions with the concatenation operator "||" in an assignment statement. As an alternative, a newer and more robust concatenation approach is specified using the CAT family of functions: CAT, CATQ, CATS, CATT, and CATX.

1. **CAT**, the simplest of concatenation functions, joins two or more strings and/or variables together, end-to-end producing the same results as with the concatenation (double bar) operator.

2. **CATQ** is similar to the default features of the CATX function, but the CATQ function adds quotation marks to any concatenated string or variable.

3. **CATS** removes leading and trailing blanks and concatenates two or more strings and/or variables together.

4. **CATT** removes trailing blanks and concatenates two or more strings and/or variables together.
CATX, perhaps the most robust CAT function, removes leading and trailing blanks and concatenates two or more strings and/or variables together with a user-specified delimiter between each.

Concatenating (or Appending) Data Sets
Concatenating data sets is the process of combining two, or more, data sets, one after the other, with the purpose of creating a single data set. The number of observations in the new data set is the sum total of observations in all the original input data sets. The order of observations in the concatenated data set is arranged sequentially with the observations from the first data set, followed by the observations from the second data set, and so on. The concatenated data set contains the same variables as the input data sets. Should an input data set contain different variables from the other input data sets, the concatenated data set will have missing values assigned to the variables from the other input data sets.

Concatenating with the DATA-SET-RUN Construct
SAS provides users with a few ways to concatenate data sets. In the first example, below, an old-style DATA-SET construct is specified to concatenate the two data sets, RUGs_2015 and RUGs_2016. Although syntactically correct, this approach does not scale well because it forces SAS to incur heavy I/O (input/output) because the observations in each input data set must be read and written to the concatenated data set.

```
data Concatenated_Results ;
  set RUGs_2015
     RUGs_2016 ;
run ;
```

```
data Concatenated_Results ;
  set RUGs_2015
     RUGs_2016 ;
run ;
```

Concatenating with a PROC SQL Outer Union CORR
A second approach uses PROC SQL to concatenate data sets. In this next example, an OUTER UNION CORR set operator is specified, and SQL reads and processes the tables in each query producing a new concatenated table of results.

```
proc sql ;
  create table Concatenated_Results as
    select * from RUGs_2015
    outer union corr
    select * from RUGs_2016 ;
  select * from Concatenated_Results ;
quit ;
```

Concatenating with a PROC SQL Outer Union CORR
A second approach uses PROC SQL to concatenate data sets. In this next example, an OUTER UNION CORR set operator is specified, and SQL reads and processes the tables in each query producing a new concatenated table of results.
A third, and more efficient, concatenation approach is available to SAS users. Using PROC APPEND (or the APPEND statement in PROC DATASETS), an input data set can be appended to another data set. The advantage of using this approach is reduced I/O, since SAS does not have to read the observations in the base data set. Appending this way offers a way to scale an application. As the number of observations in the base data set grows, the advantage of using this approach can become huge. In the next example, two PROC APPENDs are specified to concatenate the observations in the RUGs_2015 and RUGs_2016 data sets.

```sas
proc append base=Concatenated_Results
   data=RUGs_2015;
run;
proc append base=Concatenated_Results
   data=RUGs_2016;
run;
```

### Processing Multiple TABLE Statements with PROC FREQ

Benjamin (2012) describes a common problem programmers have when using PROC FREQ to produce multiple table results. Programmers will often code two, or more, individual PROC FREQ and TABLE statements even for the same input data set. Although the PROC FREQ code, illustrated below, is syntactically correct, invoking PROC FREQ multiple times in this way can result in an increase in the amount of time for processing the request.
**PROC FREQ Code:**

```sas
proc freq data=sashelp.cars ;
   table Origin / list out=work.Origin_Freq1 ;
run ;
proc freq data=sashelp.cars ;
   table Origin * Type / list out=work.Origin_Freq2 ;
run ;
proc freq data=sashelp.cars ;
   table Origin * Type * Cylinders / list out=work.Origin_Freq3 ;
r
```

To optimize the code, programmers can force a single pass over the input data set and as a result reduce the amount of processing time needed to produce the resulting data sets, as follows.

**Optimized PROC FREQ Code:**

```sas
proc freq data=sashelp.cars ;
   table Origin / list out=work.Origin_Freq1 ;
   table Origin * Type / list out=work.Origin_Freq2 ;
   table Origin * Type * Cylinders / list out=work.Origin_Freq3 ;
run ;
```

**List of Procedures Supporting a CLASS Statement**

Procedures are classified as the “workhorses” in the SAS System. The CLASS statement specifies one, or more, character or numeric variables used to group data into classification levels. A virtue of using a CLASS statement is that a SORT procedure is not required to arrange and group the data, because the stats and other information is collected in memory and reported at the end of the procedure. A partial list of SAS procedures, below, supports the use of a CLASS statement.

<table>
<thead>
<tr>
<th><strong>SAS Procedures Supporting a CLASS Statement</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>PROC ANOVA</td>
</tr>
<tr>
<td>PROC DISCRIM</td>
</tr>
<tr>
<td>PROC GENMOD</td>
</tr>
<tr>
<td>PROC GLM</td>
</tr>
<tr>
<td>PROC LOGISTIC</td>
</tr>
</tbody>
</table>

**Producing Page Numbers with ODS RTF Pagination Functions**

Page numbering is the process of applying a sequence of numbers, Roman numerals, or letters on reports, spreadsheets, documents, books or other multi-page files. Legacy applications and program code frequently use counters or code routines to generate and display page numbers. Simple page numbering routines may resemble something similar to the following code.
DATA Step Code:

FILENAME REPORT DISK 'c:\DATA_NULL_Report.LST' ;
DATA _NULL_ ;
SET SASHELP.CARS END=EOF ;
FILE REPORT HEADER=H1 ; /* Execute Page_Header Routine */
PUT @1 Origin $6.
@10 Make $13.
@25 MSRP DOLLAR12.
RETURN ;

H1: ; /* Page Header */
Page_CTR + 1 ;
PUT @15 DATA _NULL_ Detail Report
  // @22 'Page Number ' Page_CTR ;
RETURN ;
RUN ;

Page numbers can be produced and displayed in RTF output by specifying an escape character with an ODS RTF statement, any of the following functions, and an ODS RTF CLOSE ; statement:

✓ {thispage}
✓ {lastpage}
✓ {pageof}

Page Counters with ODS RTF Functions
Output Delivery System (ODS) provides powerful features that users can use when producing output. In the next example, an escape character is specified with the ODS RTF destination, where the functions: {thispage}, {lastpage}, and {pageof} are specified in the title and footnote statements to produce the page numbers and the total number of pages in the report.

ODS RTF Code:

ods escapechar='^' ;
ods RTF file='c:\Print-Report.rtf' ;
proc print data=sashelp.cars noobs ;
title 'Page ^{thispage} of ^{lastpage}' ;
footnote '^{pageof}' ;
run ;
ods RTF close ;
Automating the Process of Creating Multiple HTML Files

The Web offers incredible potential that impacts all corners of society. With its increasing popularity as a communications medium, Web publishers have arguably established the Web as the greatest medium ever created. Businesses, government agencies, professional associations, schools, libraries, research agencies, and a potpourri of society’s true believers have endorsed the Web as an efficient means of conveying their messages to the world.

The SAS software provides users with the capability to create results and deploy selected pieces of output as HTML output files. Using the Output Delivery System (ODS) HTML destination, output can be created that anyone can view using a web browser. Syntactically correct HTML code is automatically produced and made ready for deployment using one of the Internet browser software products (e.g., Internet Explorer, Google Chrome, Mozilla FireFox, Safari, etc.). As a result, the SAS System and the HTML destination create a type of “streaming” or continuous output by adding elevator bars (horizontal and/or vertical) for easy navigation.

In the following example, redundant code and hardcoding issues are avoided by using PROC SQL to determine the number of unique (or distinct) values of the Origin column exist and once known are assigned to single-value and value-list macro variables. With the unique values assigned to two macro variables, an iterative %DO statement is specified to control the propagation of one, or more, HTML files containing one-way frequency results. The results of the three distinct HTML files that were created are also displayed, below.
/* Output HTML Files Location */
filename odsout "E:\";

options symbolgen;
%macro multfiles;
proc sql noprint;
select count(distinct origin) into :morigin_cnt /* derive number of origins */
from sashelp.cars
order by origin;
select distinct origin into :morigin_list separated by "~" /* derive unique origin values */
from sashelp.cars
order by origin;
quit;

%do i=1 %to &morigin_cnt;
ods html path=odsout (URL=NONE)
file=%SCAN(&morigin_list,&i,~)_FrequencyReport (MultiHTMLFiles).html
style=styles.barrettsblue;
title "Cars with Origin in %SCAN(&morigin_list,&i,~)";
proc freq data=sashelp.cars(where=(origin = "%SCAN(&morigin_list,&i,~)")
	tables type;
	format msrp dollar12.0;
run;
quit;
title;
ods html close;
%end;
%put &morigin_list;
%mend multfiles;

%multfiles;

Automating the Process of Creating Multiple Excel Files
Statistics show that the world's most used software application is Microsoft Excel®. Due to this dominance, SAS provides users with several ways to send results, tables, statistics, images and other output directly to an Excel spreadsheet. In the next example, redundant code and hardcoding issues are avoided by using PROC SQL to determine the number of unique (or distinct) values of the Origin column and, once known, are assigned to single-value and value-list macro variables. With the values assigned to the two macro variables, an iterative %DO statement is specified to control the propagation of Excel files containing one-way frequency results. The results of the three distinct Excel files that were created are also displayed, below.
ODS Excel Code:

```sas
%macro multExcelfiles ;
  proc sql noprint ;
    select count(distinct origin) /* derive number of origins */
      into :morigin_cnt from sashelp.cars
      order by origin ;
    select distinct origin /* derive unique origin values */
      into :morigin_list separated by "~" from sashelp.cars
      order by origin ;
  quit ;

  %do i=1 %to &morigin_cnt ;
    ods Excel file="e:\%SCAN(&morigin_list,&i,~)_FreqReport (MultiExcelFiles).xlsx"
      style=styles.barrettsblue ;
    title "Cars with Origin in %SCAN(&morigin_list,&i,~)" ;
    proc freq data=sashelp.cars(where=(origin = "%SCAN(&morigin_list,&i,~)")) ;
      tables type ;
      format msrp dollar12.0 ;
    run ;
  %end ;

  %put &morigin_list ;
%mend multExcelfiles ;

%multExcelfiles ;
```

ODS Excel Results:

Discovering the Number of Occurrences of a Value in a Data Set
Discovering the number of occurrences of individual values in a data set is useful information, particularly when constructing data-driven approaches. SAS provides several ways to count and determine the number of occurrences of a value in a data set.

Discovering the Number of Occurrences of a Value in a DATA Step
One approach for discovering the number of occurrences of a variable's value(s) is to construct a DATA step counting routine. In the next example, individual counters for the number of females and males are created, and after the last observation is read and processed, the results for each counter is output to the Counts data set, and the results displayed with PROC PRINT.
Modernizing Legacy SAS® Applications and Program Code, continued

**DATA Step Code:**

```sas
data Counts(drop=Sex);
  set sashelp.Heart(keep=Sex) end=EOF;
  if Sex = "Female" then Number_Females + 1;
  else if Sex = "Male" then Number_Males + 1;
  if EOF then do;
    Total = Number_Females + Number_Males;
    format Number_Females Number_Males Total comma7.;
    output;
  end;
run;
proc print data=Counts noobs;
run;
```

**DATA Step Results:**

<table>
<thead>
<tr>
<th>Number_Females</th>
<th>Number_Males</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>2,873</td>
<td>2,336</td>
<td>5,209</td>
</tr>
</tbody>
</table>

**Discovering the Number of Occurrences of a Value with the PROC FREQ NLEVELS Option**

Another approach for counting the number of occurrences of a variable’s value(s) is to specify the NLEVELS option in PROC FREQ. In this example, the variable SEX is kept and the NLEVELS option is specified for the SASHELP.Heart data set. The results show there are two levels for the variable, SEX, with 2,873 females and 2,336 males.

**PROC FREQ Code:**

```sas
proc freq data=sashelp.Heart(keep=sex) NLEVELS;
run;
```

**PROC FREQ Results:**

![The FREQ Procedure Table](image)

**Discovering the Number of Occurrences of a Value with PROC SQL**

Another approach for counting the number of occurrences of a variable’s value is to use the SUM function with an equality expression in PROC SQL. PROC SQL’s data access and query capabilities offer SAS users with a powerful approach to summing down rows and across columns. In this example, a SELECT query is specified with a SUM function for counting the number of “Females”, “Males” and their combined totals that are found in the SASHELP.HEART data set. An optional FORMAT=COMMA7. parameter is also specified to make the results easier to read.
Using Metadata to Determine the Number of Observations in a Data Set

Metadata is everywhere and is defined as information that describes data. Other definitions include information about data, or information about the design and specification of objects and data structures. In its most basic form, metadata is found in the cataloging systems of every academic library, public library, school library, and special library in the world. The typical book, magazine, microfiche, digital file, image, or object’s metadata is stored in cataloging systems. These cataloging systems are not composed of words, sentences, paragraphs, or chapters, but contain information about its author(s), title, subject, keyword(s), description, publisher, publication date, ISBN, format, resource identifier, copyright, and other information.

Older Methods of Determining the Number of Observations in a Data Set

Before the availability of metadata in the SAS System, users developed and included code routines that determined the number of observations in a data set. An often used DATA step approach, since the beginning of SAS-time, constructs a variable that counts the number of observations. Although syntactically correct, this approach does not “scale” well – due to the amount of I/O incurred and the sizes of data sets – when computing the counter. The next example, a DATA step approach computes the total number of “Sedans” found in the SASHELP.CARS data set, and displays the results using PROC PRINT.

**DATA Step Code:**

```sas
data sedans_counter(keep=type obsCtr)
   cars_sedans(drop=obsCtr);
set sashelp.cars(keep=origin type make MSRP) end=eof;
where upcase(type) = "SEDAN";
obsCtr + 1;
output cars_sedans;
if eof then output sedans_counter;
run;

proc print data=sedans_counter noobs;
run;
```

**DATA Step Results:**

<table>
<thead>
<tr>
<th>Type</th>
<th>obsCtr</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sedan</td>
<td>262</td>
</tr>
</tbody>
</table>
Using DICTIONARY.TABLES Metadata to Determine the Number of Observations in a Data Set

The SAS System collects and populates valuable metadata about SAS libraries, data sets (tables), catalogs, indexes, macros, system options, titles, views and other useful information in a collection of read-only tables called Dictionary tables. Dictionary tables serve a special purpose for SAS users by providing system-related information about the current SAS session’s SAS databases and applications. When a query processes a Dictionary table, SAS automatically launches a discovery process at runtime to collect information pertinent to that table. This information is made available any time after a SAS session is started.

When users need more information about SAS data sets the TABLES Dictionary table can be very helpful. The TABLES Dictionary table provides detailed information about the library names, the member (or data set) names, the date a data set was created and last modified, the number of observations in a data set, and much more. The next example illustrates a popular approach that accesses the metadata content from the DICTIONARY.TABLES table to determine the number of observations in any SAS data set.

**PROC SQL Code:**

```sql
title "Number of Rows in a Table" ;
proc sql ;
    select libname, memname, nobs format=comma10.
    from Dictionary.Tables
    where nobs NE . ;
quit ;
title ;
```

**PROC SQL Results:**

<table>
<thead>
<tr>
<th>Library Name</th>
<th>Member Name</th>
<th>Number of Physical Observations</th>
</tr>
</thead>
<tbody>
<tr>
<td>WORKS</td>
<td>COUNTS</td>
<td>1</td>
</tr>
<tr>
<td>MYDATA</td>
<td>ACTORS</td>
<td>13</td>
</tr>
<tr>
<td>MYDATA</td>
<td>ACTORS_WITH_MESSY_DATA</td>
<td>15</td>
</tr>
<tr>
<td>MYDATA</td>
<td>AE</td>
<td>13</td>
</tr>
<tr>
<td>MYDATA</td>
<td>DM</td>
<td>24</td>
</tr>
<tr>
<td>MYDATA</td>
<td>EX</td>
<td>145</td>
</tr>
<tr>
<td>MYDATA</td>
<td>MOVING</td>
<td>22</td>
</tr>
<tr>
<td>MYDATA</td>
<td>MOVING_WITH_MESSY_DATA</td>
<td>11</td>
</tr>
<tr>
<td>SASHELP</td>
<td>ACME</td>
<td>2,020</td>
</tr>
<tr>
<td>SASHELP</td>
<td>AARROW</td>
<td>198</td>
</tr>
<tr>
<td>SASHELP</td>
<td>AGMEC</td>
<td>428</td>
</tr>
<tr>
<td>SASHELP</td>
<td>AHMEC</td>
<td>1,000</td>
</tr>
<tr>
<td>SASHELP</td>
<td>AIR</td>
<td>144</td>
</tr>
<tr>
<td>SASHELP</td>
<td>APPLIANCE</td>
<td>168</td>
</tr>
<tr>
<td>SASHELP</td>
<td>ASSHORE</td>
<td>402</td>
</tr>
<tr>
<td>SASHELP</td>
<td>BASEBALL</td>
<td>728</td>
</tr>
<tr>
<td>SASHELP</td>
<td>BSE</td>
<td>24,205</td>
</tr>
<tr>
<td>SASHELP</td>
<td>DHMMN</td>
<td>3,264</td>
</tr>
<tr>
<td>SASHELP</td>
<td>DHT</td>
<td>137</td>
</tr>
<tr>
<td>SASHELP</td>
<td>DMVROG</td>
<td>24,591</td>
</tr>
<tr>
<td>SASHELP</td>
<td>DUN-flat</td>
<td>50,000</td>
</tr>
<tr>
<td>SASHELP</td>
<td>EPHS</td>
<td>428</td>
</tr>
<tr>
<td>SASHELP</td>
<td>CITATION</td>
<td>1,099</td>
</tr>
<tr>
<td>SASHELP</td>
<td>CITATIONN</td>
<td>145</td>
</tr>
<tr>
<td>SASHELP</td>
<td>CITATIONR</td>
<td>45</td>
</tr>
<tr>
<td>SASHELP</td>
<td>CITIVK</td>
<td>219</td>
</tr>
<tr>
<td>SASHELP</td>
<td>CTRTR</td>
<td>10</td>
</tr>
<tr>
<td>SASHELP</td>
<td>CLASS</td>
<td>15</td>
</tr>
<tr>
<td>SASHELP</td>
<td>CLASSP</td>
<td>15</td>
</tr>
</tbody>
</table>
Using SASHELP.VTABLE Metadata to Determine the Number of Observations in a Data Set

SAS also provides users with metadata content in a number of SASHELP views. In this next example the number of observations in any SAS data set can be determined by accessing the NOBS metadata content in the SASHELP.VTABLE view. This metadata content can be displayed using any output-producing SAS procedure, as shown below.

**PROC PRINT Code:**

```sas
title "Number of Rows in a Table" ;
proc print data=sashelp.vtable noobs ;
  var libname memname nobs ;
  format nobs comma10. ;
  where nobs NE . ;
run ;
title ;
```

**PROC PRINT Results:**

```
<table>
<thead>
<tr>
<th>libname</th>
<th>memname</th>
<th>nobs</th>
</tr>
</thead>
<tbody>
<tr>
<td>WORK</td>
<td>COUNTS</td>
<td>1</td>
</tr>
<tr>
<td>MYDATA</td>
<td>ACTORS</td>
<td>13</td>
</tr>
<tr>
<td>MYDATA</td>
<td>ACTORS_WITH_MESSY_DATA</td>
<td>15</td>
</tr>
<tr>
<td>MYDATA</td>
<td>AE</td>
<td>13</td>
</tr>
<tr>
<td>MYDATA</td>
<td>DM</td>
<td>24</td>
</tr>
<tr>
<td>MYDATA</td>
<td>EX</td>
<td>166</td>
</tr>
<tr>
<td>MYDATA</td>
<td>MOVES</td>
<td>23</td>
</tr>
<tr>
<td>SASHELP</td>
<td>AADOMP</td>
<td>2,020</td>
</tr>
<tr>
<td>SASHELP</td>
<td>AARFN</td>
<td>195</td>
</tr>
<tr>
<td>SASHELP</td>
<td>ADMSIG</td>
<td>429</td>
</tr>
<tr>
<td>SASHELP</td>
<td>ARMSG</td>
<td>1,000</td>
</tr>
<tr>
<td>SASHELP</td>
<td>AIR</td>
<td>161</td>
</tr>
<tr>
<td>SASHELP</td>
<td>APLUANC</td>
<td>156</td>
</tr>
<tr>
<td>SASHELP</td>
<td>ASSACMT</td>
<td>402</td>
</tr>
<tr>
<td>SASHELP</td>
<td>BASEBALL</td>
<td>322</td>
</tr>
<tr>
<td>SASHELP</td>
<td>BB</td>
<td>24,205</td>
</tr>
<tr>
<td>SASHELP</td>
<td>BWI</td>
<td>3,284</td>
</tr>
<tr>
<td>SASHELP</td>
<td>BMT</td>
<td>137</td>
</tr>
<tr>
<td>SASHELP</td>
<td>BJRRONS</td>
<td>21,921</td>
</tr>
<tr>
<td>SASHELP</td>
<td>BUY</td>
<td>11</td>
</tr>
<tr>
<td>SASHELP</td>
<td>DWIGHT</td>
<td>50,000</td>
</tr>
<tr>
<td>SASHELP</td>
<td>EARS</td>
<td>42</td>
</tr>
<tr>
<td>SASHELP</td>
<td>OFFDAY</td>
<td>1,069</td>
</tr>
<tr>
<td>SASHELP</td>
<td>GIMON</td>
<td>143</td>
</tr>
<tr>
<td>SASHELP</td>
<td>GTHDR</td>
<td>40</td>
</tr>
<tr>
<td>SASHELP</td>
<td>GTHMK</td>
<td>349</td>
</tr>
<tr>
<td>SASHELP</td>
<td>GTYR</td>
<td>18</td>
</tr>
<tr>
<td>SASHELP</td>
<td>CLASS</td>
<td>19</td>
</tr>
<tr>
<td>SASHELP</td>
<td>CLASPI</td>
<td>19</td>
</tr>
</tbody>
</table>
```

Using PROC PRINT with Style

Hecht (2011) describes the appearance of PROC PRINT output can be customized with colors, backgrounds, fonts, justifications, and other report components using styles. Styles can be specified for all destinations (e.g., RTF, PDF, HTML, Excel, etc.) except the Listing destination. In the next example, the SASHELP.CARS data set is sorted in ascending order by the variables Origin, Type, Make and MSRP; the HTML destination is opened with the HTMLBlue style selected for output; and background and foreground styles selected for the data, obs and total parts of the PROC PRINT report output.
Modernizing Legacy SAS® Applications and Program Code, continued

**PROC PRINT Code:**

```sas
proc sort data=sashelp.Cars(keep=Origin Type Make MSRP)
  out=work.Cars_Sorted;
  where MSRP < 20000;
  by Origin Type Make MSRP;
run;
ods HTML path="/folders/myfolders" (url=none)
  file="PROC-PRINT-with-Style.html"
  style=HTMLBlue;
proc print data=work.Cars_Sorted
  style (data) = [background=Blue foreground=white]
  style (obs) = [background=red foreground=white]
  style (total) = [background=yellow foreground=black];
  by Origin Type;
  id Origin Type Make;
  format msrp dollar12.0;
  sum MSRP;
run;
ods HTML close;
```

**PROC PRINT Results:**

<table>
<thead>
<tr>
<th>Origin</th>
<th>Type</th>
<th>Make</th>
<th>MSRP</th>
</tr>
</thead>
<tbody>
<tr>
<td>USA</td>
<td>Sports</td>
<td>Ford</td>
<td>$18,345</td>
</tr>
<tr>
<td>USA</td>
<td>Truck</td>
<td>Chevrolet</td>
<td>$18,700</td>
</tr>
<tr>
<td>USA</td>
<td>Truck</td>
<td>Dodge</td>
<td>$17,930</td>
</tr>
<tr>
<td>USA</td>
<td>Truck</td>
<td>Ford</td>
<td>$14,385</td>
</tr>
<tr>
<td>USA</td>
<td>Truck</td>
<td>GMC</td>
<td>$16,330</td>
</tr>
<tr>
<td>USA</td>
<td>Truck</td>
<td>$67,305</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Origin</th>
<th>Type</th>
<th>Make</th>
<th>MSRP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Asia</td>
<td>Hybrid</td>
<td>Honda</td>
<td>$19,110</td>
</tr>
<tr>
<td>Asia</td>
<td>SUV</td>
<td>Honda</td>
<td>$18,060</td>
</tr>
<tr>
<td>Asia</td>
<td>SUV</td>
<td>Kia</td>
<td>$16,635</td>
</tr>
<tr>
<td>Asia</td>
<td>SUV</td>
<td>Mitsubishi</td>
<td>$16,582</td>
</tr>
<tr>
<td>Asia</td>
<td>SUV</td>
<td>Suzuki</td>
<td>$12,165</td>
</tr>
<tr>
<td>USA</td>
<td>Sports</td>
<td>Hyundai</td>
<td>$18,730</td>
</tr>
<tr>
<td>Asia</td>
<td>Truck</td>
<td>Mazda</td>
<td>$14,040</td>
</tr>
<tr>
<td>Asia</td>
<td>Truck</td>
<td>Nissan</td>
<td>$19,479</td>
</tr>
<tr>
<td>Asia</td>
<td>Truck</td>
<td>Toyota</td>
<td>$12,000</td>
</tr>
<tr>
<td>Asia</td>
<td>Truck</td>
<td>Toyota</td>
<td>$16,420</td>
</tr>
<tr>
<td>Asia</td>
<td>Truck</td>
<td>$66,614</td>
<td></td>
</tr>
<tr>
<td>Asia</td>
<td>Wagon</td>
<td>Kia</td>
<td>$11,005</td>
</tr>
<tr>
<td>Asia</td>
<td>Wagon</td>
<td>Mitsubishi</td>
<td>$17,405</td>
</tr>
<tr>
<td>Asia</td>
<td>Wagon</td>
<td>Sohn</td>
<td>$14,105</td>
</tr>
<tr>
<td>Asia</td>
<td>Wagon</td>
<td>Suzuki</td>
<td>$10,497</td>
</tr>
<tr>
<td>Asia</td>
<td>Wagon</td>
<td>Toyota</td>
<td>$10,685</td>
</tr>
<tr>
<td>Asia</td>
<td>Wagon</td>
<td>$70,767</td>
<td></td>
</tr>
<tr>
<td>Asia</td>
<td></td>
<td>$974,429</td>
<td></td>
</tr>
</tbody>
</table>
Using Available Memory with Hash Object Programming

Dorfman (2009) describes a SAS hash object as, “a high-performance look-up table residing completely in the DATA step memory.” Due to the costs and availability of memory resources in today’s computing environments, software vendors are doing everything they can to develop language constructs that capitalize on memory-resident operations. Dorfman further describes that, “The hash object is implemented via a Data Step Component Interface (DSCI), meaning that it is not a part of the DATA step proper. Rather, picture it as a black-box device you can manipulate from inside the DATA step to ask it for lightning-quick data storage and retrieval services.”

Lafler (2016) describes a SAS hash object as, “a data structure that contains an array of items that are used to map identifying values, known as keys (e.g., employee IDs), to their associated values (e.g., employee names or employee addresses). As implemented, a hash object in the SAS System is used as a DATA step construct and is not available to any SAS Procedures.” A hash object reads the contents of a data set into memory once allowing the SAS system to repeatedly access the data, as necessary. The contents of a hash object can be saved to a SAS data set (or table), but at the end of the DATA step the hash object and all its contents disappear. Since memory-based operations are typically faster than their disk-based counterparts, users often experience faster and more efficient table lookup, merge, sort and transpose operations.

Users with DATA step programming experience will find the hash object syntax relatively straightforward to learn and use. Available in all operating systems running SAS 9 or greater, the hash object is called using methods. The syntax for calling a method involves specifying the name of the user-assigned hash table, a dot (.), the desired method (e.g., operation) by name, and finally the specification for the method enclosed in parentheses. The following example illustrates the basic syntax for calling a method to define a key.

```
MatchTitles.DefineKey ('Title');
```

where MatchTitles is the name of the hash table, DefineKey is the name of the called method, and ‘Title’ is the specification being passed to the method.

In the next example, an essential operation frequently performed by users is the process of table lookup or search. The hash object as implemented in the DATA step provides users with the necessary tools to conduct match-merges (or joins) of two or more data sets. Data does not have to be sorted or be in a designated sort order before use as it does with the DATA step merge process. The following code illustrates a hash object with a simple key (TITLE) to merge (or join) the MOVIES and ACTORS data sets to create a new data set (MATCH_ON_MOVIE_TITLES) with matched observations. The same operation is handled in a DS2 section example here.

```
data match_on_movie_titles(drop=rc) ;
  if 0 then set mydata.movies mydata.actors ; /* load variable properties into hash tables */
  if _n_ = 1 then do ;
    declare Hash MatchTitles (dataset:'mydata.actors') ; /* declare the name MatchTitles for hash */
    MatchTitles.DefineKey ('Title') ; /* identify variable to use as key */
    MatchTitles DEFINE Data ( 'Actor_Leading',
      'Actor_Supporting' ) ; /* identify columns of data */
    MatchTitles DEFINE Done () ; /* complete hash table definition */
  end ;
  set mydata.movies ;
  if MatchTitles.find(key:title) = 0 then output ; /* lookup TITLE in MOVIES table using MatchTitles hash */
run ;
```
User-developed Macros

There are 2 kinds of user-developed macros, single use “through away” code and special purpose, multi-use macros. The latter should be stored in a central repository (or directory) for a particular project, department or an entire organization. These macros should be completely documented from top to bottom. Including an explanation of the macros purpose, its input parameters and expected output along with a sample test run, as shown in the example below. (* A FCMP Subroutine that accomplishes the same character stripping but for a dataset variable is found here *) A contents readme file documenting the macros in the collection should be kept in the directory with the macros so users can see what’s available. For more information about the SAS macro language, see Carpenter's Complete Guide to the SAS® Macro Language, Third Edition.

MSTRIPCHR Macro:

/* ***************************************************************
* MACRO: MSTRIPCHR removes a specified character recursively from the beginning or
* end of a given macro character string. %, &, ' and " are not allowed for
* any resolved parameter value and will cause an ERROR!!!
* *
* By: Clark Roberts, Decision Analytics
* *
* Updates: 2003 Program written.
* 2017 Added _LOC parameter and logic to
* strip _CHR from the end of _STR
*
* Copyright Decision Analytics 2003 - 2017
* All rights reserved
* *************************************************************** */

%macro mstripchr(_str , /* <REQUIRED> */
                _chr , /* <REQUIRED> */
                _n   , /* <REQUIRED> */
                _loc = S)
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
Modernizing Legacy SAS® Applications and Program Code, continued

were removed. It must be initialized to 0 (zero) before calling STRIPCHR.

* LOC Location of the character (_CHR) to be stripped from (_STR). S -> beginning (default), E -> end.

*********************************************************************************/

%let _loc = %upcase(%substr(&_loc,1,1));
%if &_loc ^= E and &_loc ^= S %then %let _loc = S;
%if &_loc = E %then %let &_str = %qsysfunc(reverse(%nrbquote(&&_str)));
%let &_chr = %qsubstr(%nrbquote(&&_chr),1,1);
%if %length(%nrbquote(&&_str)) > 1 %then %do;
  %if %qsubstr(%qtrim(%qleft(%nrbquote(&&_str))),1,1) = %nrbquote(&&_chr)
  %then %do;
    %let &_n = %eval(&&_n + 1);
    %let &_str = %qsubstr(%nrbquote(&&_str),2);
    %mstripchr(&_str, &_chr, &_n, _loc=S);
  %end;
%end;
%if &_loc = E %then %let &_str = %qsysfunc(reverse(&&_str));
%mend mstripchr;

MSTRIPCHR Macro Log Results:

* options symbolgen macrogen mlogicnest source source2;
%macro testmstripchr();
%local var chr loc n ;
%let n = 0;
%let var = --------------------------------------testval2;
%let chr = -;
%let loc = s;
%put before mstripchr called &n &var &chr &loc;
%mstripchr(var, chr, n, _loc=&loc);
%put after mstripchr called &n &var &chr &loc;
%let var = testval3:-------------------------------;
%let chr = :;
%let loc = e;
%let n = 0;
%put before mstripchr called &n &var &chr &loc;
%mstripchr(var, chr, n, _loc=&loc);
%put after mstripchr called &n &var &chr &loc;
%mend testmstripchr;
%testmstripchr();

before mstripchr called 0 -----------------------------testval2 - s
after mstripchr called 38 testval2 - s
before mstripchr called 0 testval3:------------------------------- : e
after mstripchr called 27 testval3 : e

PC Windows Environment
SAS provides the MODULExy family of call routines and functions (including MODULE, MODULEN, MODULEC, MODULEI, MODULEIN, and MODULEIC) to access system related API functions and other DLL function calls. It’s a very powerful tool, but somewhat obscure so it’s not widely used currently.

The use of these SAS MODULExy functions requires creation of a text file (SASCBTBL) that describes the API/DLL routine to be invoked. This file can be permanently created or created in-line with FILE and PUT statements, as the following example demonstrates. Download the Win32 API Declarations for Visual Basic reference or for a version updated for 64 bit office go to this Microsoft Office site. Scroll down and, under the “Links” section, click the link associated with “Office 2010 Help Files: Win32API_PtrSafe with 64-bit Support” to automatically download the Win32API_PtrSafe with 64-bit Support. Then, run the downloaded EXE file to install the TXT and related files. These references will give you the information you need to help build the SAS SASCBTBL file. Many of the 32 bit Win32API functions will work in a 64 bit environment without modification (refer to
Langston 2015 for information on MODULExy 64 bit usage). It seems that pointers are the main issue and the above reference to the Win32API_PtrSafe functions should hopefully resolve any of those issues.

In the following example, the OPENWORD macro opens Microsoft Word for a DDE session. It uses the Win32API calls “FindExecutableA” to find the location of Word for Windows (WinWord) and “WinExec” to execute WinWord. The authors are using the SAS University Edition which runs in a Linux virtual machine, but this macro needs direct access to Windows so an example of its use was not possible. There are a number of other good examples in the articles in the WIN32API References and Suggested Reading section.

MODULExy functions are available in other operating systems (eg. UNIX, VMS, etc.). Refer to the SAS Companion for these operating systems for details.

MODULExy Function Example Code:

| Source: OPENWORD.SAS (MS Word 95/97 under Win95/98 SAS 6.12 Product Version) |
| Type: Include Module |
| Function: Opens MS Word for a SAS to Word DDE session |
| Note: This version has been tested in MS Word 95 and 97 under Windows 95 and 98 in SAS 6.12 (TS025 and above) |
| Parameters: None |
| By: Clark Roberts, Decision Analytics |
| Updates: 5-30-97 Program written. 10-7-99 Added logic to build the SAS modulen control table in the OPENWORD macro. |
| Copyright Decision Analytics 1997 - 2017. All rights reserved |

%macro openword();
%
*** Define a global macro variable for the return code to be tested by the calling program. If the operation is successful then the return code will be > 0, If it fails then the return code will be 0.
%
%global rcword;
%let rcword = 1;
%
*** Define the parameter settings for the Win API calls that will be used by the SAS modulen function for this program. These can be stored permanently in a text file for better performance, if required. Also create an empty .DOC file for the FindExecutableA API call to use to find the path to the MS Word executable WINWORD.EXE.
***
*** Win32API Visual Basic Declarations
***
*** Declare Function WinExec Lib "kernel32" Alias "WinExec"
*** (ByVal lpCmdLine As String, ByVal nCmdShow As Long)
*** As Long
***
*** Declare Function FindExecutable Lib "shell32.dll" Alias "FindExecutableA"(ByVal lpFile As String,
*** ByVal lpDirectory As String, ByVal lpResult As String)
***     As Long
***
****************************************************************************************;

filename modtabl 'c:\modtabl.txt';

data _null_;  
    file modtabl;
        put @2 'routine WinExec';
        put @5 'maxarg=2';
        put @5 'minarg=2';
        put @5 'stackpop=called';
        put @5 'module=KERNEL32';
        put @5 'returns=ushort;';
        put @7 'arg 1 input char format=$cstr200.;';
        put @7 'arg 2 input num byvalue format=pib4.;' //;
    put @2 'routine FindExecutableA';
        put @5 'maxarg=3';
        put @5 'minarg=3';
        put @5 'stackpop=called';
        put @5 'module=shell32';
        put @5 'returns=ushort;';
        put @7 'arg 1 input char format=$cstr200.;';
        put @7 'arg 2 input char format=$cstr200.;';
        put @7 'arg 2 output char format=$cstr200.;';
        run;

filename modtabl clear;

filename wordtemp 'c:\system.doc';

data _null_;  
    file wordtemp;
        put 'this is a dummy .doc file for the FindExecutableA API';
        put 'to use to locate the MS Word .EXE file. It is created';
        put 'by the SAS OPENWORD macro';
    run;

filename wordtemp clear;

*************************************************************************

*** If MS Word is already open, then close it and allow the user to save
*** their work.
*************************************************************************

filename wordchk dde 'winword|system!selected' command;

%let status = %sysfunc(fopen(wordchk,S));

%if &status %then %do;
    data _null_;  
        file wordchk;
            put '[FileExit 1]';
        run;
    %let rc = %sysfunc(fclose(&status));
    filename wordchk clear;
%end;

*************************************************************************

*** Start a MS Word session if requested. SAS calls to the Windows APU are
*** used to find the location of MS Word and if found, starts a session.
*** The modtabl.txt file, which contains the parameters for each system call
*** and the system.doc file, which is used as a trigger for the
*** FindExecutableA API function to locate MS Word on the system. The SAS
*** MODULEN function is used for the API interface. If MS Word fails to open, *** then set the RCWORD return code to 0. The Word session is started in *** minimized state.

**************

filename saschtbl "c:\modtabl.txt";

data _null_;
  length sysdir wordpath $200;
  wordpath = '';
  sysdir = 'c:\';
  rc1 = modulen('*ei','FindExecutableA','system.doc',sysdir,wordpath);
  if rc1 > 32 then do;
    rc2 = modulen('*ei','WinExec',wordpath,2);
    if rc2 <= 0 then call symput('rcword','0');
  end;
  else do;
    put 'OPENWORD ERROR: Could not find the MS Word executable';
    call symput('rcword','0');
  end;
  put rc1= rc2= wordpath=;
run;

filename saschtbl clear;

filename wordsys dde 'Winword|System';

%mend openword;

/*********************************************************
***                  E N D   O F   M O D U L E
*** OPENWORD.SAS
***********************************************************/

** User-developed Functions with the FCMP and PROTO Procedures **

In version 9.2, SAS added the ability to create user-written functions and call routines, utilizing the SAS DATA step and DS2 programming languages, using the PROC FCMP and its cousin, PROC PROTO. PROC PROTO is used to incorporate C language functions into SAS, while PROC FCMP compiles SAS DATA Step and DS2 language elements for use in those environments. “FCMP routines are stored in a data set and can be called from several SAS/STAT®, SAS/ETS®, and SAS/OR® software procedures, like the NLIN, MODEL, and NLP procedures. In SAS 9.2, FCMP routines can be called from a DATA step.” [Secowski 2007].

User-written routines are defined using PROC FCMP and can be either a function or a CALL routine. A function is a subroutine that returns a value and accepts either zero or more arguments passed by value and will not modify the value in the calling routine. On the other hand a CALL routine is a subroutine that does not return a value explicitly and normally receives one or more arguments passed by value, or also by reference. When passed by reference one copy of the data is shared between the calling routine and the subroutine. This is how information is returned to the calling program. The next example illustrates a user-defined function and its related subroutine that converts yards to meters. The function is contained with a larger collection of functions stored in the ExampleFunctions library. To view the use of these functions in a DS2 package refer here.

libname cr "folders/myfolders/CR" ; /* permanently store functions in*/
/* ExampleFunctions library for */
/* reuse */

/****************************
Functions and Subroutines to convert various units of measurement
****************************/

options cmplib = cr.ExampleFunctions ;
Modernizing Legacy SAS® Application and Program Code, continued

```sas
proc fcmp outlib=cr.ExampleFunctions.Yards2Meters;
function Yards2Meters(inUnits);
  return(inUnits / 0.9144);
endsub;

proc fcmp outlib=cr.ExampleFunctions.Yards2Meters_sub;
subroutine Yards2Meters_sub(inUnits, outUnits);
  outargs outUnits;
  outUnits = Yards2Meters(inUnits);
  return;
endsub;

proc fcmp outlib=cr.ExampleFunctions.Inches2Centimeters;
function Inches2Centimeters(inUnits);
  return(inUnits * 2.54);
endsub;

proc fcmp outlib=cr.ExampleFunctions.Centimeters2Inches;
function Centimeters2Inches(inUnits);
  return(inUnits / 2.54);
endsub;

proc fcmp outlib=cr.ExampleFunctions.Kilograms2Pounds;
function Kilograms2Pounds(inUnits);
  return(inUnits * 2.2046226218);
endsub;
quit;

/************************************************************
  Test program for yards to meters function and subroutine
  ************************************************************/
data _null_;  
  length invar outvar 8;
  invar = 5;
  outvar = Yards2Meters(invar);
  put "function call " outvar=;
  invar = 5;
  call Yards2Meters_sub(invar, outvar);
  put "subroutine call " outvar=;
run;

FCMP User-defined Function and Subroutine Log Results:

84  /* Test program for yards to meters function and subroutine */
85  data _null_;  
86  length invar outvar 8;
87  invar = 5;
88  outvar = Yards2Meters(invar);
89  put "function call " outvar=;
90  invar = 5;
91  call Yards2Meters_sub(invar, outvar);
92  put "subroutine call " outvar=;
93  run;

function call outvar=109.36132983
subroutine call outvar=109.36132983
```

PROC FCMP routines can be recursive. Sekowski (2007) describes “Recursion as a problem-solving technique that reduces a problem to a smaller one that is simpler to solve and then combines the results of the simpler solution(s) to form a complete solution. A recursive function is a function that calls itself, either directly or indirectly.” The next example demonstrates the concept of recursion by using the character stripping paradigm, only this time a PROC FCMP subroutine is used to modify a DATA step variable instead of a macro variable (refer to the earlier macro example in the User Developed Macros section). The next example illustrates a user-defined FCMP routine and the result from the test program is illustrated, below.

**Recursive User-defined FCMP Routine:**

```sas
/******************************************************************
FCMP subroutine to recursively strip character _CHR from the
Beginning or end of a character string _STR
***************************************************************************/
options cmplib = work.ExampleFunctions;
proc fcmp outlib=work.ExampleFunctions.stripchr;
subroutine stripchr(_str $, _chr $, _n , _loc $, origloc $);
outargs _str, _chr, _n, _loc, origloc;
_loc = upcase(substr(_loc,1,1));
if _loc ^= 'E' and _loc ^= 'S' then do;
  _loc = 'S';
  origloc = _loc;
end;
if _loc = 'E' then do;
  _str = trim(left(reverse(_str)));
  _loc = "S";
end;
_chr = substr(_chr,1,1);
if length(_str) > 0 then do;
  if substr(trim(left(_str)),1,1) = _chr then do;
    _n = _n + 1;
    _str = substr(_str,2);
    call stripchr(_str, _chr, _n, _loc, origloc);
    return;
  end;
  if upcase(origloc) ^= 'S' then do;
    _str = reverse(_str);
    _loc = "E";
  end;
end;
return;
endsub;
quit;
```

**Test Program for the STRIPCHR FCMP Function:**

```sas
/******************************************************************
* Test program for stripchr FCMP subroutine
***************************************************************************/
data before after ;
length _str $200 _chr $5 _loc $5 _n 4 ;
_chr = 'x' ;
_loc = 'x' ;
```
Modernizing Legacy SAS® Applications and Program Code, continued

Recursive User-defined FCMP Routine Results:

Values before function call

<table>
<thead>
<tr>
<th>Obs</th>
<th>_str</th>
<th>_chr</th>
<th>_loc</th>
<th>origloc</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>testval1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>testval2&gt;&gt;..................</td>
<td>&gt;</td>
<td>e</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>testval4?........................</td>
<td>?</td>
<td>e</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>..................................................</td>
<td>.</td>
<td>s</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Obs</th>
<th>_str</th>
<th>_chr</th>
<th>_loc</th>
<th>origloc</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>testval1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>testval2 19 E</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>testval3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>testval4 27 E</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Values after function call
PROC FCMP provides the special function named RUN_MACRO that enables a user-written routine to run a macro. There is no restriction as to what the macro can generate in terms of SAS code: it can include a number of DATA steps and PROC steps. Since a user-written routine is callable from within a DATA step then RUN_MACRO makes it’s possible to have nested DATA steps. References by Carpenter (2016) and Deguire (2013) provide more detail on the use of RUN_MACRO.

Beginning with SAS® 9.3, hashing became available to user-defined subroutines through the FCMP Procedure. “Hashing is an important construct often used to improve performance of operations such as merging, filtering and searching. Hashing has been supported in the DATA step for more than a decade... Subroutines have always encapsulated and modularized code, making programs reusable, and the addition of hashing allows users to extend the scope of their programs, tackling larger problems without sacrificing simplicity.” (Henrick 2017). For more information about the DATA step hash construct, see Lafler (2016) and Dorfman / Henderson (2017).

PROC DS2

Introduced in SAS Version 9.4 in 2013, DS2 is a data step like programming language redesigned from the ground up with the following goals in mind:

- Process ANSI SQL natively for integration with external data sources.
- Provide syntax for multi-threaded processing for accelerated execution of CPU intensive tasks.
- Provide just enough Object Oriented Programming (OOP) constructs, including method overloading, for reusability of DS2 code. Method overloading consists of having 2 or more methods with the same name, but with different signatures. Signatures consist of the methods name and ordered list of parameter data classes.

The differences between the data step and DS2 are:

- DS2 supports ANSI native numeric data types and multi-byte character types to directly access DBMS data.
- DS2 is multi-threaded whereas the data step is single threaded providing acceleration for CPU bound processing by DS2. If the process is I/O bound the multi-threading in DS2 will not help reduce run times.
- DS2 programs can take advantage of the SAS In-Database Code Accelerator by allowing DS2 programs to execute on the Massively Parallel Processing (MPP) capabilities on the DBMS hardware with the SAS Embedded Process.
- There are almost no reserved words in the data step language. In DS2 all keywords are reserved word
- Variables referenced in a data step are global in scope, Variables referenced in a DS2 program can be either global or local in scope.
- All variables referenced in the data step are in the Program Data Vector, while DS2 variables of local scope are not in the PDV.
- Creating reusable data step code with variable encapsulation requires PROC FCMP. Reusable code in DS2 with variable encapsulation is possible using PROC DS2 syntax in a ‘package’ program.
- The data step can consume a table produced by an SQL query as input to a set statement. While DS2 can directly accept the result set of an SQL query as input to the set statement.

Some of the prerequisites for using DS2 include:

- DATA step programming experience
- Accessing data with a LIBNAME statement
- The role of the program data vector (PDV) in DATA step processing
- Conditional processing with IF-THEN-ELSE, SELECT-WHEN, etc. statements
- DO loop processing
- Array processing
- SAS Macro language
- SQL
Within a DS2 block you can define and execute three types of program blocks:

<table>
<thead>
<tr>
<th>Program Block</th>
<th>Brief Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data</td>
<td>The heart of the DS2 language, data programs manipulate input datasets to produce output result sets. They can receive input from tables, SQL result sets or thread program result sets.</td>
</tr>
<tr>
<td>Package</td>
<td>A package programming block or package refers to the stored library of variables and methods bounded by PACKAGE...ENDPACKAGE statements. The variables and methods of a package can be used by DS2 programs, threads, or other packages, enabling an object oriented approach to development.</td>
</tr>
<tr>
<td>Thread</td>
<td>Thread programs manipulate input datasets to produce output result sets that are returned to a data program. Used to simultaneously process several rows of data in parallel threads. A thread programming block, or thread program, refers to a stored program that is bounded by the THREAD...ENDTHREAD statements. The thread program can be called by the SET FROM statement in a DS2 program or package.</td>
</tr>
</tbody>
</table>

**Methods**

A method programming block or method block refers to a sub-block of programming statements that are bounded by the in methods. METHOD and END statements. User methods need to be defined before the 3 internal built-in methods INIT, RUN and TERM. The INIT method runs first where any initializations takes place. The RUN method executes next and finally the TERM method runs last. If the user doesn’t define these then DS2 automatically creates them. User methods can be called from any of the internal built-packages.

**Packaging**

A package is a collection of methods and variables that can be used in DS2 programs. “... packages are analogous to classes in other object-oriented languages such as C# or Ruby and can be used to massively improve your programming effectiveness.” (Brooks 2016) Packages should be stored in a data set within a SAS library for re-use. The first example below provides an example of defining and storing a package. To use a package; a DS2 program, another package, or a thread instantiates the package to access its methods.

“With the addition of user-defined packages to the SAS DS2 programmer’s toolbox SAS have opened up a new way of designing and programming our SAS applications. If we, as SAS developers, use this facility in a true object-oriented way we can significantly improve code re-use, encourage standardization of techniques and remove much of the routine coding which is part and parcel of the traditional procedural programming paradigm.” (Brooks 2016)

**Built-in Packages** (refer to Jordan 2016 for detailed coverage of built-in packages):

**SQLSTMT Package**

Provides a way to pass FedSQL statements to a DBMS for execution and to access the result set returned by the DBMS. There are performance benefits of using DS2 over PROC SQL. “The SQLSTMT package and the SQLEXEC function enable DS2 programs to dynamically generate, prepare, and execute FedSQL statements to update, insert, or delete rows from a table. With an instance of the SQLSTMT package or the SQLEXEC function, the FedSQL statement allocate, prepare, execute, and free occurs at run time.” (Kaufmann 2014)

**MATRIX Package**

The MATRIX package provides you with a powerful and flexible matrix programming capability.

**HTTP and JSON Packages**

The HTTP package provides the vehicle to construct an HTTP client to retrieve and post data on the internet. You can also retrieve status codes to log traffic between the client and server using the SAS Logging facility.

The JSON (JavaScript Object Notation) package gives you a means for creating and parsing JSON text.

**HASH AND HASH ITERATOR Packages**

These packages provide data hashing capability to DS2 routines like their data step equivalents.
FCMP Package
The Function Compiler procedure (FCMP) supports calls to FCMP functions and subroutines from within the DS2 language.

TIMEZONE (TZ) Package
The TZ package allows DS2 to process ANSI date, time and timestamp values from different time zones more easily.

LOGGER Package
Provides a basic interface (open, write, and level query) to the SAS logging facility.

The first example uses the SET statement of the SQLSTMT package. This is the same example as the DATA step version demonstrated earlier. In the case of the SET statement, no package instantiation is required. The SASHELP cars dataset is subset on the variable TYPE for records containing SUV, Truck or Wagon. For some reason SAS didn’t like the SASHELP libname in the DS2 block, so we created a copy in WORK first.

DS2 SQLSTMT Package Example:

```sas
data cars;
set sashelp.cars;
run;

proc ds2;
Data work.vehicles / overwrite=yes;
method run();
   set {select origin, type, msrp from work.cars
      where upcase(type) in ('SUV', 'TRUCK', 'WAGON')
      order by msrp};
end;
enddata;
run;
quit;

proc print data=vehicles(obs=17);
    format msrp dollar8.;
run;
```

Log Results from Package Example:

```
data cars;
set sashelp.cars;
run;

NOTE: There were 428 observations read from the data set SASHELP.CARS.
NOTE: The data set WORK.CARS has 428 observations and 15 variables.
NOTE: DATA statement used (Total process time):
   real time           0.00 seconds
   cpu time            0.00 seconds
```

```sas
proc ds2;
Data work.vehicles / overwrite=yes;
method run();
   set {select origin, type, msrp from work.cars
      where upcase(type) in ('SUV', 'TRUCK', 'WAGON')
      order by msrp};
```
The second example is an example of the use of the HASH package that duplicates the datastep merge example in DATA step hash example merging leading actors and supporting actors with their corresponding movie titles to provide additional information. The key, data and dataset name, etc. is entered with the package instantiation statement instead of separate statements for each of these to reduce the lines of code. The additional fields in the instantiation statement are hashexp (1), ordered (a for ascending), action for duplicate key entries (error) and multidata (no).

**DS2 Hash Package Example:**

```
libname mydata '/folders/myfolders/CR';
proc ds2;
  data match_on_movie_titles / overwrite=yes;
    dcl varchar(20) ActorLeading ActorSupporting;
    dcl varchar(30) Title;
    /* Define Hash Table MatchTitles */
    dcl package hash MatchTitles ([Title], [ActorLeading ActorSupporting], 1, 'mydata.actors', 'a', 'error', '', 'no');
    Method run();
    set mydata.movies;
    /* lookup TITLE in MOVIES table using MatchTitles hash */
```
if MatchTitles.find() = 0;
End;
enddata;
run;
quit;
proc print data=match_on_movie_titles;
var Title Length Category Year Studio Rating ActorLeading ActorSupporting;
run;

Output from DS2 Hash Package Example:

<table>
<thead>
<tr>
<th>Ob No</th>
<th>Title</th>
<th>Length</th>
<th>Category</th>
<th>Year</th>
<th>Studio</th>
<th>Rating</th>
<th>ActorLeading</th>
<th>ActorSupporting</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Brave Heart</td>
<td>177</td>
<td>Action Adventure</td>
<td>1996</td>
<td>Paramount Pictures</td>
<td>R</td>
<td>Mel Gibson</td>
<td>Sophie Marasou</td>
</tr>
<tr>
<td>2</td>
<td>Christmas Vacation</td>
<td>97</td>
<td>Comedy</td>
<td>1999</td>
<td>Warner Brothers</td>
<td>PG-13</td>
<td>Chevy Chase</td>
<td>Beverly D'Angelo</td>
</tr>
<tr>
<td>3</td>
<td>Coming to America</td>
<td>116</td>
<td>Comedy</td>
<td>1998</td>
<td>Paramount Pictures</td>
<td>R</td>
<td>Eddie Murphy</td>
<td>Antonio Banderas</td>
</tr>
<tr>
<td>4</td>
<td>Forrest Gump</td>
<td>142</td>
<td>Drama</td>
<td>1994</td>
<td>Paramount Pictures</td>
<td>PG-13</td>
<td>Tom Hanks</td>
<td>Sally Field</td>
</tr>
<tr>
<td>5</td>
<td>Ghost</td>
<td>127</td>
<td>Drama Romance</td>
<td>1996</td>
<td>Paramount Pictures</td>
<td>PG-13</td>
<td>Patrick Swayze</td>
<td>Demi Moore</td>
</tr>
<tr>
<td>6</td>
<td>Lethal Weapon</td>
<td>110</td>
<td>Action Cops &amp; Robber</td>
<td>1987</td>
<td>Warner Brothers</td>
<td>R</td>
<td>Mel Gibson</td>
<td>Danny Glover</td>
</tr>
<tr>
<td>7</td>
<td>Michael</td>
<td>105</td>
<td>Drama</td>
<td>1997</td>
<td>Warner Brothers</td>
<td>PG-13</td>
<td>John Travolta</td>
<td>Andie MacDowell</td>
</tr>
<tr>
<td>8</td>
<td>National Lampoons Vacation</td>
<td>90</td>
<td>Comedy</td>
<td>1993</td>
<td>Warner Brothers</td>
<td>PG-13</td>
<td>Chevy Chase</td>
<td>Beverly D'Angelo</td>
</tr>
<tr>
<td>9</td>
<td>Rocky</td>
<td>120</td>
<td>Action Adventure</td>
<td>1976</td>
<td>MGM / UA</td>
<td>PG</td>
<td>Sylvester Stallone</td>
<td>Barbie Hawn</td>
</tr>
<tr>
<td>10</td>
<td>Silence of the Lambs</td>
<td>118</td>
<td>Drama Suspense</td>
<td>1991</td>
<td>Orion</td>
<td>R</td>
<td>Anthony Hopkins</td>
<td>Jodie Foster</td>
</tr>
<tr>
<td>11</td>
<td>The Hunt for Red October</td>
<td>135</td>
<td>Action Adventure</td>
<td>1989</td>
<td>Paramount Pictures</td>
<td>PG</td>
<td>Sean Connery</td>
<td>Alec Baldwin</td>
</tr>
<tr>
<td>13</td>
<td>Titanic</td>
<td>194</td>
<td>Drama Romance</td>
<td>1997</td>
<td>Paramount Pictures</td>
<td>PG-13</td>
<td>Leonardo DiCaprio</td>
<td>Kate Winslet</td>
</tr>
</tbody>
</table>

User Defined Packages

- These are packages that you can use to store methods for any purpose. A user can store methods that they create in user-defined packages.
- These packages can be thought of as libraries of your methods. Any type of method can be saved in a package. Once you have stored methods in a package (using the PACKAGE statement), you can access them by creating an instance of the package with only a DECLARE statement or with the _NEW_ operator.

The following example is taken from the first FCMP example, the stored functions in ExampleFunctions in the CR library. The wrapper routine creates a package containing the stored functions for use in DS2. The test program instantiates the package as ConvertIt and calls the Kilograms2Pounds function.

By storing the “function library” as a package the routines can then be used in both a DATA step program and data program in DS2. A little confusing at first, but eventually it’ll become clearer.

### DS2 FCMP Wrapper Program for Permanent Function Storage:

```sas
/* Wrap the FCMP functions in a DS2 package */
libname cr "/folders/myfolders/CR" ;
proc ds2;
    package cr.fcmp_converters / overwrite=yes language='fcmp'
        table='cr.ExampleFunctions' ;
run;
quit:
```
Modernizing Legacy SAS® Applications and Program Code, continued

Test Program for DS2 FCMP Function Call:

```sas
/s*************************************************************/
| Test program for unit conversion FCMP DS2 package          |
| *************************************************************|
| proc ds2;                                                  |
|  data _null_;                                              |
|    dcl package cr.fcmp_converters convertIt();            |
|    dcl double invar having format 12.6;                   |
|    dcl double outvar having format 12.6;                  |
|    method run();                                          |
|      invar=13.6;                                          |
|      outvar = convertIt.Kilograms2Pounds(invar);          |
|      put 'Kilograms2Pounds ' invar= outvar= ;             |
|    end;                                                   |
| enddata;                                                  |
| run;                                                      |
| quit;                                                     |

LOG OUTPUT:

```sas
Kilograms2Pounds invar=  13.600000 outvar= 29.982868

NOTE: Execution succeeded. No rows affected.
```

NOTE: PROCEDURE DS2 used (Total process time):
  real time         0.39 seconds
  cpu time          0.23 seconds

Introduced in SAS® 9.3 the High-performance version of the DS2 procedure, PROC HPDS2, executes DS2 language statements in a SAS High-Performance Analytics environment for parallel execution. It is an efficient method for moving big data to the servers or network workstations for distributed processing. PROC HPDS2 is most useful when significant amounts of computationally intensive, row-independent logic must be applied to the data. The HPDS2 procedure offers the following features:
• Enables DS2 code to be executed on a local client machine or on the SAS High-Performance Analytics grid
• Enables control of the level of parallelism per execution node and the number of nodes to engage
• Performs a syntax check of the DS2 code on the local client machine before sending it to the grid for execution
• Manages data migration to the location of execution and movement back to the client machine as needed
• Runs within the framework of SAS Embedded Process

Parallel Processing
One of the key reasons for developing DS2 was to provide parallel processing capability to SAS. Parallel execution is accomplished by executing a program/process in DS2 on either a single machine running multiple threads or to a distributed computing environment if SAS GRID is licensed. Threads refer to the tasks a CPU core can process simultaneously. For instance, many modern PCs have at least 2 cores, and some have 4. Today each of these cores can run up to 2 threads.

When the SAS process is CPU bound the SAS log will indicate REAL time (CPU time is essentially the same) as Running time. Multiple threads in this case can decrease CPU time. If, however, the process is I/O bound, and REAL time is greater than CPU time, then no amount of additional threads will improve efficiency. In fact, adding more threads may decrease overall performance.

The next example was taken from Chapter 6, Example 6.3.1 in Jordan (2016) and modified. The next example illustrates a modified version of the SASHELP.CARS data set and adds some computations to increase processing time using parallel processing. The number of observations was increased 1000 fold to provide a better test for threading. This ran on a Windows 8.1 PC with 4 cores and we specified 4 threads to spread around the processing. The log indicates that all 4 threads were utilized, but thread number 3 received the majority of the action. In general, parallel processing of data programs in DS2 can dramatically reduce the required CPU time if the application is not I/O bound.

**Parallel Processing Code:**

```sas
data cars(drop=i);
    set sashelp.cars(keep=make model type);
    do i = 1 to 1000;
        output;
    end;
run;

proc ds2;
    thread work.myThread(double flag)/overwrite=y;
    dcl int ThreadNo;
    dcl int Count;
    method run();
        set work.cars;
        count+1;
    end;
    method term();
        ThreadNo=_threadid_
        put ThreadNo= Count=;
    end;
endthread;
run;
quit;

proc ds2;
    data cars3 / overwrite=yes;
    dcl double i j k;
    dcl thread work.myThread t;
    method init();
        t.setparms(1);
    end;
    method run();
        set from t threads=4;
        do i = 1 to 100;
            do j = 1 to 50;
                k = (j*i)**3;)
```
Modernizing Legacy SAS® Application and Program Code, continued

PROC DS2;
thread work.myThread(double flag)/overwrite=y;
dcl int ThreadNo;
dcl int Count;
method run();
   set work.cars;
   count+1;
end;
method term();
   ThreadNo=_threadid_;
   put ThreadNo= Count=;
end;
endthread;
run;
NOTE: Created thread mythread in data set work.mythread.
NOTE: Execution succeeded. No rows affected.
quit;

NOTE: PROCEDURE DS2 used (Total process time):
   real time           0.05 seconds
   cpu time            0.04 seconds

proc ds2;
data cars3 / overwrite=yes;
dcl double i j k;
dcl thread work.myThread t;
method init();
   t.setparms(1);
end;
method run();
   set from t threads=4;
   do i = 1 to 100;
      do j = 1 to 50;
         k = (j*i)**3;
      end;
   end;
enddata;
run;
NOTE: There were 428 observations read from the data set SASHELP.CARS.
NOTE: The data set WORK.CARS has 428000 observations and 3 variables.
NOTE: DATA statement used (Total process time):
   real time           0.06 seconds
   cpu time            0.06 seconds

62         data cars(drop=i);
63         set sashelp.cars(keep=make model type);
64         do i = 1 to 1000;
65         output;
66         end;
67         run;
NOTE: There were 428 observations read from the data set SASHELP.CARS.
NOTE: The data set WORK.CARS has 428000 observations and 3 variables.
NOTE: DATA statement used (Total process time):
   real time           0.06 seconds
   cpu time            0.06 seconds

68
69         proc ds2;
70         thread work.myThread(double flag)/overwrite=y;
71         dcl int ThreadNo;
72         dcl int Count;
73         method run();
74            set work.cars;
75            count+1;
76         end;
77         method term();
78            ThreadNo=_threadid_;
79            put ThreadNo= Count=;
80         end;
81         endthread;
82         run;
NOTE: Created thread mythread in data set work.mythread.
NOTE: Execution succeeded. No rows affected.
83         quit;

NOTE: PROCEDURE DS2 used (Total process time):
   real time           0.05 seconds
   cpu time            0.04 seconds

84
85         proc ds2;
86         data cars3 / overwrite=yes;
87         dcl double i j k;
88         dcl thread work.myThread t;
89         method init();
90            t.setparms(1);
91         end;
92         method run();
93            set from t threads=4;
94            do i = 1 to 100;
95               do j = 1 to 50;
96                  k = (j*i)**3;
97               end;
98            end;
99         enddata;
100        run;
ThreadNo=3 Count=220640
ThreadNo=4 Count=120240
ThreadNo=1 Count=25440
ThreadNo=2 Count=61680
NOTE: Execution succeeded. 428000 rows affected.
102 quit;

NOTE: PROCEDURE DS2 used (Total process time):
real time           19.59 seconds
cpu time            20.80 seconds

SAS Grid
Introduced in SAS 9.1.3, the separately licensed SAS Grid Manager adds to the parallel processing capabilities of SAS/CONNECT and many other capabilities required by large data, enterprise grid deployments. It provides efficient resource allocation, load balancing and prioritization for SAS solutions running in a shared grid environment. It also separates the SAS applications from the infrastructure executing the applications. This allows hardware resources to grow or shrink as needed, and provides tolerance for hardware failures in the grid system. The SAS Grid Manager can modernize your existing SAS environment.

- Multiple users - SAS® Enterprise Guide users can submit their SAS programs and tasks to a SAS grid for execution.
- Parallel workloads – Some SAS applications consists of sub-tasks that are independent units of work and can be distributed across a grid and executed in parallel.
- Enterprise scheduling - Scheduling production jobs is an important function in just about every enterprise. SAS provides the Schedule Manager plug-in with SAS® Management Console for creating SAS workflows and schedule them.

SAS Grid Manager integrates the resource management and scheduling capabilities of the Platform Suite for SAS with the SAS 4GL syntax, and subsequently with several SAS products, like Enterprise Miner, SAS® Forecast Server, SAS/STAT, SAS/OR and SAS/ETS. This enables you to create a managed, flexible, and shared environment to efficiently process and analyze data.

According to Galati (2008), “SAS/OR takes advantage of SAS Grid functionality in solving some of the following problems:"

- Monte Carlo methods (Fishman 1995)
- Decomposition methods such as Dantzig-Wolfe, Benders, Lagrangian relaxation (Galati and Ralphs 2005)
- Multi-start methods for global optimization (Horst and Tuy 1993)
- Genetic and evolutionary algorithms (Levine 1994)
- Simulation (Robinson 2004)
- Location problems (Pierre, et al. 2007)

With the SAS In-Database Code Accelerator processing of the data on the server can be transferred to high performance host database systems like HADOOP or TERADATA.

Exploring PROC SCAPROC – The SAS Code Analyzer
Rabb (2010) describes PROC SCAPROC – the SAS Code Analyzer as an effective tool for maintaining and modifying legacy applications and program code. Users are able to capture information about input, output, and macro variables from an executing SAS job with the SCAPROC procedure. In this next example, a file is created with the RECORD statement and the content of the SAS Code Analyzer is written to the file with the WRITE statement.

PROC SCAPROC Code:

```
proc scaproc;
    record '/folders/myfolders/PROC SCAPROC with Record and Write.txt';
run;

proc sort data=sashelp.Cars(keep=Origin Type Make MSRP)
    out=work.Cars_Sorted;
    where MSRP < 20000;  
    by Origin Type Make MSRP;
run;
```
ods HTML path="/folders/myfolders" (url=none)
   file="PROC-PRINT-with-Style.html"
   style=HTMLBlue ;
proc print data=work.Cars_Sorted
   style (data) = [background=Blue foreground=white]
   style (obs)   = [background=red foreground=white]
   style (total) = [background=yellow foreground=black] ;
by Origin Type ;
id Origin Type Make ;
format msrp dollar12.0 ;
sum MSRP ;
run ;
ods HTML close ;

proc scaproc ;
   write ;
run ;

PROC SCAPROC Results:
/* JOBSPLIT: JOBSTARTTIME 09JUL2017:07:10:42.53 */
/* JOBSPLIT: TASKSTARTTIME 09JUL2017:07:10:42.53 */
/* JOBSPLIT: DATASET INPUT SEQ #C00003.CARS.DATA */
/* JOBSPLIT: LIBNAME #C00003 V9 '/opt/sasinside/SASHome/SASFoundation/9.4/sashelp' */
/* JOBSPLIT: LIBNAME SASHELP V9 '(/opt/sasinside/SASHome/SASFoundation/9.4/nls/u8/sascfg'
   '/opt/sasinside/SASHome/SASFoundation/9.4/nls/u8/sashelp'
   '/opt/sasinside/SASHome/SASFoundation/9.4/nls/en/sascfg'
   '/opt/sasinside/SASHome/SASFoundation/9.4/sashelp')' */
/* JOBSPLIT: DATASET OUTPUT SEQ WORK.CARS_SORTED.DATA */
/* JOBSPLIT: LIBNAME WORK V9 '/tmp/SAS_workEF5700002373_localhost.localdomain/SAS_work98EC00002373_localhost.localdomain' */
/* JOBSPLIT: DATASET OUTPUT SEQ WORK.SORTTMP000000000000000000012.DATA */
/* JOBSPLIT: LIBNAME WORK V9 '/tmp/SAS_workEF5700002373_localhost.localdomain/SAS_work98EC00002373_localhost.localdomain' */
/* JOBSPLIT: ITEMSTORE INPUT WORK.TEMPLAT */
/* JOBSPLIT: ITEMSTORE INPUT SASUSER.TEMPLAT */
/* JOBSPLIT: FILE OUTPUT /folders/myfolders/PROC-PRINT-with-Style.html */
/* JOBSPLIT: SYMBOL GET SYSSORTDETAILS */
/* JOBSPLIT: SYMBOL GET SYSSORTTRACELEVEL */
/* JOBSPLIT: SYMBOL GET SYSSORTTRACE */
/* JOBSPLIT: ELAPSED 222 */
/* JOBSPLIT: SYSSCP LIN X64 */
/* JOBSPLIT: PROCNAME SORT */
/* JOBSPLIT: STEP SOURCE FOLLOWS */
proc sort data=sashelp.Cars(keep=Origin Type Make MSRP)
   out=work.Cars_Sorted ;
where MSRP < 20000 ;
by Origin Type Make MSRP ;
run ;
ods HTML path="/folders/myfolders" (url=none)
   file="PROC-PRINT-with-Style.html"
   style=HTMLBlue ;
/* JOBSPLIT: TASKSTARTTIME 09JUL2017:07:10:42.75 */
/* JOBSPLIT: DATASET INPUT MULTI WORK.CARS_SORTED.DATA */
Modernizing Legacy SAS® Applications and Program Code, continued

Conclusion

Large inventories of SAS applications and program code have been developed and supported by organizations since the mid-1970s. In many organizations, novice and experienced SAS® programmers have been tasked with the responsibility to support their organization’s legacy applications, programs and code. This paper explores many tips, techniques and examples to help guide the modernization of applications and program code well into the 21st century and beyond. The recommended techniques and approaches are designed to provide a foundation of things to consider during the modernizing process. We also provide guidance on the use of the SCAPROC procedure – the SAS code analyzer – to analyze metadata about the contents of SAS code, and to streamline, scale and modernize code constructs, algorithms, functions, and legacy application program code.

References

DS2 References and Suggested Reading


Efficiency and Performance Tuning References and Suggested Reading


General References and Suggested Reading


Hash Object References and Suggested Reading


Macro References and Suggested Reading


Operations Research (OR) References and Suggested Reading


PROC FCMP References and Suggested Reading


**PROC SCAPROC References and Suggested Reading**


**SAS Grid References and Suggested Reading**


**SAS Programming Techniques References and Suggested Reading**


Text Analytics References and Suggested Reading


WIN32API References and Suggested Reading


Modernizing Legacy SAS® Applications and Program Code, continued


Microsoft Office Site (2010), Microsoft Office 2010 Win32API_PtrSafe for Visual Basic to download the Win32API_PtrSafe.txt file and related information, then install it).


Microsoft Corporation, MDSN: Develop desktop applications and drivers.


Acknowledgments
The authors thank Clarence Jackson and Greg Gengo, SouthCentral SAS Users Group (SCSUG) Conference Co-Chairs for accepting our abstract and paper; the SouthCentral SAS Users Group (SCSUG) Executive Board; and SAS Institute for organizing and supporting a great conference!

Trademark Citations
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.

Data Sets Used in Examples
The examples presented in this paper include the RUGs_2015 and RUGs_2016 data sets; and several in the SASHELP library including the CARS, HEART, SHOES, and STOCKS data sets You’ll be able to use these data sets for example purposes and for testing the enclosed code examples.

The RUGs_2015 data set consists of 4 observations and 3 variables, illustrated below.

Data Set #1. RUGs_2015

The RUGs_2016 data set consists of 4 observations and 3 variables, illustrated below.

Data Set #2. RUGs_2016

The MOVIES data set consists of 22 observations and 6 variables, illustrated below.
Data Set #3. MOVIES

The ACTORS data set consists of 13 observations and 3 variables, illustrated below.

<table>
<thead>
<tr>
<th>Title</th>
<th>Last Name</th>
<th>First Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shawshank</td>
<td>Redford</td>
<td>Tim</td>
</tr>
<tr>
<td>Untouchable</td>
<td>Pacino</td>
<td>Al</td>
</tr>
<tr>
<td>Pulp Fiction</td>
<td>Tarantino</td>
<td>Quentin</td>
</tr>
<tr>
<td>GoodFellas</td>
<td>DeNiro</td>
<td>Robert</td>
</tr>
<tr>
<td>Taxi Driver</td>
<td>Driver</td>
<td>Baldi</td>
</tr>
<tr>
<td>The Sopranos</td>
<td>Sopranos</td>
<td>James</td>
</tr>
<tr>
<td>Scarface</td>
<td>Pacino</td>
<td>Al</td>
</tr>
<tr>
<td>Silence of the Lambs</td>
<td>Redford</td>
<td>Tim</td>
</tr>
<tr>
<td>The Devil Wears Prada</td>
<td>Zellweger</td>
<td>Meryl</td>
</tr>
<tr>
<td>The Departed</td>
<td>Wahlberg</td>
<td>Martin</td>
</tr>
<tr>
<td>The English Patient</td>
<td>Judd</td>
<td>Richard</td>
</tr>
<tr>
<td>The Godfather</td>
<td>Brando</td>
<td>Marlon</td>
</tr>
<tr>
<td>The Godfather</td>
<td>Pacino</td>
<td>Al</td>
</tr>
<tr>
<td>The Godfather</td>
<td>DeNiro</td>
<td>Robert</td>
</tr>
</tbody>
</table>

Data Set #4. ACTORS

The SASHELP.CARS data set consists of 428 observations and 15 variables, illustrated below.

<table>
<thead>
<tr>
<th>Make</th>
<th>Model</th>
<th>Type</th>
<th>Origin</th>
<th>DriveTrain</th>
<th>EngineType</th>
<th>EngineSize</th>
<th>Cylinders</th>
<th>Horsepower</th>
<th>MPG_City</th>
<th>MPG_Highway</th>
<th>Weight</th>
<th>Wheelbase</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bmw</td>
<td>X5 4.4i</td>
<td>Suv</td>
<td>Asia</td>
<td>4WD</td>
<td>8-cyl</td>
<td>4.4</td>
<td>8</td>
<td>330</td>
<td>18</td>
<td>20</td>
<td>4950</td>
<td>113</td>
<td>191</td>
</tr>
<tr>
<td>Bmw</td>
<td>X5 3.0i</td>
<td>Suv</td>
<td>Europe</td>
<td>AWD</td>
<td>6-cyl</td>
<td>3.0</td>
<td>6</td>
<td>270</td>
<td>25</td>
<td>32</td>
<td>4450</td>
<td>113</td>
<td>191</td>
</tr>
<tr>
<td>Bmw</td>
<td>X1 2.0i</td>
<td>Suv</td>
<td>Europe</td>
<td>AWD</td>
<td>4-cyl</td>
<td>2.0</td>
<td>4</td>
<td>180</td>
<td>26</td>
<td>33</td>
<td>2890</td>
<td>112</td>
<td>176</td>
</tr>
<tr>
<td>Bmw</td>
<td>X3 3.0i</td>
<td>Suv</td>
<td>Europe</td>
<td>AWD</td>
<td>6-cyl</td>
<td>3.0</td>
<td>6</td>
<td>270</td>
<td>25</td>
<td>32</td>
<td>4450</td>
<td>113</td>
<td>191</td>
</tr>
<tr>
<td>Bmw</td>
<td>Z4 2.5i</td>
<td>Suv</td>
<td>Europe</td>
<td>Rear</td>
<td>6-cyl</td>
<td>2.5</td>
<td>6</td>
<td>210</td>
<td>24</td>
<td>33</td>
<td>3380</td>
<td>113</td>
<td>191</td>
</tr>
<tr>
<td>Bmw</td>
<td>Z4 3.0i</td>
<td>Suv</td>
<td>Europe</td>
<td>AWD</td>
<td>6-cyl</td>
<td>3.0</td>
<td>6</td>
<td>270</td>
<td>25</td>
<td>32</td>
<td>4450</td>
<td>113</td>
<td>191</td>
</tr>
<tr>
<td>Bmw</td>
<td>Z4 2.0i</td>
<td>Suv</td>
<td>Europe</td>
<td>AWD</td>
<td>4-cyl</td>
<td>2.0</td>
<td>4</td>
<td>180</td>
<td>26</td>
<td>33</td>
<td>2890</td>
<td>112</td>
<td>176</td>
</tr>
<tr>
<td>Bmw</td>
<td>M5 4.4i</td>
<td>Suv</td>
<td>Europe</td>
<td>4WD</td>
<td>8-cyl</td>
<td>4.4</td>
<td>8</td>
<td>330</td>
<td>18</td>
<td>20</td>
<td>4950</td>
<td>113</td>
<td>191</td>
</tr>
</tbody>
</table>

Data Set #5. SASHELP.CARS
The SASHELP.HEART data set consists of 5,209 observations and 17 variables, illustrated below.

Data Set #6. SASHELP.HEART

The SASHELP.SHOES data set consists of 395 observations and 7 variables, illustrated below.

Data Set #7. SASHELP.SHOES
The SASHELP.STOCKS data set consists of 699 observations and 8 variables, illustrated below.

<table>
<thead>
<tr>
<th>Stock</th>
<th>Date</th>
<th>Open</th>
<th>High</th>
<th>Low</th>
<th>Close</th>
<th>Volume</th>
<th>AdjClose</th>
</tr>
</thead>
<tbody>
<tr>
<td>IBM</td>
<td>01DEC05</td>
<td>$89.15</td>
<td>$89.92</td>
<td>$81.56</td>
<td>$82.20</td>
<td>5.762.252</td>
<td>$81.37</td>
</tr>
<tr>
<td>IBM</td>
<td>01NOV05</td>
<td>$91.05</td>
<td>$90.54</td>
<td>$80.64</td>
<td>$90.90</td>
<td>5.556.471</td>
<td>$90.01</td>
</tr>
<tr>
<td>IBM</td>
<td>01OCT05</td>
<td>$90.22</td>
<td>$94.60</td>
<td>$70.62</td>
<td>$81.88</td>
<td>7.019.666</td>
<td>$90.96</td>
</tr>
<tr>
<td>IBM</td>
<td>01SEP05</td>
<td>$90.16</td>
<td>$92.11</td>
<td>$76.93</td>
<td>$80.22</td>
<td>5.772.280</td>
<td>$79.22</td>
</tr>
<tr>
<td>IBM</td>
<td>01AUG05</td>
<td>$86.00</td>
<td>$84.20</td>
<td>$79.67</td>
<td>$80.62</td>
<td>4.801.368</td>
<td>$79.52</td>
</tr>
<tr>
<td>IBM</td>
<td>01JUL05</td>
<td>$74.30</td>
<td>$85.11</td>
<td>$74.16</td>
<td>$83.46</td>
<td>8.056.580</td>
<td>$82.23</td>
</tr>
<tr>
<td>IBM</td>
<td>01JUN05</td>
<td>$75.57</td>
<td>$77.73</td>
<td>$73.45</td>
<td>$74.20</td>
<td>6.459.336</td>
<td>$73.10</td>
</tr>
<tr>
<td>IBM</td>
<td>02MAY05</td>
<td>$76.98</td>
<td>$78.11</td>
<td>$72.50</td>
<td>$75.55</td>
<td>5.896.994</td>
<td>$74.43</td>
</tr>
<tr>
<td>IBM</td>
<td>01APR05</td>
<td>$71.19</td>
<td>$71.76</td>
<td>$71.05</td>
<td>$73.30</td>
<td>10.709.500</td>
<td>$75.05</td>
</tr>
<tr>
<td>IBM</td>
<td>01MAR05</td>
<td>$32.54</td>
<td>$33.73</td>
<td>$39.09</td>
<td>$31.88</td>
<td>5.925.627</td>
<td>$39.79</td>
</tr>
<tr>
<td>IBM</td>
<td>01FEB05</td>
<td>$33.67</td>
<td>$34.97</td>
<td>$31.55</td>
<td>$32.86</td>
<td>4.485.557</td>
<td>$30.57</td>
</tr>
<tr>
<td>IBM</td>
<td>03JAN05</td>
<td>$88.97</td>
<td>$89.18</td>
<td>$81.44</td>
<td>$83.42</td>
<td>5.960.545</td>
<td>$91.52</td>
</tr>
<tr>
<td>IBM</td>
<td>01DEC04</td>
<td>$94.50</td>
<td>$99.00</td>
<td>$84.47</td>
<td>$86.58</td>
<td>5.043.800</td>
<td>$96.63</td>
</tr>
<tr>
<td>IBM</td>
<td>01NOV04</td>
<td>$89.33</td>
<td>$89.63</td>
<td>$89.23</td>
<td>$94.24</td>
<td>5.754.676</td>
<td>$92.42</td>
</tr>
<tr>
<td>IBM</td>
<td>01OCT04</td>
<td>$95.95</td>
<td>$102.07</td>
<td>$94.29</td>
<td>$98.75</td>
<td>5.039.742</td>
<td>$97.85</td>
</tr>
<tr>
<td>IBM</td>
<td>01SEP04</td>
<td>$84.03</td>
<td>$87.20</td>
<td>$83.24</td>
<td>$85.74</td>
<td>4.719.252</td>
<td>$83.53</td>
</tr>
<tr>
<td>IBM</td>
<td>02AUG04</td>
<td>$86.77</td>
<td>$87.39</td>
<td>$81.96</td>
<td>$84.69</td>
<td>4.238.680</td>
<td>$82.90</td>
</tr>
<tr>
<td>IBM</td>
<td>01JUL04</td>
<td>$88.28</td>
<td>$94.44</td>
<td>$82.42</td>
<td>$87.67</td>
<td>5.923.023</td>
<td>$95.05</td>
</tr>
<tr>
<td>IBM</td>
<td>01JUN04</td>
<td>$88.20</td>
<td>$91.21</td>
<td>$87.30</td>
<td>$88.15</td>
<td>4.504.608</td>
<td>$96.10</td>
</tr>
<tr>
<td>IBM</td>
<td>03MAY04</td>
<td>$88.13</td>
<td>$89.75</td>
<td>$85.12</td>
<td>$89.89</td>
<td>5.395.055</td>
<td>$86.53</td>
</tr>
<tr>
<td>IBM</td>
<td>01APR04</td>
<td>$91.67</td>
<td>$94.55</td>
<td>$88.01</td>
<td>$91.17</td>
<td>5.672.214</td>
<td>$96.95</td>
</tr>
<tr>
<td>IBM</td>
<td>01MAR04</td>
<td>$96.50</td>
<td>$97.60</td>
<td>$90.28</td>
<td>$91.84</td>
<td>5.612.521</td>
<td>$95.93</td>
</tr>
<tr>
<td>IBM</td>
<td>02FEB04</td>
<td>$99.15</td>
<td>$100.43</td>
<td>$95.20</td>
<td>$96.50</td>
<td>5.352.468</td>
<td>$94.07</td>
</tr>
<tr>
<td>IBM</td>
<td>03JAN04</td>
<td>$92.85</td>
<td>$93.65</td>
<td>$89.01</td>
<td>$90.23</td>
<td>5.275.305</td>
<td>$96.57</td>
</tr>
<tr>
<td>IBM</td>
<td>01DEC03</td>
<td>$80.90</td>
<td>$84.12</td>
<td>$80.03</td>
<td>$82.68</td>
<td>4.582.206</td>
<td>$80.20</td>
</tr>
<tr>
<td>IBM</td>
<td>01NOV03</td>
<td>$89.20</td>
<td>$91.48</td>
<td>$87.72</td>
<td>$86.54</td>
<td>5.271.663</td>
<td>$88.12</td>
</tr>
<tr>
<td>IBM</td>
<td>01OCT03</td>
<td>$98.75</td>
<td>$95.54</td>
<td>$87.53</td>
<td>$95.48</td>
<td>6.996.200</td>
<td>$98.59</td>
</tr>
<tr>
<td>IBM</td>
<td>01SEP03</td>
<td>$92.40</td>
<td>$93.47</td>
<td>$82.30</td>
<td>$89.32</td>
<td>6.523.800</td>
<td>$85.81</td>
</tr>
</tbody>
</table>

Data Set #8. SASHELP.STOCKS

Author Bios

Kirk Paul Lafler is an entrepreneur, consultant and founder of Software Intelligence Corporation, and has been using SAS since 1979. Kirk is a SAS application developer, programmer, certified professional, provider of SAS consulting services, mentor, advisor and professor at UC San Diego Extension, educator to SAS users around the world, and emeritus sasCommunity.org Advisory Board member. As the author of six books including Google® Search Complete (Odyssey Press. 2014) and PROC SQL: Beyond the Basics Using SAS, Second Edition (SAS Press. 2013); Kirk has written hundreds of papers and articles; been an Invited speaker and trainer at hundreds of SAS International, regional, special-interest, local, and in-house user group conferences and meetings; and is the recipient of 25 “Best” contributed paper, hands-on workshop (HOW), and poster awards.

Clark Roberts is the principle consultant and founder of Decision Analytics and has been a SAS user since 1979. As a SAS application developer, data analyst, and programmer; Clark is the author of several published papers at SAS International, regional, special-interest, and local user group conferences; and is the recipient of a “Best” contributed paper award.

Comments and suggestions can be sent to:

Kirk Paul Lafler
SAS® Consultant, Application Developer, Programmer, Data Analyst, Educator and Author
Software Intelligence Corporation
E-mail: KirkLafler@cs.com
LinkedIn: http://www.linkedin.com/in/KirkPaulLafler
Twitter: @sasNerd

Clark Roberts
Principle SAS® Consultant and Programmer
Decision Analytics
E-mail: dacmr@hotmail.com