ABSTRACT

The SCL list is a flexible and powerful data structure which first appeared in version 6.07 of the SAS® system. This tutorial will help you:

- understand what SCL lists are, and how they differ from other data structures
- learn list terminology
- categorize lists and items by access, type, and attributes
- manipulate lists using SCL functions and routines
- decide when your application might use lists

INTRODUCTION

An SCL list is an ordered collection of data in the Screen Control Language (SCL) environment. The elements of a list are called items.

SCL lists have the following features which distinguish them from arrays:

- mixed types - A single list may contain items of various types.
- dynamic sizing - The number of items in a list can change at run time.
- easy manipulation - Special SCL functions and routines facilitate operations on lists.
- permanent storage - Lists may be stored between sessions and reused.
- indexed and named access - List items may be accessed either by position or by name.
- data sharing - A list can be readily accessed within an application or a session by passing a single id number between entries.

Do not confuse SCL lists with selection lists. A selection list is a display window object, which disappears as soon as you make your choices. An SCL list is not usually displayed, but remains in memory and stores data. The contents of an SCL list may be used to populate a selection list, and items selected from a selection list can be used to fill an SCL list. The functions and routines used to manage SCL lists are distinct from those used to generate selection lists. In this tutorial, the term "list" means an SCL list.

The primary source of information for SCL lists is Reference 1, chapters 8 and 17-20. You can find additional information in Reference 2, chapter 4, and obtain on-line assistance by issuing the command HELP SCL.

We occasionally use different terminology than do the references, when we believe this makes the concepts easier to understand. We include the "official" terminology in parentheses.

ACCESS

List access

Lists are either active or stored.

Active lists reside in memory and may be used in an SCL program. When an active list is first created at run time, the SAS system assigns it a unique numeric identifier, called a listid.

Stored lists may reside in either catalog entries or external files. Only catalog entries of type SLIST store the complete structure and contents of any list. Other catalog entry types** and external files store only the values of character items. Lists are stored and restored using the SAVELIST and FILLIST functions.

Item access

You can access items by either index (position) or name.

Indexed access identifies an item by its absolute position within a list. The index may be either positive (counting from the beginning of the list) or negative (counting from the end of the list). For example:

<table>
<thead>
<tr>
<th>Pos. index</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Neg. index</td>
<td>-4</td>
<td>-3</td>
<td>-2</td>
<td>-1</td>
</tr>
<tr>
<td>Value</td>
<td>SASUSER</td>
<td>SUGI</td>
<td>DEMO</td>
<td>PROGRAM</td>
</tr>
</tbody>
</table>

Named access identifies an item by an optional name. By default, these names:

- can be up to 200 characters long
- do not have to be unique
- are automatically upcased
- contain no trailing blanks

For example:

<table>
<thead>
<tr>
<th>Pos. index</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Neg. index</td>
<td>-4</td>
<td>-3</td>
<td>-2</td>
<td>-1</td>
</tr>
<tr>
<td>Name</td>
<td>LIBRARY</td>
<td>CATALOG</td>
<td>ENTRYNAMES</td>
<td>ENTRYPThE</td>
</tr>
<tr>
<td>Value</td>
<td>SASUSER</td>
<td>SUGI</td>
<td>DEMO</td>
<td>PROGRAM</td>
</tr>
</tbody>
</table>

TYPES

List types

Lists can be grouped into four types, based on scope and owner.

The scope (or environment) of a list is either local or global. Local lists are available within the application** in which they are created, and are automatically deleted when that application ends. Global lists are available within the entire SAS session in which they are created, and are automatically deleted when the session ends.

The owner of a list is either the system or the user. The SAS system automatically creates and deletes one global list per session and one local list per application. System lists are also known as environment lists. At any point in an SCL program, the user may create and delete local and global lists, using (primarily) the MAKELIST and DELLIST functions.

Listid numeric values are a function of scope and type:

<table>
<thead>
<tr>
<th>Scope</th>
<th>Owner</th>
<th>system</th>
<th>user</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>local</td>
<td>1</td>
<td>34</td>
</tr>
<tr>
<td></td>
<td>global</td>
<td>2</td>
<td>4.68</td>
</tr>
</tbody>
</table>

Jeremy Stum, Paul Waldron Consulting Inc., West Nyack, NY
Resist the temptation to reference lists with numeric constants in your programs, for at least two reasons:

- List numbers are "recycled" after user-owned lists are deleted
- The above pattern may change in future releases of SAS software

Lists for user lists must be passed between entries using parameters, macro variables, or system lists. Lists for system lists can be determined by calling the ENVLIST function.

You may select any one active list, regardless of type, to serve as the current list. The purpose of this special list is to store values:

- Chosen by the user from selection lists
- Generated by the LVARLEVEL function

Applications start with no list designated as current. You assign and change the current list using the CURLIST function. You may even have no current list, by assigning 0 as the current list.

**Item types**

Items may be one of three types, depending on the values they store:

- Character
- Numeric
- List

Character and numeric items are similar to SCL variables. You do not declare character and numeric item lengths, however, since they change in size dynamically. Character values exceeding 200 bytes are truncated without warning.

List items, also known as sublists, are merely pointers to other lists. Since sublists are identified by their listid, these lists must already exist before they can become sublists. The sublist feature is very powerful, since:

- A list may be a sublist of multiple lists.
- Sublists may be nested without limit.
- Sublists may be recursive. For example, if A and B are two lists, B may be a sublist of A at the same time that A is a sublist of B. A may even be a sublist of itself. Needless to say, you must be very careful with recursive structures!

**ATTRIBUTES**

Both lists and items have attributes, which determine the operations which may legally be performed on them. A function call which violates these attributes will result in either a non-zero return code or termination of execution.

These attributes may be queried (GETLATTR, HASATTR) and modified (SETLATTR) at any time, and may be stored and restored using SAVELIST and FILLIST.

The following tables show these attributes. Arrows indicate default values.

### List attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEFAULT</td>
<td>All defaults</td>
</tr>
<tr>
<td>DELETE ⦿</td>
<td>Can the list be deleted?</td>
</tr>
<tr>
<td>NODELETE</td>
<td></td>
</tr>
<tr>
<td>UPDATE ⦿</td>
<td>Can the list be updated?</td>
</tr>
<tr>
<td>NOUPDATE</td>
<td></td>
</tr>
<tr>
<td>FIXEDLENGTH ⦿</td>
<td>Can the list length change?</td>
</tr>
<tr>
<td>NOFIXEDLENGTH</td>
<td></td>
</tr>
<tr>
<td>COPY ⦿</td>
<td>Can sublists be copied?</td>
</tr>
<tr>
<td>NOCOPY</td>
<td></td>
</tr>
<tr>
<td>ANYNAMES ⦿</td>
<td>Must all items have valid SAS names?</td>
</tr>
<tr>
<td>SASNAMES</td>
<td></td>
</tr>
<tr>
<td>DUPNAMES</td>
<td>Must all item names be unique?</td>
</tr>
<tr>
<td>MODUPNAMES</td>
<td></td>
</tr>
<tr>
<td>FIXEDTYPE ⦿</td>
<td>Can items within the list change type?</td>
</tr>
<tr>
<td>NOFIXEDTYPE</td>
<td></td>
</tr>
<tr>
<td>CHARONLY ⦿</td>
<td>Must all items be character?</td>
</tr>
<tr>
<td>NOCHARONLY</td>
<td></td>
</tr>
<tr>
<td>NUMONLY ⦿</td>
<td>Must all items be numeric?</td>
</tr>
<tr>
<td>NONUMONLY</td>
<td></td>
</tr>
</tbody>
</table>

### Item attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEFAULT</td>
<td>All defaults</td>
</tr>
<tr>
<td>DELETE ⦿</td>
<td>Can the item be deleted?</td>
</tr>
<tr>
<td>NODELETE</td>
<td></td>
</tr>
<tr>
<td>UPDATE ⦿</td>
<td>Can the item be updated?</td>
</tr>
<tr>
<td>NOUPDATE</td>
<td></td>
</tr>
<tr>
<td>FIXEDTYPE ⦿</td>
<td>Can item change type?</td>
</tr>
<tr>
<td>NOFIXEDTYPE</td>
<td></td>
</tr>
<tr>
<td>ACTIVE ⦿</td>
<td>Is the item available for selection in a POPMENU?</td>
</tr>
<tr>
<td>INACTIVE</td>
<td></td>
</tr>
<tr>
<td>WRITE ⦿</td>
<td>Should the item be included when the list is stored?</td>
</tr>
<tr>
<td>NOWRITE</td>
<td></td>
</tr>
</tbody>
</table>

**SCL LANGUAGE ELEMENTS**

**General information**

Most of the SCL language elements which manage lists are functions. We have grouped these elements into categories, to help you learn them. When an element seemed to fit multiple categories, we chose the category based on our estimate of most common usage.

Several functions come in families of three. They end with C, L, or N depending on whether they refer to character, list, or numeric items. You may find that you need to use the ITEMTYPE function first to determine the correct family member to use.

Since References 1 and 2 cover syntax comprehensively, we have chosen to focus, instead on:

- Concepts
- Similarities and differences
- Subtle details
- Examples
The examples in this section refer to the following lists and data set:

- **list1** (listid=5)
  - Position 1 2 3 4 5
  - Name LAST NAME FIRST NAME SALARY HISTORY EMP # AGE
  - Type C C L N N
  - Value SMITH John $7 1111

- **list2** (listid=7)
  - Position 1 2 3
  - Name GOFER FACILITATOR VP
  - Type N N N
  - Value 10,000 50,000 200,000

- **list3** (listid=9)
  - Position 1 2 3
  - Name LAST NAME NAME HISTORY
  - Type C C L N
  - Value Valuo SMITH John +7

- **list4** (listid=11)
  - POSITION 1 2
  - NAME ENGINE LABEL
  - TYPE
  - VALUE

- **list5** (listid=13)
  - Position 1 2 3 4
  - Name TYPE
  - Value Value Walk Run Run Walk

- **WORKLISTS:**
  - DATE 01MARS LON 198
  - DATE 01MARS FRA 207
  - DATE 01MARS LON 265
  - DATE 01MARS PAR 138
  - DATE 02MARS LON 147
  - DATE 02MARS FRA 176
  - DATE 02MARS LON 201
  - DATE 02MARS PAR 172
  - DATE 03MARS LON 197
  - DATE 03MARS FRA 180
  - DATE 03MARS LON 151
  - DATE 03MARS PAR 147

**Creating lists**

MAKELIST creates a user list, returning a listld. You can specify the scope of the list and the initial number of items (which may be zero). Any initial items are numeric missing value and unnamed. For example:

```c
new1 = MAKELIST(); /* listid=5, local, 0 items */
new2 = MAKELIST([6],[G]); /* listid=4, global, 5 numeric items with missing values */
```

MAKELIST is similar to MAKELIST, except that you provide names for all initial items, of which there must be at least one.

COPYLIST copies the contents of a list into a new or existing list, returning a listid.

```c
new = COPYLIST(list1); /* new is an exact copy of list1, with respect to both attributes and contents. The only difference is the listid. */
append = COPYLIST(list3,'N',list1); /* append has eight items, with Bill Jones' data following John Smith's. append has the same listid as list1. Names are not unique. */
merge = COPYLIST(list3,'N MERGE',list1); /* merge has five items, with Bill Jones' data overlaying John Smith's by name, as if list3 contained transactions. merge has the same listid as list1. */
```

List attributes, types, and length

HASATTR reports whether a list has a particular attribute, and returns a boolean value. For example:

```c
rc = HASATTR(list1,'DELETE');
```

GETLATTR returns a character string containing all the attributes of a list. For example:

```c
attrs = GETLATTR(list1);
```

If list1 contains default attributes, then the value of attrs is the string:

```c
DELETE UPDATE NOFIXEDTYPE NOFIXEDLENGTH ANYNAMES UPPNAMES NOCHARONLY NONUMONLY COPY
```

SETLATTR changes the attributes of a list, and returns a code indicating success or failure. For example:

```c
rc = SETLATTR(list1,'FIXEDLENGTH CHARONLY');
```

changes the attributes of list1 so that all items must be character, and items may not be added or deleted.

CURLIST returns the listid of the current list, and allows you to designate a different list as current. The current list is automatically filled by selection list functions.'

```c
current = CURLIST(); /* identifies the current list. */
current = CURLIST(new); /* identifies the list which used to be current, and makes new the current list. */
```

The current list reverts to 0 (null list) when the application ends.

ENVLIST identifies a system (environment) list, returning a listid. For example:

```c
global = ENVLIST('G'); /* the value returned to global is 2. */
```

LISTLEN reports the number of items in a list, returning an integer. It does not count sublist items. For example:

```c
len = LISTLEN(list1); /* the value returned to len is 5. */
```

Creating items

The INSERT family creates a new item by specifying an insertion position, returning a listid. Insertion "bumps" items to the right to make room. For example:

```c
new1 = INSERT([6],[B]); /* listid=6, global, 5 numeric items with missing values */
```

```c
new2 = INSERT([6],[K],[3]); /* listid=6, global, 5 numeric items with missing values */
```
LVARLEVEL fills a list with the unique, formatted values of a variable from a SAS data set, returning a code indicating success or failure. If the list is not specified, it defaults to the current list. The third argument must be a numeric variable, initialized to 0, in which LVARLEVEL will place the number of values found. For example, if DEST in WORK.FLIGHTS has been formatted and the following code is executed:

```plaintext
cities = CURLIST();
vals = 8;
rc = LVARLEVEL(ds1,'DEST',vals);
```

then cities is:

<table>
<thead>
<tr>
<th>POSITION</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>NAME</td>
<td>C</td>
<td>C</td>
<td>C</td>
</tr>
<tr>
<td>VALUE</td>
<td>Frankfurt</td>
<td>London</td>
<td>Paris</td>
</tr>
</tbody>
</table>

and the value of vals is 3.

Item attributes and types

HASATTR reports whether an item has a particular attribute, returning a Boolean value. For example:

```plaintext
rc = HASATTR(list1,'DELETE',2);
```

GETLATTR returns a character string containing all the attributes of an item. For example:

```plaintext
atts = GETLATTR(list1,3);
```

If item 3 contains default attributes, then the value of atts is the string:

```
ACTIVE WRITE NOAUTO NOEDIT DELETE UPDATE NOFIXEDTYPE
```

SETLATTR changes the attributes of an item, returning a code indicating success or failure. For example:

```plaintext
rc = SETLATTR(list3,'INACTIVE',4);
```

changes the attributes of item 4 so that a user may not select it if it is displayed by POPMENU.

ITEMTYPE determines the type of an item, returning a 1-character value: C, L, or N. For example:

```plaintext
type = ITEMTYPE(list1,3);
```

The value returned to type is 'L'. '/'

Changing item order

REVLIST reverses the order of items in a list, and returns a listid. For example:

```plaintext
list1 = REVLIST(list1);
```

ROLIST rotates the items in a list, and returns a listid. For example:

```plaintext
list1 = ROLIST(list1);
```

<table>
<thead>
<tr>
<th>POSITION</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>NAME</td>
<td>FIRST</td>
<td>NAME</td>
<td>SALARY</td>
<td>HISTORY</td>
<td>EMPLOYEE</td>
</tr>
<tr>
<td>VALUE</td>
<td>John</td>
<td>7</td>
<td>30</td>
<td>1111</td>
<td>15</td>
</tr>
</tbody>
</table>

SORTLIST sorts the items in a list, and returns a listid. You can sort the entire list or only part of the list, by value or name, ascending or descending, case-sensitive or case-insensitive, keeping or deleting duplicate items. For example, if cities is:

<table>
<thead>
<tr>
<th>POSITION</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>NAME</td>
<td>LAST</td>
<td>NAME</td>
</tr>
<tr>
<td>TYPE</td>
<td>C</td>
<td>C</td>
</tr>
<tr>
<td>VALUE</td>
<td>Smith</td>
<td>John</td>
</tr>
</tbody>
</table>

then

```plaintext
cities = SORTLIST(cities, 'DESCENDING');
```

Changing item values

DESCRIBE fills a list with descriptive information about a SAS file, returning a code indicating success or failure. The list must already contain items with names corresponding to file attributes. For example:

```plaintext
rc = DESCRIBE('WORK.FLIGHTS', list4);
```

<table>
<thead>
<tr>
<th>POSITION</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>NAME</td>
<td>'ENGINE'</td>
<td>'LABEL'</td>
</tr>
<tr>
<td>TYPE</td>
<td>C</td>
<td>C</td>
</tr>
<tr>
<td>VALUE</td>
<td>V608</td>
<td>'International Airlines'</td>
</tr>
</tbody>
</table>

The SETITEM family assigns a value to a new or existing item specified by position, returning a listid. For example:

```plaintext
list1 = SETITEM(list1, 2, 3, 4, 5, 6);
```

<table>
<thead>
<tr>
<th>POSITION</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>NAME</td>
<td>LAST</td>
<td>NAME</td>
<td>SALARY</td>
<td>HISTORY</td>
<td>EMPLOYEE</td>
</tr>
<tr>
<td>TYPE</td>
<td>C</td>
<td>C</td>
<td>L</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>VALUE</td>
<td>Smith</td>
<td>John</td>
<td>15</td>
<td>1111</td>
<td>30</td>
</tr>
</tbody>
</table>

The SETNITEM family assigns a value to a new or existing item specified by name, returning a listid. For example:

```plaintext
list1 = SETNITEM(list1, newlist, 'SALARY HISTORY');
```

If the value of newlist is 15, then list1 would be:

<table>
<thead>
<tr>
<th>POSITION</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>NAME</td>
<td>LAST</td>
<td>NAME</td>
<td>SALARY</td>
<td>HISTORY</td>
<td>EMPLOYEE</td>
</tr>
<tr>
<td>TYPE</td>
<td>C</td>
<td>C</td>
<td>L</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>VALUE</td>
<td>Smith</td>
<td>John</td>
<td>15</td>
<td>1111</td>
<td>30</td>
</tr>
</tbody>
</table>
Searching lists

Note: None of the following functions searches sublists.

NAMEDITEM searches for an item with a specified name, returning the position of the first instance found. For example:

pos = NAMEDITEM(list1,'AGE');
/* assigns the value pos to pos. */

NAMETEMP searches for an item at a specified position, returning, and optionally changing, its name. For example:

name = NAMETEMP(list1,1);
/* assigns the value 'SALARY HISTORY' to name. */
name = NAMETEMP(list1,1,'RAISES');
/* assigns the value 'SALARY HISTORY' to name, and changes the name of the item to 'RAISES'. */

The GETITEM family searches for an item at a specified position, returning its value. For example:

sublist = GETITEM(list1,3);
/* assigns the value 7 to sublist. */

The GETNITEM family searches for an item with a specified name, returning the value of the first instance found. For example:

sublist = GETNITEM(list1,'SALARY HISTORY');
/* assigns the value 7 to sublist. */

The SEARCH family searches for an item with a specified value, returning the position of the first instance found. For example:

pos = SEARCH(list1,'Smith');
/* assigns the value 1 to pos. */

Here is a table summarizing the search functions:

<table>
<thead>
<tr>
<th>Function</th>
<th>Position</th>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>You specify</td>
<td>pos = POPMENUP(list5); choice = GETITEM(list5,pos);</td>
<td>NAMEDITEM</td>
<td>GETITEM</td>
</tr>
<tr>
<td>Name</td>
<td>GETITEM</td>
<td>SEARCH</td>
<td></td>
</tr>
<tr>
<td>Value</td>
<td>SEARCH</td>
<td>SEARCH</td>
<td></td>
</tr>
</tbody>
</table>

Note that, although there is no single function which returns a name from a specified value, you can perform this task by passing the result of SEARCH to NAMETEMP.

Displaying list contents

POPMENU displays a selection list containing the values of all items in a list, returning the position of the selected item. The selection list includes scroll bars if its length exceeds the screen size. For example:

pos = POPMENUP(list5);
choice = GETITEM(list5,pos);
displays a selection list of four items:

<table>
<thead>
<tr>
<th>Walk</th>
<th>Run</th>
<th>Run</th>
<th>Walk</th>
</tr>
</thead>
</table>

and captures the user's selection in choice.

POPMENU has several limitations. It:
- allows only one selection
- will fail if list contains numeric or sublist items
- has no option to display names
- displays all, not just unique, values
- does not return value directly

The PUTLIST routine displays names, values, positions, and listids for a specified list and its sublists, in the MESSAGE window. It can be very helpful for debugging list program logic. The indentation option, although requiring more room, is easier to read. For example:

CALL PUTLIST(list1,'LIST1: ',1);

Storing and restoring lists

SAVELIST stores a list in a catalog entry or external file, returning a code indicating success or failure. If you save an SLIST catalog entry, the structure and contents of the list are preserved, and can be restored using FILLIST. If you save anywhere else:
- only character values are saved
- SAVELIST will fail if there are any numeric or sublist items

For example, if you want to store list1 so that both contents and structure can be restored:

rc = SAVELIST('CATALOG',
'SASUSER.SUGLEMP.SLIST',list1);

FILLIST can populate an active list with any of the following:
- information from a SAS catalog entry
- information from an external file
- icon numbers
- catalog names in the search path

Returning a code indicating success or failure. For example, if you saved list2 to a catalog entry named SASUSER.SUGLEMP.SLIST, then you can restore it later:

new = MAKELIST();
rc = FILLIST('CATALOG',
'SASUSER.SUGLEMP.SLIST',new);

Deleting Lists

DELIST deletes a list, optionally including its sublists, returning a code indicating success or failure. For example, to delete list1, but not its sublists:

rc = DELIST(list1);

Deleting Items

CLEARLIST removes all items from a list, returning a code indicating success or failure. For example, to delete list1:

rc = CLEARLIST(list1);

DELETITEM deletes an item by position, returning a listid. For example:

list1 = DELITEM(list1,1);

DELTITEM deletes an item by name, returning a listid. For example:

list1 = DELITEM(list1,'SALARY HISTORY');
The POP family removes an item from a list by position, returning the value of the deleted item. Items at higher positions automatically drop down to the next lower position. For example:

\[ \text{value} = \text{POPC(list, 2)}; \]

places the value 'John' in value and transforms list1:

<table>
<thead>
<tr>
<th>Position</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>LAST NAME</td>
<td>SALARY HISTORY</td>
<td>EMPLOYEE ID</td>
<td>AGE</td>
</tr>
<tr>
<td>Type</td>
<td>C</td>
<td>L</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>Value</td>
<td>SMITH</td>
<td>9</td>
<td>1111</td>
<td>26</td>
</tr>
</tbody>
</table>

**USAGE**

What are some of the factors which might lead you to use lists instead of arrays, data sets, or macro variables?

- You are already in an SCL environment.
- You need to read or write data frequently. (Active lists are stored in memory, so I/O is rapid.)
- Your data are inherently hierarchical in nature. (Sublist structures are ideal for representing hierarchies.)
- You want to store data across sessions.
- You like the idea of referencing items using long names, so that you do not need to keep track of positions.
- You want to store data of mixed type in one structure.
- You need to reorder your data frequently.

**BACKGROUND EXECUTION**

Suppose you have an application for which you would like to use lists, but you need to run that application in a batch or background mode? If the application does not require interaction with a user, then you can store the program in entries of type SCL, which have no display window.

But how do you invoke the application with no command line from which to issue an AF command? There are two possible solutions:

- There is a BATCH option on PROC DISPLAY. For example:
  
  \[ \text{PROC DISPLAY C='library.catalog.entryname.SCL'}; \]

  \[ \text{BATCH;} \]

  \[ \text{RUN;} \]

- There is a v. 6.09 system option called DMSBATCH, which allows you to include Display Manager statements in your program, as long as they have no interactive requirements. For example:
  
  \[ \text{OPTIONS DMSBATCH;} \]

  \[ \text{DM 'AF C='library.catalog.entryname.SCL args';} \]

**POSSIBLE ENHANCEMENTS**

Several improvements to list functions suggest themselves:

- The search functions should be optionally able to examine sublists, and to find multiple occurrences by name or value.
- POP/PUSHP/MENU needs to be enhanced to overcome the limitations listed earlier.
- LVARLEVEL should have an option to fill a list with unique unformatted, as well as formatted, values.
- Whenever possible, families of functions should be replaced by single functions, which would determine item type automatically.

**SUMMARY**

SCL lists open new horizons in data storage and manipulation. Many tasks can be accomplished with less effort and greater flexibility than ever before in the SAS system. We hope that this tutorial will help you learn list technology quickly and painlessly.

**REFERENCES**


To contact the primary author, write to:

Roger Staum
SAS Institute Inc.
520 MADison Avenue
New York, NY 10022