List-Less Code: Energizing Your Frame Programs Using SCL Lists

David Bromberg, Prodigy Services Company

ABSTRACT

Screen Control Language (SCL) Lists are a critical element of SAS/AF® FRAME entries, objects, and widgets and can be used to enhance the flexibility and capabilities of SCL programs. Lists are data depositories that can contain an unrestricted mix of character and numeric data that can be extended to memory limit. Lists are declared, accessed, transformed, managed, and displayed via functions. Lists process rapidly because they are manipulated in RAM. Lists are critical to FRAMEs because they drive the display of objects. Understanding lists is important for capitalizing on the full capabilities of the SAS FRAME entry, the SAS/AF application environment, and OOP within SAS.

INTRODUCTION

In version 6.07, SAS/AF introduced (experimental release) FRAME entries, objects, and widgets as building blocks for Graphical User Interfaces (GUIs). FRAMEs and objects offered interactive, sophisticated, and flexible interfaces, especially when compared to PROGRAM entries. Object-oriented programming (OOP) had been brought to the SAS system.

The introduction of FRAMEs and objects also quietly heralded the introduction of lists. Yet SCL lists come packaged with an incredible set of features which read, build, transform, and protect its' elements. Clearly such a repertoire of capabilities suggests an importance that is not obvious on the surface.

In this paper we will therefore explore:

- the importance of lists as the "fuel" that drives the object "engines."
- features of SCL lists.
- strategies for using lists.

ENGINE

Objects

Object-oriented programming is often interpreted as the ability to localize, customize, and use list boxes, push buttons, and other reusable objects on screen. But objects are also defined by the processes or actions they can take. From a coders perspective, an object is code that branches, takes particular paths, or produces a result based on the requests made of that object.

For instance, a FRAME is an object that defines the GUI display area. The data required for a FRAME includes size of FRAME area, color, help facility, function key capabilities, or menu source. In fact, a compiled FRAME will reveal the following list (annotated; Figure 1) of named data values:

- KEYS="SASUSER.COMMON.STANDARD.KEYS"
- pmenu="SASUSER.COMMON.STANDARD.PMENU"
- help="SASUSER.HELP.MAINMENU.CBT"
- banner = 'NONE'
- vbar=0
- hbar=0
- srow=0
- scol=0
- nrow=0
- ncoll=0
- bordercolor=19
- bannercolor=18
- cmdcolor=21
- msgcolor=22
- backcolor=16
- forecolor=28
- borderattr=0
- bannerrattr=0
- FRAME =53

This list reveals elements (Items or instance variables) with values that reflect the FRAME's display on screen. When these values are processed by the FRAME generating code (methods), an instance of a FRAME is created.

GUI

Objects and FRAMEs are generated and their lists are populated without having to directly associate values to the instance variables. That is, we are not required to write something like:

%FRAME (keys=suser help=sasuser...)

So demystifying the SCL list and OOP requires recognizing that a readable interface (read GUI) provides access to and manipulation of a list by a user. An interface acts to error check the values.

Consider what happens when creating a new FRAME. We open a catalog via the Program Management Window, select File/NewEntry from the pull-down menu, and select FRAME as the entry type. We provide an acceptable SAS name for the entry and select OK. At that moment, an SCL list is created where all the FRAME instance variables are given default values, the FRAME generating methods are run, and a FRAME appears on screen.

Next, the FRAME'S General Attributes guides us through features for the FRAME. Interfaces appear prompting us for decisions on FRAME size, appearance, menus, locations, etc. Once OK is pressed, the FRAME methods are re-run and the FRAME is updated with features as designated by the values that now populate the list.

Therein lies the power of lists and SCL. Data values drive the code and processes towards a result. Put a face on the data values to make them accessible to the user, include error checking, and the fuel (list) and engine (object) are in place to generate the desired outcome.
FEATURES

Now that a context for the SCL list is established, the discussion turns to the features that contribute to their power and flexibility. Discussions of list features will include function calls that are applied against a list in support of that feature. In general, the syntax for applying a function to a list is:

\[
\text{returned value} = \text{function}(\text{list}, \text{options})
\]

The returned value can be a success/failure return code, a list identifier, a value, a name, a position in the list, or information about the list.

Declaration

As discussed previously, lists associated with FRAMES, objects, and widgets are defined when an object is instantiated. User defined lists are declared explicitly during the SCL program via a makelist or makenlist (make named list) function:

\[
\text{listid} = \text{makelist}();
\]

There are no restrictions on the number of items that can appear in a list (outside of memory constraints).

The makelist function sets aside a memory location in RAM at which data can be inserted. This memory location is referred to via a number called a list identifier (list id).

At the bottom of Figure 2, a [53] is displayed. That [53] is the list id for the FRAME’s list. Note that a list id for the same object can be different in a different SCL application session. For this reason, list ids are assigned to an SCL variable rather than being referred to by the numeric value.

To eliminate a list, use:

\[
\text{rc} = \text{dellist}([\text{listid}]);
\]

Data

A list is a holder of data values (items). Items can be character or numeric. An item that was previously character can be made numeric and vice versa. Items can be up to 200 characters in length.

Each item can optionally be associated with a 200 character name or label (which is always converted to uppercase). Names do not have to be unique.

Figure 2: A List with Named Elements

<table>
<thead>
<tr>
<th>Name</th>
<th>Region</th>
<th>Business</th>
<th>Year</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data</td>
<td>125</td>
<td>East</td>
<td>1995</td>
</tr>
<tr>
<td></td>
<td>Coast</td>
<td>Coast</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Shipping</td>
<td>Shipping</td>
<td></td>
</tr>
</tbody>
</table>

Lists hold one special type of numeric data - another list id. By carrying ids of other lists, lists can refer to the data from other lists.

Memory limits determine the amount of information a list can hold. Data is inserted with functions such as insert1, setitem1, and getitem1. The setitem* functions differ from the insert* and setitem* functions by virtue of the naming of items that are being inserted into the list.

Sources of data for a list include: manual addition via the above insert functions, SAS data sets, catalog entries, or text files. Data sets are placed into a list using the havedata function. Varied selects unique members of a character variable for the list. Catalog entries (such as SOURCE, OUTPUT, and LOG files: SLiST entries) and text files are put into lists with the fillist function.

Version 6.11 will include a describe function which lists an SCL list with data set, view, or catalog information on variable names, labels, formats, etc.

Access

Inserting or retrieving data in a list requires knowing an item’s type (character, numeric, or list). If the type is not known, it can be determined using the itemtype function. Note that using an incorrect type during access will cause an error condition.

Items can be added or replaced at any point in the list (unless that list is protected from modifications) by referring to a list position (index). An index can be a positive number, which indicates starting from the front of the list, or a negative number, which counts from the back of the list to determine position. Inserting an item causes all higher ordered index items to be pushed further down the list (index values are increased by one).

As previously discussed, insert* functions insert an item where the index indicates. The setitem* and setnitem* functions only perform inserts if the index points to the end of the list (-1) or if a value greater than the current list size. In this second case, a list is padded with missing data elements and an item is inserted at the requested point. When the index value of a setitem* or setnitem* function refers to a previously populated item position, that item is replaced rather than an insert being conducted.

Setitem* functions can perform their activity without referring to an index. The ‘n’ in setnitem* refers to the ability to insert or replace information on the basis of a named item search. If the named item is found, a replacement of value occurs. If the named item is not found, a named item is added to the end of the list.

Retrieval of information has a parallel set of functions in the getitem* and getnitem* functions. The getitem* function retrieves information by index position. If the index position of an item is unknown, the search* functions can be used to return the index of a numeric, character, or list value.

Getitem* functions return values by searching through the named items for the requested name. When a match is made, a value is returned.

\[1\] The * is a place holder in the function call for one of the following: N (Numeric), C (Character), or L (List) which designates the type of the item that is being acted upon.

128
One set of functions, `pop*`, deletes items as it returns data from them. `Pop*` causes a list to work like a stack by retrieving an item based on the index request, then deleting the item retrieved. All remaining items then move up (index value is decreased by one).

To process through a list sequentially without going beyond the upper bound of list items, use the `lslist` function to determine the number of items in the list. The returned value can then be used as an upper limit for the list index to avoid system errors that occur when that upper bound is exceeded.

Transformation
There are functions which rehull the order of items in a list. For instance, the `sortlist` function orders a list while the `revlist` function reverses the order of items in a list. `Rotlist` moves items from the front of the list to the back or vice versa.

Management
Another set of functions works against entire lists to copy them (copylist), blank out the list leaving a list id but no items (clearlist) or remove all remnants of the list from RAM (delist). Delist is important for releasing occupied RAM space during the application session.

List contents can be saved to a catalog entry using the `savellst` function. Contents can then be recalled from a catalog using `fillist`. User selected preferences can be stored in a list and saved so they can be passed from session to session.

Attributes
Attributes can be assigned to lists or their items which provide control over data features, access modes, and update or deletion availability.

For the list, attributes can be applied which:
- prevent copying, updating, or deleting
- restricts names to SAS names and items to character only data or numeric only data
- disallows the use of duplicate names
- defines the number of items in the list

Individual items can also have attributes applied to them. By including an index value in the function call, an item is the target of the attribute feature. Without an index, the attribute is applied to the list.

For items, attributes can be used to:
- prevent updating or deleting
- restrict the item to character or numeric data
- declare an item to be inactive, i.e., not to be displayed in popmenu (see Display later)
- prevent items from being written to a catalog SUSL when a savelst is conducted.

To apply an attribute to a list or item, use the `setattr` function. The `getattr` function is used to retrieve the current set of attributes. It returns a string of all currently active attributes of the list. With the `hasattr` function, the user can determine if a particular attribute is in force. For `hasattr`, a return code of 1 indicates that the requested attribute is indeed an element of a list.

Note that many of the discussed attributes must be activated via the SCL functions because they are paired with a default alternative (e.g., no updates allowed) which has "turned off" the feature.

Display
Dumping the contents of a list for review is done with the `putlist` function. This function is essential during the development process.

Character items in a list can be converted into an on screen menu by using the `popmenulist` function. When the popmenu list comes up, the character items in the list are displayed as menu selections. Selections of an item in the menu return that item to an SCL variable. An appropriate process can then be spawned based on the selection.

Environment
With the opening of a SAS application via the AF command, an "application environment" is created in RAM which contains all the events and outcomes of activities that occur during execution of the application. All lists that are created are therefore associated with an "application environment" and are said to be "local" to that environment.

Each time an application branches to a window via a call command (e.g., display, fscedi, fscbrowse), the application is still conducting processes within the local environment. Any lists created to that point still reside within the local environment (unless they have been deleted). So list data is in fact shared data for all FRAME and call events that occur within the application or local environment.

Despite the existence of these lists within the local environment, without knowing the list id for a list, they are unusable. List ids must therefore have a methodology for communicating their presence.

Not surprisingly, each application environment has a list associated with it. This is a "local environment list" and is automatically created when an application is opened. The list identifier for the local list can be determined by the following command:

`lenviist = env/islstr('L');`

By inserting the list id of any other list into the local environment list, that list id (and therefore the list) is made available to all other FRAMES initiated by the application.

There is one list that does get automatically nested into the local environmental list. At invocation, a named list called the `_CMDBLIST_` is generated. This list is comprised of the libname, catalog, entry, and type of the catalog member that was called to initiate the application.

One other environmental list exists that is of interest. Each SAS session creates an environment in RAM. A SAS application occupies a piece of the SAS session environment. A SAS session is therefore considered to be global to the SAS application environment. The global environment also has a list associated with it. That list is revealed using the following:

`genviist = env/islstr('G');`

There are no lists automatically nested into the global list. Nested lists within the global list must be explicitly inserted. Once they are inserted however, that list is defined to the entire SAS session.
Applications Development

Even if the originating application is shut down, provided the SAS session is not ended, that list can be used within another application session.

However, in order to take advantage of this feature, the list to be nested must have been declared as a global list as follows:

\[ \text{listid} = \text{makelist}(0, 'G') \]

So lists have the capability to communicate across FRAMEs and applications.

Miscellaneous

Lists are created at runtime and exist in RAM. Since they are RAM based, response time in list manipulation occurs at CPU clock rates. List manipulation therefore, is an extremely fast and efficient process.

SCL functions which invoke selection list boxes (i.e., calllist, colorlist, detaillist, datalist, devlist, divlist, fieldlist, list, sheet, showlist, and varlist) return selected value(s) to an SCL variable (these functions were released into use prior to the list capable versions of SAS/Ap). However, if a list is declared as current using:

\[ \text{rc} = \text{curlist}(\text{listid}) \]

prior to the call to the selection list box, then all selected values will be returned into both the SCL variable and the list that was just made current.

If there exists one shortcoming in list usage, it is the inability to pass list data into a submit block. There is no equivalent to the SCL variable Replace statement for lists. Therefore, list items must be assigned to SAS variables if those items are to appear in a submit block.

STRATEGIES

AF Invocation and Lists

Invoking a SAS application is done with an af command. As mentioned, the af command creates a local application environment and a _CMDLIST_ list which is nested inside the local environment list.

Parameters can be used on the af command line to pass data or named data into the _CMDLIST_ list. This feature can be useful when trying to debug a production program that relies heavily on lists.

This opportunity arises due to a development phase requirement to constantly trace list values. Rather than eliminating the printlist in the code, they can be placed into Boolean expressions. The expression will only resolve to true when a debugging parameter in the _CMDLIST_ is set to YES.

The code that capitalizes on this scenario is as follows. Consider the following statement:

\[ \text{af callib=mylib.mycat.aframe.frame showlist='YES'} \]

This generates the following local environmental list:

### Figure 3: A Local Environment List

| _CMDLIST_ = (LIBNAME='MYLIB' CATALOG='MYCAT' NAME='AFRAME' TYPE='FRAME' SHOWLIST='YES') [3] |
| _CMD_ = 'AF' [1] |

Within each FRAME (or within a method called by the FRAME) is the following code:

### Figure 4: Using a _CMDLIST_ Parameter

\[ \text{lev} = \text{envlist}(\text{Lev}) \]

\[ \text{cmdlist} = \text{getniteml}(\text{lev}, '_\text{CMDLIST}_') \]

if cmdlist gt 0 then

\[ \text{showlist} = \text{getnitemc}(\text{cmdlist}, 'SHOWLlST', 1.1, ') \]

if upcase(showlist) = 'YES' then call putlist(listid);

The code searches for the SHOWLIST named item to see if the value it carries is YES.

In the event a problem should arise in production level code, the application can now be invoked with the showlist parameter set to YES. List information will be printed to the log as the session is conducted. Lists can be scanned as the program runs to pinpoint the problem more quickly.

Passing parameters into the application session through the af command line can be used effectively to set up session defining parameters.

Communication Using Lists

Application development requires giving the user choices. This may be as simple as offering control over fonts and colors of screens to complex tasks of choosing parameters which subset analysis fields or data elements that are displayed on screen. Whatever the particular task, the procedure often requires the user to make selections in one FRAME and having those selections impact the display or process within another FRAME.

Using call display is one possibility. However, as the number of values passed grows, data transfer becomes more complex and unwieldy. Call display also communicates with only one Frame at a time. Lists organize data efficiently and can communicate with all FRAMEs at the same time.

Macros don’t quite do the job either. Lists maintain the numeric and character distinctions of the data rather than converting data to strings. Lists can also be removed from memory at any time.

Based on these advantages, lists should be the tool of choice. Sample code for passing lists among FRAMEs follows. Assuming the list to be passed (parmlst) is already created, the first step is to nest the target list into a local list.
Fig 5: Populating a Local Environment List

```plaintext
lenv = envlist('L');
rc = setnitem(lenv,parmlist,'PARMLIST');
call display('test.frame');
```

If inlist is populated with a value other than 0 (last value in the function call which is a default value to return if the named item is not found), a process is run to access information the FRAME requires.

Information is thereby easily and effectively passed between FRAMES by virtue of lists.

**Submitting Lists**

Data in a list is often needed for processing within a submit block. Unfortunately there is no direct methodology for moving data from a list into a submit block such as the SCL Replace command. Therefore list data must pass through an SCL variable before it can appear in a submit block.

Consider the above limit within the following system: A system is designed as a front end to a number of standardized reports. Each report is keyword parameter driven. Reports can have varying numbers and types of parameters.

The following SCL code simulates the above application with two lists being populated.

Fig 7: Populating Reporting Lists

```plaintext
printds = makelistO;
printds = setnitemc(printds,'sasuser','libname');
printds = setnitemc(printds,'test','dataset');
printds = setnitemc(printds,',','varorder');
rprtlist = makelistO;
rc = setnitemc(rprtlist,printds,'PRINTDS');
```

The obvious answer is to assign each list item to an SCL variable and pass those into the submit block. But our reports and their parameters could become increasingly sophisticated, with more and more parameters contributing to a given report.

The number of SCL variables could be manipulated but the tradeoff here is increased maintenance (code must be enhanced when the number of report parameters exceeds the number of designated SCL variables), increased processing time for the entire SCL program, and more variables to track during development.

Instead, the following code will manage the issue using only three SCL variables regardless of the number of parameters:

Fig 8: Code for Submitting SCL Lists

```plaintext
1. replace report '+&report('; 2. replace delim '+&delim'; 3. do i = 1 to listlen(rprtlist); 4. report = nameitem(rprtlist,i); 5. submit; 6. &report 7. endsubmit; 8. parmlist = getiteml(rprtlist,i); 9. do j = 1 to listlen(parmlist); 10. parm = nameitem(parmlist,j); 11. val = getitemc(parmlist,j); 12. if j ne 1 then delim = ';'; 13. submit; 14. &delim &parm = &val 15. endsubmit; 16. if j=listlen(parmlist)then do; 17. submit continue; 18. ); 19. endsubmit; 20. end; 21. end; 22. end;
```

Line by line, the following events occur in this code:

1. The macro call is created with the report name.
2. Variable delim and the Replace will provide the comma delimiter between parameters.
3. Since more than one report can exist in rprtlist, be sure to cycle through all the reports.
4. Retrieve report name and return to variable report.
5. Submit and hold in the preview buffer.
6. Replace SCL variable report with macro call.
7. Return to SCL program.
8. Retrieve the list for this report's parameters.
9. Cycle through all the parameters in this list.
10. Retrieve item name and return to variable parm.
11. Retrieve the data and return to variable var.
12. Assign comma to delim for all iterations through the parameters except the first (no comma between the macro call and the first parameter).
13. Submit and hold in the preview buffer.
14. Replace SCL variable delim with a comma or space. Replace SCL variables parm and val with their respective values.
15. Return to SCL program.
16. Execute steps to follow after last item in the list.
17. Submit all code in preview buffer upon endsubmit.
18. Close the macro call with ‘});’.
19. Submit preview buffer.
20. End of conditionals and continue loops.

The code as submitted from the preview buffer is:

**Figure 9: Submitted SCL Code**

```sas
%PRINTDS(
  LIBNAME = sasuser,
  DATASET = test,
  VARORDER =
);```

This code is maintenance free and driven by list length.

**SUMMARY**

The power and flexibility of lists is attributable to a combination of the dynamic manner in which data is stored and the functionality that can be applied against that data. Lists also offer a powerful medium for communication between applications, objects, methods, and SCL programs.

Understanding lists offers insight into the functioning of FRAMEs and objects. By understanding lists, one can begin to understand the machinations of the SAS object-oriented programming paradigm.

**REFERENCES**


**ACKNOWLEDGEMENTS**

My thanks to Glen R. Walker of SAS Institute for reviewing portions of this document.

For more information contact:

David Bromberg
Prodigy Services Company
445 Hamilton Ave
White Plains, NY 10601

Work Phone: (914)448-8617
Home Office: (914)939-4507
Internet: dbromberg@prodigy.com