Paper SAS420-2017

More Than Matrices: SAS/IML® Software Supports New Data Structures

Rick Wicklin, SAS Institute Inc.

ABSTRACT

The SAS/IML® language excels in handling matrices and performing matrix computations. A new feature in SAS/IML 14.2 is support for nonmatrix data structures such as tables and lists.

In a matrix, all elements are of the same type: numeric or character. Furthermore, all rows have the same length. In contrast, SAS/IML 14.2 enables you to create a structure that contains many objects of different types and sizes. For example, you can create an array of matrices in which each matrix has a different dimension. You can create a table, which is an in-memory version of a data set. You can create a list that contains matrices, tables, and other lists.

This paper describes the new data structures and shows how you can use them to emulate other structures such as stacks, associative arrays, and trees. It also presents examples of how you can use collections of objects as data structures in statistical algorithms.

INTRODUCTION

Prior to SAS/IML 14.2, the matrix was the only data type for SAS/IML computations. A matrix is a rectangular data type that contains n rows and m columns. Vectors ($n = 1$ or $m = 1$) and scalars ($n = m = 1$) are special cases of matrices.

Matrices arise naturally in all areas of multivariate statistics and are used in statistical computations such as optimization, linear algebra, and simulation. However, for storing data, two-dimensional matrices have some limitations:

- A matrix must contain either all numeric or all character values. You cannot create a matrix in which one column has numeric values and another column has character values.
- A matrix is a fixed size. If you want to insert a new column into an existing matrix, you need to allocate a new larger matrix and copy the old and new data into the larger matrix.
- Each column (and row) of a matrix contains the same number of elements. You cannot create a matrix in which one column contains four elements and another contains forty.
- A matrix is inherently two-dimensional. For some applications you might want to create an array of matrices.
- A matrix is not an ideal structure for representing hierchical or nested data.

Beginning with SAS/IML 14.2, you can create new nonmatrix data structures. One new structure is a *data table* (also called simply a *table*), which is an in-memory version of a data set. A table is a rectangular data structure, but it can hold both numerical and character data. You can efficiently add new columns to tables without reallocating memory and copying the old data. You can create a table from a SAS® data set or from a SAS/IML matrix.

SAS/IML 14.2 also supports lists. The objects in a list can be of different sizes and types. A list can contain disparate data such as numeric matrices, character matrices, tables, and other lists. Lists are a convenient way to store related or hierachical data.

In this paper, the word “variable” can have several meanings. To avoid confusion, this paper uses the following conventions and shows the font used for each term:

- A **variable** is a column in a data set.
- A **column** is a column in a SAS/IML table.
- A **`symbol`** is the name of any SAS/IML matrix, vector, or table.
- A **`dynamic variable`** is a symbol that is named in the DYNAMIC statement in the TEMPLATE procedure.

DATA TABLES IN THE SAS/IML LANGUAGE

Tables are a convenient way to store mixed-type data and to pass data to built-in and user-defined functions. Tables are rectangular, but they do not support algebraic operations. For example, you cannot add two tables together, even if all the columns of the table are numeric.

CREATE A TABLE FROM A SAS DATA SET

You can use the new TableCreateFromDataSet function to create an in-memory table directly from a SAS data set. The following statements read the **Sashelp.Class** data set into a table:

```
proc iml;
tClass = TableCreateFromDataSet("Sashelp", "Class");
```

The **tClass** symbol is the name of a SAS/IML table. The symbol can be passed as an argument to any SAS/IML function that supports tables. For example, you can use the NROW and NCOL functions in SAS/IML to retrieve the number of rows and columns, respectively, for the **tClass** table:

```
nrow = nrow(tClass);
ncol = ncol(tClass);
print nrow ncol;
```

These statements produce Figure 1.

Figure 1 Number of Rows and Columns for a Table

nrow	ncol
19	5

The TableCreateFromDataSet function supports an optional third argument with which you can specify data set options that filter the data. For example, you can use the DROP=, KEEP=, OBS=, and WHERE= options to restrict data, as shown in the following statements:

```
dsOpt = "drop=Name rename=(Weight=Mass) where=(Sex='M')";
tBoys = TableCreateFromDataSet("Sashelp", "Class", dsOpt); /* filter data */
```

Because of the WHERE= option, the **tBoys** table contains only male students. Because of the DROP= option, the table does not contain the **Name** variable. Because of the RENAME= option, the **Weight** variable is renamed to **Mass**.

In a similar way, the TableWriteToDataSet subroutine creates a SAS data set from an in-memory table. The subroutine supports an optional argument with which you can specify data set options.

EXTRACT AND ADD DATA COLUMNS

A powerful feature of the SAS/IML table is that you can use the TableAddVar subroutine to efficiently add new columns of data to the table. You can use the TableGetVarData function to extract one or more columns from the table into a matrix.

For example, the following statements create a data table from the **Sashelp.Class** data set, extract the **Weight** and **Height** variables into vectors, and apply a standard formula to compute the body mass index (BMI) for each student in the data. The BMI values are then added to the table as a new column.

```
/* Compute BMI from the Height and Weight vars */
X = TableGetVarData(tClass, {"Weight" "Height"});
weight = X[,1];  height = X[,2];
BMI = weight / height##2 * 703;       /* standard formula        */
call TableAddVar(tClass, "BMI", BMI); /* add "BMI" col to tClass */
```

As indicated by the example, the TableGetVarData function takes two arguments: a table and a vector of column names. The return value is a SAS/IML matrix. Similarly, the TableAddVar subroutine takes three arguments: a table, a vector of column names, and a SAS/IML matrix that contains the data.

GET AND SET ATTRIBUTES OF VARIABLES

SAS/IML provides many functions that you can use to obtain attributes of columns in a table. Most functions return a row vector that provides information about the columns, such as names, labels, and types (character or numeric). For example, suppose that you create a table from the **Sashelp.Cars** data set and you want to obtain attributes of some of the variables. In Base SAS® software, you can use the CONTENTS procedure to discover the type, format, and label of each variable. In SAS/IML, you can call functions that provide the same information, as shown by the following example:

```
tCars = TableCreateFromDataSet("Sashelp", "Cars");       /* create table */
varNames = {"Origin" "Invoice" "EngineSize" "MPG_City"}; /* column names */
Type   = TableGetVarType(tCars, varNames);               /* get types    */
Format = TableGetVarFormat(tCars, varNames);             /* get formats  */
Label  = TableGetVarLabel(tCars, varNames);              /* get labels   */
Attribs = Type // Format // Label;             /* vertical concatenation */
print Attribs[colname=varNames rowname={"Type" "Format" "Label"}];
```

Figure 2 shows attributes for four variables. The output shows that three variables are numerical, one has a format, and two have labels.

Figure 2 Attributes of Variables

	Attribs			
	Origin	**Invoice**	**EngineSize**	**MPG_City**
Type	C	N	N	N
Format		DOLLAR8.		
Label			Engine Size (L)	MPG (City)

In a similar way, SAS/IML provides subroutines that enable you to set attributes for columns of a table. For example, the TableSetVarFormat subroutine sets formats and the TableSetVarLabel subroutine sets labels that are associated with columns.

PASS TABLES TO MODULES

Tables are ideal for holding the results of statistical computations. The following SAS/IML function computes descriptive statistics and places the results in a table. The function returns a table of statistics.

```
proc iml;
/* Compute descriptive statistics for numerical vars in 'tbl'.
   Return k x 6 table with statistics */
start DescStats( tbl );
   cols = loc( TableIsVarNumeric(tbl) ); /* get column numbers               */
   if ncol(cols)=0 then do;              /* no numeric columns in table      */
      return TableCreate();              /* return an empty table            */
   end;
   vars = TableGetVarName(tbl, cols);    /* names of numerical vars          */
   T = TableCreate("Variable", T(vars));
   m = TableGetVarData(tbl, cols);       /* extract numeric data into matrix */
   N    = T(countn(m, "col"));           /* compute descriptive stats        */
   mean = T(mean(m));
   std  = T(std(m));
   min  = T(m[><, ]);
   max  = T(m[<>, ]);

   call TableAddVar(T, "N", N);              /* num obs for each column */
   call TableAddVar(T, "Mean", mean);        /* mean    for each column */
   call TableAddVar(T, "StdDev", std);       /* std dev for each column */
   call TableAddVar(T, "Minimum", min);      /* minimum for each column */
   call TableAddVar(T, "Maximum", max);      /* maximum for each column */
   call TableSetVarFormat(T, {"Mean" "StdDev"}, {"7.3" "7.3"});
   return T;
finish;
```

```
table = TableCreateFromDataSet("Sashelp", "Class");
stats = DescStats(table);              /* stats for each numeric column */
```

The preceding program demonstrates a few features of tables:

- You can pass tables to user-defined functions. You can also return tables from functions.
- The TableIsVarNumeric function returns a binary vector that indicates which columns in a table are numeric.
- To perform numerical computations, you can use the TableGetVarData function to extract numerical columns into a matrix.
- You can use the TableSetVarFormat function to set the formats for columns. Those formats are used if you save or print a table.

PRINT TABLES

SAS/IML supports the TablePrint subroutine for displaying an in-memory table. The previous section creates a table called **stats**. If you want to see the contents of the table, call the TablePrint subroutine as follows:

```
call TablePrint(stats);
```

Figure 3 shows the output from the most basic usage of the TablePrint subroutine.

Figure 3 Descriptive Statistics for Numeric Columns

stats

Obs	Variable	N	Mean	StdDev	Minimum	Maximum
1	Age	19	13.316	1.493	11	16
2	Height	19	62.337	5.127	51.3	72
3	Weight	19	100.026	22.774	50.5	150

The following list describes a few of the basic options for the TablePrint subroutine. See the *SAS/IML User's Guide* for the complete list.

- The ID= option specifies a column to use as row headers. By default, row numbers are used as row headers.
- The JUSTIFY= option specifies the horizontal alignment (left, center, or right) for each column. By default, character columns are left-justified and numeric columns are right-justified.
- The LABEL= option specifies a header for the entire table. By default, the subroutine prints the name of the SAS/IML symbol as the table header.
- The NUMOBS= option specifies the number of rows to print. By default, all rows are printed.
- The VAR= option specifies which columns to print. By default, all columns are printed.

The simplest way to use the options is to specify keyword-value pairs after the TablePrint call but before the semicolon, as follows. To demonstrate these options, the following statement prints the **stats** table that was defined earlier. Notice that keyword-value pairs are used to specify the arguments and that you can specify the arguments in any order.

```
call TablePrint(stats)  var={"Minimum" "Mean" "Maximum"}
                        ID="Variable"
                        label="Descriptive Statistcs"
                        justify={'R' 'C' 'R' 'C'};
```

Figure 4 shows the output.

Figure 4 Table Printed by Using Options

Descriptive Statistcs			
Variable	**Minimum**	**Mean**	**Maximum**
Age	11	13.316	16
Height	51.3	62.337	72
Weight	50.5	100.026	150

PRINT TABLES BY USING A CUSTOM TEMPLATE

For more sophisticated printing, you can specify an ODS template to be used to display the table. You can use a built-in template from a statistical procedure, or you can define your own template. This section demonstrates how you can use the TEMPLATE procedure to define and use a custom template. For an introduction to using PROC TEMPLATE, see Smith (2007, 2013).

Recall that the **Sashelp.Class** data set includes variables that indicate gender and weight. When you display the **Sashelp.Class** data, you might want to highlight males by using a light blue background and females by using a pink background. Furthermore, you might want to highlight any child who weighs more than 100 pounds by using a light orange background. The following call to PROC TEMPLATE defines a table template (named CustomColor1) that uses the CELLSTYLE statement to color cells that satisfy certain criteria:

```
proc template;
define table CustomColor1;
   cellstyle _COL_ = 2 && _VAL_ = "M" as {backgroundcolor=LightBlue},
             _COL_ = 2 && _VAL_ = "F" as {backgroundcolor=Pink},
             _COL_ = 5 && _VAL_ > 100 as {backgroundcolor=LightOrange};
end;
run;
```

The following call to the TablePrint subroutine uses the TEMPLATE= option to specify the name of the template to be used to display the table:

```
proc iml;
tbl = TableCreateFromDataSet("Sashelp", "Class");
call TablePrint(tbl) numobs=6
                     template="CustomColor1";
```

In Figure 5, the cells in the **Sex** column are colored pink or blue. Cells in the **Weight** column are colored orange for students who weigh more than 100 pounds. The table does not have a main header because the template does not define one.

Figure 5 Color Cells According to Their Values

Name	Sex	Age	Height	Weight
Alfred	M	14	69	112.5
Alice	F	13	56.5	84
Barbara	F	13	65.3	98
Carol	F	14	62.8	102.5
Henry	M	14	63.5	102.5
James	M	12	57.3	83

The previous example uses hard-coded values for the colors, but templates also support dynamic variables that can be specified at run time. The DYNAMIC= option in the TablePrint subroutine enables you to specify values for dynamic variables in templates.

The following call to PROC TEMPLATE defines a table (named CustomColor2) that uses dynamic variables:

```
proc template;
define table CustomColor2;
   dynamic MaleColor FemaleColor OverweightColor;
   cellstyle _COL_ = 2 && _VAL_ = "M" as {backgroundcolor=MaleColor},
             _COL_ = 2 && _VAL_ = "F" as {backgroundcolor=FemaleColor},
             _COL_ = 5 && _VAL_ > 100 as {backgroundcolor=OverweightColor};
end;
run;
```

This version of the template uses the DYNAMIC statement to indicate that the values for the symbols **MaleColor**, **FemaleColor**, and **OverweightColor** can be specified at run time. You can specify dynamic variables by using the DYNAMIC= option in the TablePrint subroutine.

The value of the DYNAMIC= option in the TablePrint subroutine is a character vector of keyword-value pairs. Each element of the vector specifies the name of a dynamic variable in a template, an equal sign, and the name of a SAS/IML symbol whose value will be used for the corresponding dynamic variable.

For example, the following program contains three SAS/IML symbols: **M**, **F**, and **Wt**. Each symbol contains a string that specifies a valid SAS color name. The **dynamicVar** vector contains three elements, each of which specifies a keyword-value pair, where the keyword specifies a dynamic variable in the CustomColor2 template. The first element of the vector indicates that the value of the MaleColor dynamic variable will be "LightGreen," which is the value of the symbol **M**. The second element indicates that the value of the FemaleColor dynamic variable will be "LightRed," and so on.

```
proc iml;
tbl = TableCreateFromDataSet("Sashelp", "Class");
M = "LightGreen";                  /* color for males          */
F = "LightRed";                    /* color for females        */
Wt = "LightGrey";                  /* color for heavy students */
/* Syntax: DynamicVar1=Symbol1, DynamicVar2=Symbol2, ...  */
dynamicVar = {"MaleColor=M" "FemaleColor=F" "OverweightColor=Wt"};
call TablePrint(tbl) numobs=6
                     template="CustomColor2"
                     dynamic=dynamicVar;
```

The output is shown in Figure 6.

Figure 6 Output from a Custom Template That Contains Dynamic Variables

Name	Sex	Age	Height	Weight
Alfred	M	14	69	112.5
Alice	F	13	56.5	84
Barbara	F	13	65.3	98
Carol	F	14	62.8	102.5
Henry	M	14	63.5	102.5
James	M	12	57.3	83

For even more advanced printing options, you can use the DEFINE COLUMN statement in PROC TEMPLATE to customize the format, style, justification, and other features of a column. The TablePrint subroutine supports column-specific rendering through the COLTEMPLATE= option. See the *SAS/IML User's Guide* for details.

LISTS IN THE SAS/IML LANGUAGE

In this paper, the word "list" refers to a SAS/IML data structure that can contain other data structures. A list is a container object. An item in a list can be a matrix, a table, or another list. Lists enable you to store and access objects of different types, shapes, and sizes. Lists are a convenient way to store related data and to pass that data to modules.

You can use the ListAddItem and ListInsertItem subroutines to insert new items into a list. You can use the ListDeleteItem subroutine to remove items from a list. You can use the ListSetItem subroutine to modify an existing item.

You cannot perform arithmetic operations on lists. For example, you cannot add two lists. However, you can extract the data into matrices and perform a computation on the matrices.

LISTS AS DYNAMIC ARRAYS

A SAS/IML list is similar to a *dynamic array*. A dynamic array (also called a *growable array*) is a random-access array that can grow and shrink. Items in a dynamic array are directly accessed by specifying their position (index).

You can use lists to hold matrices of various sizes and shapes. This section shows you how to do the following:

- create a list by using the ListCreate function
- add items to a list by using the ListAddItem subroutine
- modify items in a list by using the ListSetItem subroutine
- insert items in a list by using the ListInsertItem subroutine
- delete items in a list by using the ListDeleteItem subroutine

Suppose you want to store two matrices. You can use the ListCreate function to create a list that contains room for two items. You can then use the ListSetItem subroutine to set values for the items, as follows:

```
proc iml;
L = ListCreate(2);                  /* L is a list of 2 items      */
call ListSetItem(L, 1, 1:3);        /* 1st item of L is vector 1:3 */
call ListSetItem(L, 2, {4 3, 2 1}); /* 2nd item of L is 2x2 matrix */
```

As demonstrated by the ListSetItem subroutine, the first argument to most list-related functions is the name of the list object (**L**). The second argument specifies positions or names of items. For the ListSetItem subroutine, the third item specifies the value. This is the same order (item, index, value) that you would use to assign a matrix element by using the matrix syntax **M[i]=value**.

If you use the ListAddItem subroutine to add more items to the list, the list will automatically grow to accommodate the new items. The new items are added to the end of the list, so after the call the list **L** contains three items.

```
X = {3 1, 4 2, 5 3};                /* define 3x2 matrix      */
call ListAddItem(L, X);             /* add X as 3rd item of L */
```

You can also insert a new item into the middle of a list. The indices of later items are incremented. For example, the following statement inserts a new item as the second item of the list. After the call, the items that were originally the second and third items become the third and fourth items, respectively.

```
call ListInsertItem(L, 2, -1:1);    /* insert new item into L */
```

You can use the ListSetItem subroutine to replace or modify an item. For example, the following statement replaces the first item in the list, which is currently a three-element numeric vector, with a four-element character matrix:

```
call ListSetItem(L, 1, {A B, C D}); /* modify 1st item of L */
```

You can use the ListGetItem function to extract an item. For example, the following statement extracts the third list item into a SAS/IML matrix:

```
S = ListGetItem(L, 3);              /* copy 3rd item of L to S */
```

If you decide that you no longer want an item in a list, you can use the ListDeleteItem subroutine to delete the item. For example, the following statement deletes the third item in the list:

```
call ListDeleteItem(L, 3);          /* delete 3rd item in list */
```

It is worth mentioning that the ListGetItem function (and other list-related functions) supports an optional third argument that specifies whether the list changes after the item is extracted. By default, a list item is copied from the list ('c') but the list does not change. Alternatively, you can delete the item ('d') or move the item ('m'). When you delete an item, the length of the list decreases. When you move an item, the list item is set to the empty matrix. So, for example, the previous two statements could have been combined into a single statement: **`S = ListGetItem(L, 3, 'd')`**. An example is provided later in this paper.

THE LISTUTIL PACKAGE AND THE STRUCTURE OF LISTS

The previous section shows that you can dynamically add, insert, delete, and modify items in a list. When you modify a list multiple times, it can be difficult to know the state of the list. For example, at the end of the previous section, how many items does the list **`L`** contain? What types of items are in the list? What are the sizes of the items?

Fortunately, there are programmatic ways to answer these questions. Continuing the example from the previous section, you can use the ListLen function to obtain the number of items in a list, as shown in Figure 7:

```
numItems = ListLen(L);
print numItems;
```

Figure 7 The Number of Items in a List

numItems
3

If you want detailed information about the type and size of items in a list, you can load the ListUtil package, which is distributed and installed as part of SAS/IML 14.2. The ListUtil package contains two modules that display information about lists:

- The Struct subroutine prints a table that shows the structure of any SAS/IML object.
- The ListPrint subroutine prints items in a list. If a list contains sublists, they are also printed.

For example, you can call the Struct module to display a high-level description of a list, as shown in Figure 8:

```
package load ListUtil;
run Struct(L);
```

Figure 8 Display the Structure of a List

				L				
Name	**Level**	**NRow**	**NCol**	**Type**	**Value1**	**Value2**	**Value3**	**Value4**	**More**
L	0	.	3	List	L[1]	L[2]	L[3]		
=> L[1]	1	2	2	Char	A	B	C	D	
=> L[2]	1	1	3	Num	-1	0	1		
=> L[3]	1	3	2	Num	3	1	4	2	...

The Struct subroutine displays a table that summarizes the list. The first row shows that the list is named **`L`** and that it contains three items. The items are not named, so they are displayed as L[1], L[2], and L[3].

The second row of Figure 8 summarizes L[1], the first item in the list. The item is a 2×2 character matrix whose first four values are A, B, C, and D. The third row summarizes L[2], the second item in the list. The item is a 1×3 numeric matrix whose values are –1, 0, and 1. The fourth row summarizes L[3], which is a 3×2 numeric matrix. The ellipsis in the last column indicates that the item contains values that are not displayed. Notice that values are listed in row-major order.

Figure 8 is a convenient summary of the items in a list. If you prefer to print the items, use the ListPrint subroutine, as follows:

```
run ListPrint(L);
```

The output from the ListPrint subroutine is shown in Figure 9. The output is self-explanatory. The first output is a text string that shows the name of the list. Each item is then printed. If the items were named, the matrix labels would contain the item names instead of the generic labels "Item 1," "Item 2," and "Item 3."

Figure 9 Print Each Item in a List

--------- List = L---------

Item 1

A	B
C	D

Item 2

-1	0	1

Item 3

3	1
4	2
5	3

OPERATIONS ON LIST ITEMS

As stated earlier, you cannot perform arithmetic operations on the items of a list. However, you can extract the item into a SAS/IML matrix or vector and then perform any matrix computation.

To illustrate this technique, consider a list that holds three vectors. In the following example, the data vectors have different lengths, so the data do not fit neatly into a matrix:

```
proc iml;
X = {1,3,5,7,9};                    /*  5 obs */
Y = {2,4,3,4,4,1};                  /*  6 obs */
Z = {1,2,3,4,5,6,7,8,9,10};         /* 10 obs */
Lst = ListCreate();
call ListAddItem(Lst, X);
call ListAddItem(Lst, Y);
call ListAddItem(Lst, Z);
```

Suppose you want to compute the mean for each item in a list. You can iterate over the items in the list. For each item, you can extract the data into a vector and compute the mean, as follows:

```
means = j(1, ListLen(Lst));         /* allocate result vector */
do i = 1 to ListLen(Lst);           /* for each item          */
   v = ListGetItem(Lst, i);         /*    extract data        */
   means[i] = mean(colvec(v));      /*    save mean of vector */
end;

print means[colname={X Y Z}];
```

Figure 10 shows the results.

Figure 10 Mean Value of Each List Item

means

X	Y	Z
5	3	5.5

LISTS, STRUCTS, and ASSOCIATIVE ARRAYS

By using lists in SAS/IML, you can emulate many different data structures. This section describes structs and associative arrays. Stacks and trees are discussed in subsequent sections.

A *struct* is a collection of named items called *members*. The members can be *inhomogeneous*, which means they do not have to be the same type or size.

You can use SAS/IML lists to emulate a struct. For example, suppose you are storing information about students who are taking a statistics course. For each student, you want to record the student's name, class period, and test scores. If you use matrices to store these data, you need to use multiple matrices, as shown by the following example:

```
proc iml;
Name = "Ronald Fisher";             /* name of student     */
Period = 3;                         /* in 3rd period class */
Scores = {100 97 94 100 100};       /* test scores         */
```

It is often convenient to use a list to group these data into a single object that you can pass into modules:

```
Student = ListCreate(3);
call ListSetItem(Student, 1, Name);
call ListSetItem(Student, 2, Period);
call ListSetItem(Student, 3, Scores);
```

The **Student** symbol is a list object that contains three items. For an object that contains many items, you might find it hard to remember what information is stored in each item. However, you can name the items in the list and refer to the items by their names. When you name items in a list, the list is similar to an associative array.

An *associative array* (also called a *map* or a *dictionary*) is a set of key-value pairs. You can specify items in an associative array by using the key. In the SAS/IML language, you can use the ListSetName subroutine to assign names to some or all items, as follows:

```
call ListSetName(Student, 1:3, {"Name" "Class Period" "Scores"});
```

In the previous statement, the second argument (**1:3**) specifies the indices of the items and the third argument specifies the corresponding names for the items.

You can access a named item by its name in addition to its index. For example, the following statements are equivalent; they each retrieve the test scores for the student:

```
S1 = ListGetItem(Student, "Scores"); /* extract by name  */
S2 = ListGetItem(Student, 3);        /* extract by index */
```

You might know at the outset that you want to access items by using names. In that case, you can use an alternate syntax of the ListCreate function to create a named array. The following statements show an alternate way to create and fill a named array:

```
Student = ListCreate({"Name" "Class Period" "Scores"});
call ListSetItem(Student, "Name",         Name);
call ListSetItem(Student, "Class Period", Period);
call ListSetItem(Student, "Scores",       Scores);
```

In the alternative syntax, the item names are defined in the call to the ListCreate function, which allocates space for three items and assigns the names. The subsequent calls to the ListSetItem subroutine use a name instead of an index to reference the items.

As described previously, you can use the Struct module in the ListUtil package to display a summary of the contents of the list:

```
package load ListUtil;
run Struct(Student);
```

Figure 11 shows the results.

Figure 11 Summary of Student Information

Student									
Name	**Level**	**NRow**	**NCol**	**Type**	**Value1**	**Value2**	**Value3**	**Value4**	**More**
Student	0	.	3	List	Name	Class Period	Scores		
=> Name	1	1	1	Char	Ronald Fisher				
=> Class Period	1	1	1	Num	3				
=> Scores	1	1	5	Num	100	97	94	100	...

PASS LISTS TO MODULES

Suppose you want to write a SAS/IML module that computes a regression. To perform the computation, the module requires information about the data and the model. For example, you might pass several parameters to the module, such as a data table, a string that specifies the response variable, and a vector of strings that specify the independent variables. For a sophisticated regression algorithm, the algorithm might also require character or numeric arguments that control the regression algorithm. For a complicated regression module, you might need to pass in a dozen or more parameters.

A primary usage of SAS/IML lists is to store related items so that you can pass a single object to modules. As shown in the preceding section, you can pack the data and parameters into an associative array. You can then pass that one object to the module. The module can unpack the relevant components and perform its computations.

For example, the following module implements an ordinary least squares regression:

```
proc iml;
/* Regression module takes a list that contains three items:
   "Data"     : Table that contains the data to analyze
   "DepVar"   : Name of the dependent variable
   "IndepVar" : Row vector that contains the names
                of the independent variables in the model
   Assume numerical and nonmissing values.
*/
start Regression(S);
  /* 1. Extract data and variable names from argument                */
  tbl = ListGetItem(S, "Data");          /* data in table             */
  YName = ListGetItem(S, "DepVar");      /* dependent variable name   */
  Y = TableGetVarData(tbl, YName);       /* response values           */
  XNames = ListGetItem(S, "IndepVar");   /* independent variable names */
  n = nrow(tbl);                         /* assume nonmissing values  */
  p = 1 + ncol(XNames);                  /* number of effects         */
  X = j(n, p, 1);                        /* add intercept column      */
  X[,2:p] = TableGetVarData(tbl, XNames);  /* data matrix             */

  /* 2. Compute parameter estimates and related statistics           */
  xpxi = inv(X`*X);                    /* inverse of X'X            */
  b = xpxi * (X`*Y);                   /* parameter estimates       */
  resid = y - X*b;                     /* residuals                 */
  dfe = n - p;                         /* error degrees of freedom  */
  mse = ssq(resid)/dfe;                /* mean square error         */
  stdb = sqrt(vecdiag(xpxi)*mse);      /* standard error of estimates */
  t = b / stdb;                        /* t statistic               */
  prob = 1-cdf("F", t#t, 1, dfe);      /* p-values                  */

  /* 3. Create and print parameter estimates table                   */
  PE = TableCreate({"Estimate" "StdErr" "tValue" "Probt"},
                       b    || stdb  || t     || prob );
  call TableAddVar(PE, "Variable", "Intercept" // XNames`);
  call TablePrint(PE) template="Stat.Reg.ParameterEstimates";
finish;
```

The argument to the module is a named list that contains three items. The “data” item contains the data. The “DepVar” item is a character string that specifies the name of the response variable in the table. The “IndepVar” item is a vector

of strings that specifies the names of the independent variables in the table. The module performs the following three tasks:

1. Unpacks the items from the list and extracts the data into a response vector **Y** and a data matrix **X**.
2. Uses matrix computations to obtain parameter estimates and related statistics for the regression model.
3. Creates a table that contains the statistics and prints the table by using the same ODS template (Stat.Reg.ParameterEstimates) that the REG procedure uses.

To call the module, you need to pack the data and options into a list. The following statements create a data table from the **Sashelp.Class** data. The data are added to a list, as are the names of the dependent and independent variables. Finally, the module is called. The results are shown in Figure 12 for a model that predicts **Weight** as a linear function of **Height** and **Age**.

```
/* Pack data and options into a list and call the module */
Model = ListCreate({"DepVar" "IndepVar" "Data"});
tClass = TableCreateFromDataSet("Sashelp", "Class");
call ListSetItem(Model, "Data",      tClass);
call ListSetItem(Model, "DepVar",    "Weight");
call ListSetItem(Model, "IndepVar", {"Height" "Age"});
run Regression( Model );
```

Figure 12 Parameter Estimates Table

Parameter Estimates

Variable	Parameter Estimate	Standard Error	t Value	Pr > |t|
Intercept	-141.22376	33.38309	-4.23	0.0006
Height	3.59703	0.90546	3.97	0.0011
Age	1.27839	3.11010	0.41	0.6865

Figure 12 indicates that the **Age** variable is not statistically significant. The following statements drop the **Age** variable from the model and refit the model. The final result is shown in Figure 13.

```
call ListSetItem(Model, "IndepVar", {"Height"});
run Regression( Model );
```

Figure 13 Revised Parameter Estimates Table

Parameter Estimates

Variable	Parameter Estimate	Standard Error	t Value	Pr > |t|
Intercept	-143.02692	32.27459	-4.43	0.0004
Height	3.89903	0.51609	7.55	<.0001

STACKS

An elementary data structure in computer science is a stack. A *stack* is a linear array in which objects can be inserted and removed only at the beginning of the array. A push operation adds an item to the front of the array; a pop operation removes the item at the front of the array. A stack obeys the last-in first-out (LIFO) principle.

When you use SAS/IML lists to emulate a sophisticated data structure, it is often useful to define helper functions that use list-related functions to implement methods for working with the data structure.

For example, if you want to use SAS/IML lists to emulate a stack, you can use the ListInsertItem operation to implement the push operation and use the ListGetItem function to implement the pop operation. To demonstrate this fact, the SAS/IML Sample Library contains the following modules that use lists and list operations to emulate stacks and stack operations:

- The StackCreate function initializes a stack. When called with zero arguments, the function returns an empty stack.
- The StackPush subroutine pushes an item onto the top of an existing stack.
- The StackPop function returns the item at the top of the stack and removes the item from the stack.
- The StackPeek function returns the item at the top of the stack but does not change the stack.
- The StackIsEmpty function returns 1 if the stack is empty and 0 if the stack contains at least one item.

The modules are thin wrappers around list-related functions. Many of the functions contain only one line. The wrappers hide implementation details from the user. For example, in theory the push operation adds items to the top of a stack. In practice, however, the StackPush subroutine uses the ListAddItem subroutine to add items to the *end* of a list. The following functions implement the push, pop, and peek operations for stacks while hiding the details of how these operations are implemented:

```
proc iml;
/* Define helper functions for a stack, which is a 1-D LIFO structure */
start StackCreate( item= );
   S = ListCreate();                /* create empty list     */
   if ^IsSkipped(item) then         /* if item is specified, */
      call ListAddItem(S, item);    /*    add item to list   */
   return S;
finish;

/* Push an item onto the stack */
start StackPush(S, item);
   call ListAddItem(S, item);       /* add item to the end */
finish;

/* Pop an item from the stack */
start StackPop(S);
   return ListGetItem(S, ListLen(S), 'd'); /* get last item; delete from list */
finish;

/* Peek at the item at the top of the stack without removing it */
start StackPeek(S);
   return ListGetItem(S, ListLen(S)); /* get last item; list unchanged */
finish;

/* Return 1 if stack is empty; 0 otherwise */
start StackIsEmpty(S);
   return (ListLen(S) = 0);
finish;
```

To demonstrate how to use these functions, the following example implements a common programming task that uses stacks. The task is to use the push operation to build a stack of words, and then use the pop operation to retrieve the words in reverse order. The following statements define a string and use the COUNTW and SCAN functions in Base SAS to break the string into a vector that has one word per element:

```
/* Create sentence */
str = "Now is the time for all good men to come to the aid of their party.";
n = countw(str, " .");              /* blanks and periods are delimiters */
words = scan(str, 1:n, " .");       /* break string into vector of words */

S = StackCreate();                  /* create an empty stack             */
do i = 1 to ncol(words);
   run StackPush(S, words[i]);      /* push each item onto the stack     */
end;
print (StackPeek(S))[L="Top of Stack"]; /* the last word is on top       */
```

```
/* Retrieve the data in reverse order */
w = j(1, ncol(words), "     ");
do i = 1 to ncol(w);                /* pop each item; insert into vector */
   w[i] = StackPop(S);
end;
print w[L="Reversed Words"];

if StackIsEmpty(S) then
   print "Stack is empty";
else print (StackPeek(S))[L="Top of Stack"];
```

The output from the program is shown in Figure 14. The program illustrates pushing items onto a stack and popping items off the stack. In practice there are simpler ways to reverse the words in a sentence, such as the matrix expression **words[,ncol(words):1]**. However, this exercise demonstrates how you can use SAS/IML lists to emulate a stack.

Figure 14 Using a Stack to Reverse Words

Top of Stack
party

Reversed Words
party their of aid the to come to men good all for time the is Now

Stack is empty

In the same way, you can define helper functions for a queue, which is a one-dimensional first-in first-out (FIFO) data structure. See the *SAS/IML User's Guide* and the SAS/IML Sample Library for details.

LISTS OF LISTS

A previous example uses a list to store data for a student in a class. If you have data for multiple students, you might want to create an array of structures. This section demonstrates how to implement an array of data structures by using a list of lists.

Rather than continue with the student example, this section uses a list of lists to represent a multivariate finite-mixture distribution. Recall that the density for a finite-mixture distribution is a linear combination of component densities, where the coefficients are the mixing probabilities. For example, if $X_i \sim N(\mu_i, \Sigma_i)$, are multivariate normal random variables, then $Y = \Sigma_i^k \pi_i X_i$ is a mixture of k normal components, where the π_i are the mixing probabilities such that $\Sigma_i^k \pi_i = 1$. Thus to specify the parameters of the mixture distribution, you need to specify the mixing probabilities, the mean vectors, and the covariance matrices for each of the component distributions.

It makes sense to use a struct to hold the information for each component and to use an array of structs to hold the information for the mixture distribution. In the following program, the **comp** symbol is a three-item structure that contains the mixing probability, the mean vector, and the covariance matrix for each normal component. The **Mixture** symbol is an array that contains parameters for a mixture of $k = 3$ components.

```
proc iml;
Mixture = ListCreate();
/* First component of mixture of multivariate normals */
comp = ListCreate({"MixProb" "mu" "Sigma"});
call ListSetItem(comp, "MixProb", 0.35);
call ListSetItem(comp, "mu",    {32 16  5});
call ListSetItem(comp, "Sigma", {17  7  3,
                                  7  5  1,
                                  3  1  1});
call ListAddItem(Mixture, comp);    /* copy structure into Mixture */
```

```
/* Second component of mixture (reuse same structure) */
call ListSetItem(comp, "MixProb", 0.5);
call ListSetItem(comp, "mu",    {30  8  4});
call ListSetItem(comp, "Sigma", {90 27 16,
                                 27  9  5,
                                 16  5  4});
call ListAddItem(Mixture, comp);    /* copy structure into Mixture */

/* Third component of mixture (reuse same structure) */
call ListSetItem(comp, "MixProb", 0.15);
call ListSetItem(comp, "mu",    {49  7  5});
call ListSetItem(comp, "Sigma",{103 16 11,
                                 16  4  2,
                                 11  2  2});
call ListAddItem(Mixture, comp);    /* copy structure into Mixture */

package load ListUtil;
run Struct(Mixture);
```

Figure 15 shows the structure of the list of lists. The main list (**Mixture**) contains three sublists. The **Level** column in the output indicates whether an item is a list, a sublist, or an item in a sublist.

Figure 15 A List of Lists

Mixture									
Name	**Level**	**NRow**	**NCol**	**Type**	**Value1**	**Value2**	**Value3**	**Value4**	**More**
Mixture	0	.	3	List	Mixture[1]	Mixture[2]	Mixture[3]		
=> Mixture[1]	1	.	3	List	MixProb	mu	Sigma		
=> => MixProb	2	1	1	Num	0.35				
=> => mu	2	1	3	Num	32	16	5		
=> => Sigma	2	3	3	Num	17	7	3	7	...
=> Mixture[2]	1	.	3	List	MixProb	mu	Sigma		
=> => MixProb	2	1	1	Num	0.5				
=> => mu	2	1	3	Num	30	8	4		
=> => Sigma	2	3	3	Num	90	27	16	27	...
=> Mixture[3]	1	.	3	List	MixProb	mu	Sigma		
=> => MixProb	2	1	1	Num	0.15				
=> => mu	2	1	3	Num	49	7	5		
=> => Sigma	2	3	3	Num	103	16	11	16	...

An advantage of storing parameters in a list is that you can pass the list to user-defined functions. For example, the following function simulates a random sample from a mixture of multivariate Gaussian distributions. The function accepts a list that contains any number of component functions. The components can be for data of any dimension. The module performs the following computations, which are described in Wicklin (2013, p. 140):

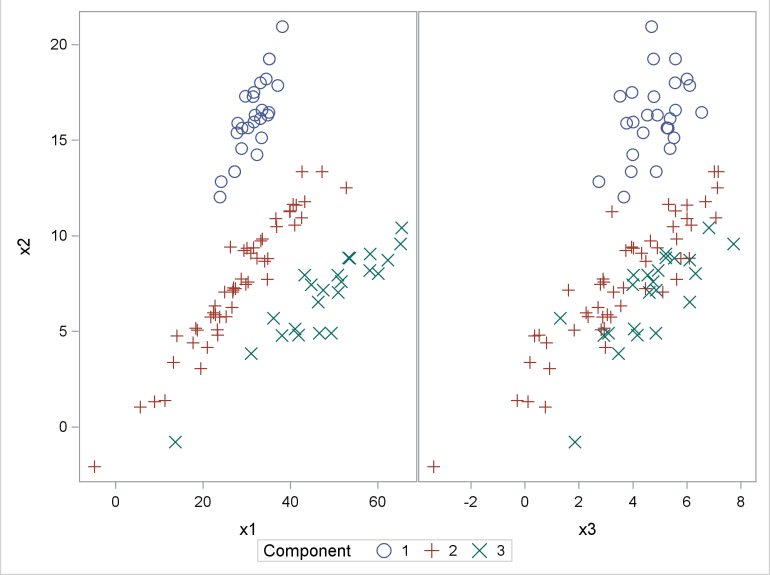
- Gets the number of components, k.
- Gets the vector of mixing probabilities, π. You can use the ListGetSubItem function to directly retrieve the mixing probability for each sublist.
- Uses the RandMultinomial function to draw a random observation from the multinomial distribution with parameters π. The resulting vector of counts specifies the number of observations, N_i, that should be drawn from the ith component.
- Simulates N_i random observations from each component.
- Returns the matrix that contains the random observations from the component densities.

```
start RandMVNMixture(NumObs, L);
/* Get vector of mixing probabilities */
k = ListLen(L);                     /* k = number of components              */
pi = j(1, k);                       /* vector of mixing probs                */
do i = 1 to k;
   pi[i] = ListGetSubItem(L, i||1); /* mixing probs: 1st item in i_th sublist */
end;
N = RandMultinomial(1, NumObs, pi); /* multinomial counts such as {26 51 23 } */
X = j(NumObs, k);                   /* allocate matrix for data              */
b = 1;                              /* beginning row                         */
do i = 1 to k;                      /* for each component                    */
   e = b + N[i] - 1;                /* ending row                            */
   mu  = ListGetSubItem(L, i||2);   /* mean vector: 2nd item in i_th sublist  */
   Cov = ListGetSubItem(L, i||3);   /* cov matrix: 3rd item in i_th sublist   */
   X[b:e, ] = RandNormal(N[i], mu, Cov); /* generate i_th MVN sample         */
   b = e + 1;                       /* next group starts at this row         */
end;
return X;
finish;

call randseed(12345);
X = RandMVNMixture(100, Mixture);
```

Figure 16 shows a panel of scatter plots for the simulated data. To illustrate the contributions of each component, each observation is displayed by using a marker that indicates the component from which it was generated. You can see that the clusters are well separated in the (x_1, x_2) projections, but that the data from the second and third components overlap in the (x_3, x_2) projection.

Figure 16 Simulated Clustered Trivariate Data



BINARY TREES

A *tree* contains nodes and directed edges. A tree starts with a root node. The root node is connected via branches to other nodes, called child nodes. Every node except the root node has exactly one parent node. A *binary tree* is a tree in which each parent node has at most two child nodes.

As shown in the previous section, SAS/IML lists can contain other lists. You can use this fact to emulate a binary search tree (BST). In a binary search tree, each node has a value (called a key), a link to a left child node, and a link to a right child node. Either or both child nodes might be null.

A binary search tree is useful when you want to search a list of items to see whether it contains a specified value. By starting at the root node, you can quickly determine whether the value is in the tree. If it is not, you can insert the value into the tree by modifying one of the null child nodes of an existing node. In a balanced binary tree that contains n items, the expected time to search the tree is $\mathcal{O}(\log(n))$.

The *SAS/IML User's Guide* shows how to use lists to emulate a binary search tree. The main idea is to represent a node by using a three-item list that contains a key value, a left node, and a right node. By setting the left or right node to be another three-item list, you can create a list of lists (to an arbitrary depth) that reflects the structure of a binary search tree.

The following example shows how to use the BST modules that are defined in the SAS/IML Sample Library. The %INCLUDE statement defines the BST modules, and the LOAD statement loads the definitions into a SAS/IML session. (You can specify the SOURCE option in the %INCLUDE statement to display the contents of the program file.) The BST is created from a vector that contains six unique values. Figure 17 shows a graphical representation of the tree.

```
/* L[1] is key value, L[2] is left child, L[3] is right child */
proc format;
    value BSTFmt  1='Key'  2='Left'  3='Right';
    value TorF    0='False' other='True';
run;
%include sampsrc(LstBST.sas);          /* define BST modules  */
proc iml;
load module = _all_;                   /* load BST modules    */
package load ListUtil;

x = {5 3 1 9 1 6 4}`;
bst = BSTCreate(x);                    /* initialize BST      */

title "Diagram of Binary Search Tree";
run BSTPlot(bst);                      /* plot the BST        */
run Struct(bst);                       /* summarize structure */
```

Figure 17 Visualization of a Binary Search Tree

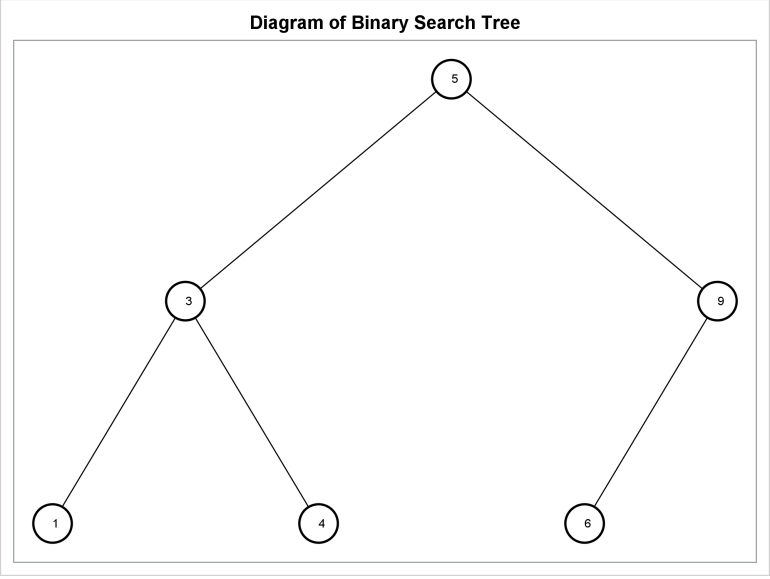


Figure 18 shows a summary of the BST as a list of lists. The Struct subroutine displays items of sublists, but not sublists of sublists. Consequently, although level-2 sublists are indicated in the output, the items in these branches are not displayed.

Figure 18 Structure of a List of Lists

Diagram of Binary Search Tree

bst									
Name	**Level**	**NRow**	**NCol**	**Type**	**Value1**	**Value2**	**Value3**	**Value4**	**More**
bst	0	.	3	List	bst[1]	bst[2]	bst[3]		
=> bst[1]	1	1	1	Num	5				
=> bst[2]	1	.	3	List	bst[2][1]	bst[2][2]	bst[2][3]		
=> => bst[2][1]	2	1	1	Num	3				
=> => bst[2][2]	2	.	3	List					
=> => bst[2][3]	2	.	3	List					
=> bst[3]	1	.	3	List	bst[3][1]	bst[3][2]	bst[3][3]		
=> => bst[3][1]	2	1	1	Num	9				
=> => bst[3][2]	2	.	3	List					
=> => bst[3][3]	2	0	0	Empty					

After you construct the tree, you can search the tree to see whether it contains specified values. In the following statements, the value 6 is found in the tree whereas the value 10 is not found.

```
targets = {6 10};                          /* look up key value */
do i = 1 to ncol(targets);
   Target = targets[i];
   Found = BSTLookup(Path, bst, Target); /* is value in tree? */
   print Target Found[format=TorF.] Path[format=BSTFmt.];
end;
```

Figure 19 shows whether each value is found and displays the sequence of paths that the search algorithm follows to determine whether the value is in the tree.

Figure 19 Search for Values in a Binary Search Tree

Target	Found	Path
6	True	Right Left

Target	Found	Path
10	False	Right Right

CONCLUSIONS

This paper demonstrates the new table and list data structures in SAS/IML 14.2.

A table is a rectangular data structure that contains both character and numeric data. Tables are convenient for passing mixed-type data to modules. The TablePrint subroutine provides enhanced capabilities for printing tables, such as support for formats, text alignment, and printing subsets of a table. The TablePrint subroutine also enables you to define and use a custom template to print a table.

A list is a general object that can contain matrices, tables, and other lists. A list is similar to a dynamic array in that it can grow or shrink as items are added or deleted. You can pass a list to an SAS/IML module, which enables you to pass parameters and data to an algorithm. Lists can be used to emulate well-known data structures such as structs, associative arrays, stacks, and binary trees.

REFERENCES

Smith, K. D. (2007). "PROC TEMPLATE Tables from Scratch." In *Proceedings of the SAS Global Forum 2007 Conference*. Cary, NC: SAS Institute Inc. http://www2.sas.com/proceedings/forum2007/221-2007.pdf.

Smith, K. D. (2013). *PROC TEMPLATE Made Easy: A Guide for SAS Users*. Cary, NC: SAS Institute Inc.

Wicklin, R. (2013). *Simulating Data with SAS*. Cary, NC: SAS Institute Inc.

APPENDIX: FREQUENTLY ASKED QUESTIONS ABOUT SAS/IML TABLES AND LISTS

Q: I have not yet upgraded to SAS/IML 14.2. Can I still use lists?

A: No. Tables and lists were introduced in SAS/IML 14.2. You cannot use these features in a previous release.

Q: Can I print a list? The PRINT statement doesn't seem to work on lists.

A: You can use modules in the ListUtil package, which is distributed with SAS/IML, to examine the contents of lists. As shown in this paper, the ListPrint module prints items in a list and the Struct module displays a summary of the list items. Neither module recursively prints all levels of a list; they display items in a list and items in a sublist.

Q: Lists in other languages support subscripts to assign items or to extract items from a list. Can I do that in SAS/IML?

A: Not yet, but soon! The SAS/IML developers are implementing a more succinct syntax for lists in a future release.

Q: Can I apply a function to every item of a list? For example, if every item in a list contains a vector, can I easily compute the mean of each item?

A: To apply a function to every item of a list, write a loop over the list items, extract each item, and compute the result. For the example of computing the mean of each item, see the example in the section "Operations on List Items."

CONTACT INFORMATION

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

Rick Wicklin
SAS Institute Inc.
SAS Campus Drive
Cary, NC 27513