ArrayList XFCN: A Free HyperCard Utility

By Ari Halberstadt

ABSTRACT

An external function implementation of a general purpose list data structure for HyperCard (on the Macintosh). The lists are manipulated as arrays, and basic operations are provided on the arrays. Operations implemented include insert, delete, search, sort, stack operations, and some set operations. ArrayList was written to provide a fairly fast way of maintaining ordered lists in HyperTalk scripts. Source code in C is included. The program is free; for distribution terms see the appropriate sections in the file “Common Manual”.

This manual is intended for people who write scripts for HyperCard and who have some understanding of arrays.

Contents

Sections

Using ArrayList

- Subscripts
  - Writing subscripts
  - Subscript formats
  - Subscript format summary
- Attributes
  - Sorted
  - Compare
- Special operations
  - Set operations
  - Stack operations

Function descriptions

- Syntax description notation
- Functions
  - "!"
  - "?"
  - Add
  - BinSearch
  - Delete
  - Difference
  - Dispose
  - Error
  - Get
  - GetAttribute
  - GetDimension
  - Insert
  - Intersection
  - New
  - Pop
  - Push
  - QuickSort
  - Search
  - SeqSearch
  - Set
  - SetAttribute
  - SetDimension
  - ShellSort
  - Size
  - Sort
  - Top
  - Union

Limitations and bugs

- Limitations
- Known bugs

Version information

- Changes from earlier versions

Future plans

Abou

t the program

Appendix A. Functions (by operation)

Appendix B. Function quick reference

Appendix C. Resources used

Appendix D. Revision history

Figures

Figure 1. Array with dimensions 10x10
Figure 2. Venn diagrams for the three set operators
Figure 3. Stack operations

Tables

Table 1. Subscripts
Table 2. Subscripts for New and SetDimension
Table 3. Attributes
Table 4. ArrayList limits
Table 5. Function quick reference
Table 6. Resources used
Table 7. Revision history

Scripts

Script 1. Build sorted list

Using ArrayList

Naming

Arrays are always referred to by a name, which may be any HyperTalk string consisting of letters, underscores, and digits. As in HyperCard, capitalization does not matter in a name; thus, "ARRAY" is considered the same as "array". (Unlike HyperCard, case is important for characters with diacritics: "Å" is not the same as "å".) An optional list of subscripts may follow the array's name.

Creating

Before an array is used it must be created using the `New` function. The array will then continue to exist until it is explicitly disposed of using the `Dispose` function. ArrayList maintains a single internal index to all trees created by it, which means that all trees are globally accessible throughout your scripts.

Syntax

All functions implemented on an array are executed from a single XFCN. When calling ArrayList, the basic syntax is

```
alist(function[, parameters])
```

The first parameter always selects the function to perform on the array; subsequent parameters vary with each function.

Result

HyperCard expects an XFCN to return a value, and the calling script is required to put this return value somewhere. Since all of the operations in this program are part of a single XFCN, the script must always do something with the result returned, even if nothing useful is returned. It's simplest to use HyperTalk's `get` keyword, which will place the result into the temporary variable `it`. For instance,

```
get alist(add, array , "add me") -- adds item to array
if (it ≠ empty) then return it -- return error code
-- continue with script
```

Efficiency

The delay between the time an XFCN is called and the time it starts to execute can be quite noticeable, especially when many calls are made to the XFCN. ArrayList provides several means by which the number of individual calls may be reduced. Specifically, it allows for a condensed form of passing or getting many parameters from a single function. For instance, to retrieve a single item from an array, the following statement could be used:

```
get alist(get, array[1])
```

whereas to retrieve the entire array the following statement is more efficient:

```
get alist(get, array)
```

The result in the latter statement will be a comma separated list of the items in the array.

An inefficient way to execute some action on all the items in an array would be to write a loop, as in the following script:

```
put alist(size, array) into sz
repeat with i=1 to sz
      get alist(get, array[i])
      put it -- do something to the i'th item
end repeat
```

A much more efficient way to accomplish the same thing is shown in the following script:

```
put alist(size, array) into sz
put alist(get, array) into list
repeat with i=1 to sz
      put item i of it -- do something to the i'th item
end repeat
```

The latter example is more efficient since it executes only two calls to ArrayList, while the former executes size+1 calls to ArrayList. In general, operations which only manipulate strings, as is done in the loop in the second script, are more efficient than operations involving many calls to an external function. ArrayList is, however, very efficient once it is up and running.

Subscripts

Subscripts are used to refer to specific items or ranges of items in an array. They are also used to specify the dimensions of an array to the New and SetDimension commands. Subscripts, if specified, always follow the name of an array. Each subscript is enclosed by square brackets ([]). A list of subscripts may be given after an array's name by placing one subscript after another. In such a list, the first subscript corresponds to the first dimension of the array, the second subscript to the second dimension of the array, and so on up to the last dimension of the array. For instance, a single subscript would be written as

```
array_name[single_subscript]
```

a list of two subscripts (for an array with at least two dimensions) would be written as

```
array_name[first_subscript][second_subscript]
```

and so on for higher dimensions.

Note: Since only one-dimensional arrays are currently supported, there may only be one subscript following an array's name. Future versions will support multiple subscripts, and the behavior of ArrayList's functions will depend on the format used to specify a subscript. Multi-dimensional arrays and subscripts should be fully backwards compatible with the current one-dimensional form. Material discussing multi-dimensional arrays has been marked with a vertical bar to its left; you may skip this marked material.

Syntax

Each subscript may specify a single item, or a range of items. Indexes in a subscript are always integers, and are inclusive. Subscript ranges are specified using the three-dots character (…), which is entered using Option-semicolon; do not use three periods. Since subscripts are inclusive, a range such as 1…10 refers to items 1 through 10. Though the default subscript range is 1 to infinity, a subscript may also be negative, provided that the correct range is specified to the SetDimension command.

Internal index

Note: The information in this section is not essential to using ArrayList, so you may skip to the next section.

Arrays are really stored as a single continuous, one-dimensional, list (hence the name “ArrayList”). ArrayList calculates the actual index to an item from a given subscript by multiplying the size of the current dimension by the given subscript minus one, except for the last subscript which is simply added to the result. For instance, if an array is created with

```
get alist(new, array3D[10][10][4])
```

then the list containing the actual array has 10*10*4 = 4000 items. When an item is referred to in this array, using, for instance,

```
get alist(get, array3D[4][3][2])
```

then ArrayList retrieves the item whose internal index is

```
10*(4-1) + 10*(3-1) + (2-1) = 51
```

In the case of a one dimensional array, no multiplication is needed. For instance, if an array is created with

```
get alist(new, array1D[10])
```

then the list containing the array has 10 items. When an item is referred to in this array, using, for instance,

```
get alist(get, array1D[5])
```

then ArrayList retrieves the item whose internal index is

```
5 - 1 = 4
```

If any of the dimensions is negative, then ArrayList first adds an offset to the subscript corresponding to the negative dimension, and then multiplies by the size of the dimension. Thus, you could create an array with

```
get alist(new, arrayNegative[-4…4])
```

ArrayList will add 4 to the subscript, so that the item referred to in the following statement

```
get alist(get, arrayNegative[0])
```

corresponds to the item whose internal index is

```
0 + 4 = 4
```

The following illustration shows a two dimensional array with dimensions 10x10:

Figure 1. **Array with dimensions 10x10**

	1	2	3	4	5	6	7	8	9	10
1	0	1	2	3	4	5	6	7	8	9
2	10	11	12	13	14	15	16	17	18	19
3	20	21	22	23	24	25	26	27	28	29
4	30	31	32	33	34	35	36	37	38	39
5	40	41	42	43	44	45	46	47	48	49
6	50	51	52	53	54	55	56	57	58	59
7	60	61	62	63	64	65	66	67	68	69
8	70	71	72	73	74	75	78	77	78	79
9	80	81	82	83	84	85	86	87	88	89
10	90	91	92	93	94	95	96	97	98	99

Items in the above array have been numbered as they are indexed in the one-dimensional list used internally by ArrayList. This list is always indexed from 0 up to the size of the array, which is unlike the actual arrays, whose indexes start from 1. For instance, the item with subscript

```
array[4][7]
```

is at index 36, since

```
10*(4-1) + (7-1) = 36
```

If the second subscript were omitted, then array[4] would refer to the entire fourth row, which extends from index

```
10*(4-1) = 30
```

through

```
10*(4-1) + 9 = 39
```

Writing subscripts

I intended to use a modified form of the subscripting conventions of the C programming language. In C, subscripts are written with square brackets, so that an expression such as

```
array[3][2][1]
```

would refer to a specific item in a three dimensional array (C does not support ranges in a subscript). Unfortunately, HyperCard does not allow this format, which is used by several popular programming languages. For instance, the following HyperTalk command will not work:

```
get alist(get, array[3…5])
```

HyperCard complains when it encounters the first square bracket. One solution is to enclose the array and subscript in quotes:

```
get alist(get, "array[3…5]")
```

However, if the array's name is contained in a variable, then the variable must remain unquoted, so we can use the following:

```
get alist(get, array&"[3…5]")
```

This format is still fairly easy to read, but if the subscripts are also contained in variables we may be forced to use an even uglier form:

```
get alist(get, array & "[" & low & "…" & high & "]")
```

This format bears little resemblance to the originally intended format. Because writing lines like the last example is somewhat tedious, I will use the original format shown in the first example, omitting the quotes and ampersands, with the understanding that in an actual program the subscripts will be written in a format acceptable to HyperCard.

Your suggestions

I have been unable to arrive at a satisfactory method for specifying subscripts which is simple to specify in HyperTalk and as flexible as the current method. I am still exploring possibilities, and will be glad to receive any suggestions. Remember, though, that any method employed must be compatible with arrays having more than one dimension.

Subscript formats

Subscripts may have any one of the forms listed below. The description of each form applies to all commands except New and SetDimension, which are described after this list. Subscript ranges are specified using the three-dots character (…), which is entered using Option-semicolon; do not use three periods.

array

A missing subscript always refers to the entire array, which is treated as one long list of items. For instance, to get every item in an array, use:

```
get alist(get, array)
```

array[]

An empty subscript refers to the entire array, and is a synonym for a missing subscript. This form of subscripting will be useful when more than one dimension is supported, in which case an empty subscript will refer to the entire contents of the corresponding dimension. You should avoid using this form (even for single-dimensional arrays) since the behavior of ArrayList for multi-dimensional arrays is not yet defined.

array[integer]

Refers to the indexed item only. This is equivalent to writing array[integer… integer].

array[integer…]

Refers to the indexed item through the last item.

array[low…high]

Refers to items low through high.

array[…integer]

Refers to the first item through the indexed item.

array[…]

Refers to the first item through the last item. This is not the same as an empty or missing subscript, and will have a specific meaning when multi-dimensional arrays are supported.

Subscript format summary

The following tables show how subscripts are interpreted. The first table simply summarizes the information given in the list above, while the second table shows how the New and SetDimension commands interpret subscripts. Notice that a call to the New command may be interpreted as a call to the New command without any subscript, followed by a call to the SetDimension command with whatever subscript is desired. For instance,

```
get alist(new, array[10])
```

is the same as

```
get alist(new, array)
get alist(setdimension, array[10])
```

Because of this equivalence, the second table describes the effects of the SetDimension command only, with the understanding that the only difference between it and the New command is that the New command first creates an empty array, and then calls SetDimension.

Table 1. **Subscripts**

Form	Evaluated As
no subscript	A missing subscript refers to the entire array.
[]	Entire array (avoid using this form until multi-dimensional arrays are supported).
[i]	The i'th item.
[i…]	The i'th item through end of array.
[i…j]	Items i through j.
[…i]	The first item through the i'th item.
[…]	Entire array (not the same as []; reserved for use in multi-dimensional arrays).

The following table shows the effects of subscripts in the New and SetDimension commands. The first column shows how the subscript is specified to the New and SetDimension commands, and the second column shows the permissible range for subscripts after the command has been executed. The symbol **n** indicates an integer, and the symbol **i** indicates a subscript into the array.

Table 2. **Subscripts for New and SetDimension**

Form	Range	Evaluated As
[]	no change	No effect.

[n]	$1 \le i \le n$	Array has dimensions 1 to n. If n is smaller than the array's old size, then excess items are discarded, while if n is greater than the array's old size, then empty items are appended to the array.
[n…]	$n \le i \le \infty$	Subscripts now range from n to infinity, instead of their

previous range (the default is 1 to infinity). The size of the array isn't affected.

[a…b]	$a \le i \le b$	The array's size is set to |b-a| (the array's size is adjusted as in the case of a "[n]" subscript). Subscripts may range from a through b.
[…n]	$1 \le i \le n$	This is the same as writing "[1…n]". Notice that n must not be zero or negative.
[…]	(undefined)	Reserved for future use.

Attributes

Every array has some attributes used to modify the way an array behaves, add extra functionality to an array, and make some operations more efficient. The functions SetAttribute and GetAttribute are used to set and get the attributes' values. Attributes may have different types, such as Boolean or integer. Every attribute also has a default value that is set when an array is first used. Following is a table giving the names, possible values, and default values for all of the attributes. Following the table are descriptions of each of the attributes.

Table 3. **Attributes**

Name	Values	Default
Sorted	True	False
	False	
Compare	Exact	IgnoreCase
	IgnoreCase	
	International	
	Numeric	

Sorted

Indicates whether the array is sorted or not. Set automatically to true when the Sort function is called on the entire array, and may be set to false by some functions if a violation of sorted order is detected. When the attribute is true it modifies the operation of some functions. For instance, the Search function uses binary search if the array is sorted, otherwise it uses sequential sort. Functions that insert or delete items generally work slightly slower when this attribute is true.

Note: You should only set this attribute to true if you know for a fact that the array is sorted. If the array isn't sorted, but you have set the attribute to true, then ArrayList may exhibit odd and erratic behavior, especially when searching for items. When in doubt use the Sort function.

This attribute is most useful when you have some presorted data from which you have formed an array. By setting the sorted attribute to true you can avoid the nned to call the Sort function. Remember that the array must have been sorted according to the exact same rules used by ArrayList, otherwise it is not in sorted order as defined here. One safe method is to create a new empty array, set the sorted attribute to true, and then add the supposedly sorted data. If the data were indeed sorted correctly then the sorted attribute will still be true, otherwise it will be false. Following is an example script:

Script 1. **Build sorted list**

```
function buildSorted list, data
      get alist(new, list) -- create list
      if (it = empty)
      then get alist(setattribute, list, sorted, true)
      if (it = empty)
      then get alist(insert, list, data) -- set the data
      if (it = empty and alist(getattribute, list, sorted) ≠ true)
      then get alist(sort, list) -- sort list
      return it
end buildSorted
```

Compare

The compare attribute controls the rules for comparing items when searching and sorting. Descriptions of each of its possible values are given below.

The sorted attribute is automatically set to false when this attribute's value changes.

Exact

Items are compared exactly as they are, and ASCII ordering is used for sorting items. For instance, the following items are in sorted order: "Aardvark, Hello, ^[\, me, them, you". These ordering rules are obviously not suitable for sorting text.

IgnoreCase

Upper and lower case letters are correctly compared and sorted. When searching for an item, distinctions between upper and lower case letters are ignored, so that "UPPER" is considered the same as "upper". When sorting, letters will all be grouped together, but upper case letters will come before lower case letters. For instance, the following items are in sorted order: "aardvark,UPPER,Upper,upper". Character case is not ignored for letters with diacritics, so that "å" is not the same as "Å", even though "a" would be the same as "A".

International

Correctly compares and sorts non-English text containing diacritical marks and special characters, depending on the international resources in your System file. This is similar to the international style of HyperCard's sort command.

Numeric

Compares and sorts items numerically. Any white spaces preceding the items are ignored (eg, spaces, tabs, returns). The comparison is first done on the numeric component of the items, and then, if the items are equal, a sub-comparison is done on any non-numeric characters following the items. For instance, the item “4a” is smaller than the item “4b”. The capitalization of any extra characters is ignored, so that the item “5B” would match the item “5b”.

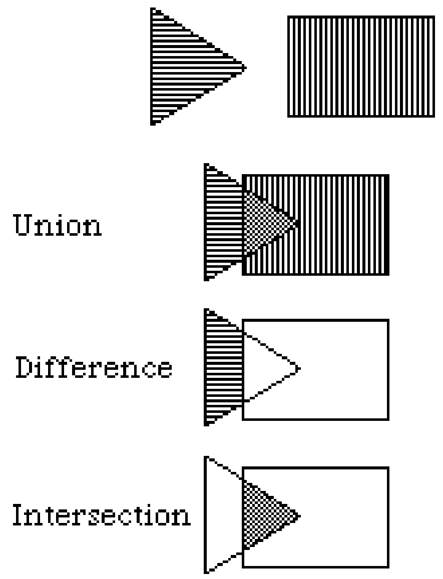
Special operations

This section describes several special types of operations provided by ArrayList.

Set operations

Several set operations adapted from Pascal are provided by ArrayList. Currently supported are the union, difference, and intersection of two sets. Each of these operations is described with the corresponding function description, but will be easier to understand with the aid of Venn diagrams, shown below.

Figure 2. **Venn diagrams for the three set operators**[1]



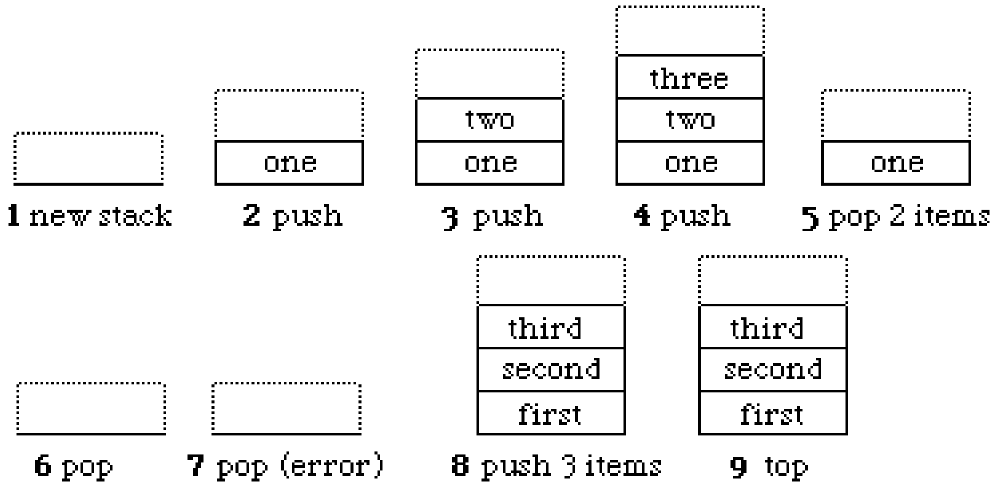
[1]E. Glinert, “Introduction to Computer Science Using Pascal”, Prentice-Hall, 1983, p. 228.

Stack operations

The three basic operations on stacks: push, top, and pop, are implemented by ArrayList. These operations, however, will only work with one dimensional arrays. The figure below shows the contents of a stack during the execution of the following commands (the lines have been numbered for reference):

```
1 get alist(new, stack)
2 get alist(push, stack, "one")
3 get alist(push, stack, "two")
4 get alist(push, stack, "three")
5 put alist(pop, stack, 2) -- prints "three,two"
6 put alist(pop, stack) -- prints "one"
7 put alist(pop, stack) -- error, stack is empty
8 get alist(push, stack, "first,second,third", ",")
9 put alist(top, stack) -- prints "third"
```

Figure 3. **Stack operations**



Function descriptions

This section contains an alphabetical list of all of the functions implemented.

Syntax description notation

The syntax descriptions use the following typographic conventions[2]. Words or phrases in `typewriter` type are to be typed literally to the computer, exactly as shown. Words in *italic* type describe general elements, not specific names — you must substitute the actual instances. Square brackets ([]) enclose optional elements which may be included if you need them. Don't type the square brackets, and don't confuse them with subscripts, since no subscripts are actually shown in syntax descriptions, though they are implicitly allowed.

The word "array" in the syntax descriptions always refers to the name of an array. Unless otherwise noted, subscripts may be appended to the names of arrays, as described above in the section about subscripts. When a subscript is specified, it limits a function's operation to the items indexed by the subscript. When no subscript is specified the function operates on the entire array.

Some examples have brief comments showing the contents of the array. For instance,

```
-- array="one,two,three"
get alist(delete, array[2])
      -- array="one,three"
```

The equals sign simply means that the array contains the items shown. If you use the Get function on the array, those are the items it would return.

Some functions may return an index into an array. This index can be used to refer to the item. For instance, to retrieve an item matching a certain string, you could use the following command

```
get alist(get, array&"["&alist(search, array, string)&"]")
```

in this example, the Search function returns an index to the item containing the string, and the Get function uses this index to get the item.

Functions

"!"

[2]Adapted from “HyperCard Script Language Guide: The HyperTalk Language”, Addison-Wesley (1988).

Syntax

```
string alist("!")
```

Description

Returns a string giving the version of ArrayList, the full name of the program, the author, a copyright notice, and the date and time of compilation. The string has the basic form "Version 0.9, ArrayList XFCN, by Ari Halberstadt, Copyright © 1990, date time".

Examples

```
get alist("!")
```

"?"

Syntax

```
string alist("?")
```

Description

Returns a string giving the a brief summary of the functions and calling methods for ArrayList.

Examples

```
get alist("?")
```

Add

Syntax

error alist(Add, *array*, *string*[, *separator*])

Description

Adds the string to *array*. If the sorted attribute is false then adds the string to the end of *array*, otherwise the string is inserted at its correct position in *array* so as to maintain sorted order. If *separator* parameter is given then Add is called on all substrings of the string delimited by the separator character.

Notes

This function only works with one-dimensional arrays.

Examples

```
                -- array is initially empty
get alist(add, array, "Item3")

                   -- array="Item3"
get alist(add, array, "Item1")

                   -- array="Item3,Item1"
get alist(sort, array)

                   -- array="Item1,Item3"
get alist(add, array, "Item2,Item0", ",")
```

```
    -- array="Item0,Item1,Item2,Item3"
```

BinSearch

Syntax

`index alist(BinSearch,` *array*, *string*`)`

Description

Uses the binary search algorithm to search *array* for an item matching *string*. The range of items being searched should be in sorted order; if the items aren't sorted then the function will not work correctly.

The binary search algorithm can locate an item in an array in time proportional to O(lgN).

See the description of the Search command for more details.

Notes

If more than one item could match *string*, then BinSearch is not guaranteed to return the first item in the sequence that matches *string*. Instead, BinSearch may return any item in the list. For instance, if part of the array contains "Joe,Mary,Mary,Mary,Samantha", and *string* is "Mary", then BinSearch could return an index to the first, second, or third instance of "Mary". In future versions, BinSearch may return the first item in such a sequence.

Delete

Syntax

`error alist(Delete,` *array*`)`

Description

Deletes items from *array*. If no subscript is given then all items in *array* are deleted; otherwise, only the subscripted items are deleted.

Examples

```
              -- Array initially contains "Item1,Item2,Item3,Item4"
get alist(delete, array[2])
                    -- now array contains "Item1,Item3,Item4"
get alist(delete, array[2…3])
                    -- now array contains "Item1"
get alist(delete, array[1])
                    -- array is now empty
```

Difference

Syntax

`error alist(Difference,` *array1*, *array2*, *array3*`)`

Description

Places the difference of *array2* from *array1* into *array3*. Both *array1* and *array2* must exist; *array3* is created to hold the results. The difference of two arrays is defined as the set of items in *array1* that are not in *array2*.

Notes

Subscripts may be used to limit the range of items selected from array1 and array2, but any subscript used with array3 is ignored. Each range of items referred to by the subscripts specified for array1 and array2 should be in sorted order, and array1 and array2 should use the same comparison rules (e.g., both should use exact, ignorecase, international, or numeric). If either list isn't sorted then the operation will not place the correct items into array3.

Array3 will be very close to sorted order; if you want to sort it you should use the ShellSort function. Also, array3 will have the default array attributes.

Examples

Following is a short script which puts "0,5,6,7" into the message box.

```
on demoDifference
      get alist(new, array1)
      get alist(new, array2)
      get alist(insert, array1, "0;3;4;5;6;7", ";")
      get alist(insert, array2, "1;2;3;4", ";")
      -- array1="0,3,4,5,6,7" and array2="1,2,3,4"
      get alist(difference, array1, array2, array3)
      put alist(get, array3)
end demoDifference
```

Dispose

Syntax

`error alist(Dispose,` *array*`)`

Description

Completely disposes of all the items in *array* and of *array* itself. Use this function when completely finished with an array.

Notes

Any subscripts are ignored.

Examples

```
get alist(dispose, array) -- array ceases to exist
```

Error

Syntax

```
error alist(Error)
```

Description

Returns the number of the error set by the last function executed. If the last function was executed successfully, then returns empty.

Examples

```
get alist(error)
```

Get

Syntax

`string alist(Get,` *array*`[,` *separator*`]])`

Description

Returns items in *array*. If no subscript is given, then returns the entire contents of *array*, otherwise returns the subscripted items. If more than one item is returned, then each item is separated by a comma (,), unless *separator* is given, in which case that character is used to separate the items.

Examples

```
                    -- Array contains "Item1,Item2,Item3,Item4"
get alist(get, array[2])

                            -- Returns "Item2"
get alist(get, array[2…4])

                            -- Returns "Item2,Item3,Item4"
get alist(get, array[2…3], ":")

                            -- Returns "Item2:Item3"
```

GetAttribute

Syntax

`string alist(GetAttribute,` *array*`,` *attribute*`)`

Description

Returns the value of the named attribute. See descriptions of array attributes for more details.

Notes

The type of the returned value depends on the type of the attribute.

Examples

```
get alist(getattribute, array, sorted)

get alist(getattribute, array, compare)
```

GetDimension

Syntax

`alist(GetDimension,` *array*`)`

Description

Returns the dimensions of *array*, as set when the array was created using the New function or by the SetDimension function. The dimensions are returned in the form of a comma separated list of items. Each item contains the size of the corresponding dimension.

Notes

The format of the data returned by this function have not been finalized.

Examples

-- array has dimensions 10*7
get alist(GetDimension, array)

```
                            -- Returns "10,7"
-- array is one dimensional, and has 10 items
get alist(GetDimension, array)

                            -- Returns "10"
```

Insert

Syntax

`error alist(Insert,` *array*`,` *string*`[,` *separator*`])`

Description

If no subscript is specified then *string* is inserted before the first item in *array*. Otherwise, *string* is inserted before the subscripted item. If *separator* is given then every substring in the string delimited by the separator character is inserted in turn into *array*. If the sorted attribute is true and the insertion would result in a violation of sorted order then the sorted attribute is set to false.

Notes

To append an item to an array use a subscript equal to the size of the array plus 1.

The subscript may range from 1 to the size of the array plus 1. In the latter case the inserted items are appended to the end of the array. This feature was included to allow for insertion into an empty array (in which the size is 0 and the index is 1).

Examples

```
-- Array initially contains "Item1,Item3"
get alist(insert, array[2], "Item2")
      --array="Item1,Item2,Item3"
get alist(insert, array[2], "ItemX:ItemY", ":")
      --array="Item1,ItemX,ItemY,Item2,Item3"
get alist(insert, array[6], "Appended")
      --array="Item1,ItemX,ItemY,Item2,Item3,Appended"
get alist(insert, array, "First")
      --array="First,Item1,ItemX,ItemY,Item2,Item3,Appended"
```

Intersection

Syntax

`error alist(Intersection,` *array1, array2, array3*`)`

Description

Places the intersection of *array1* and *array2* into *array3*. Both *array1* and *array2* must exist; *array3* is created to hold the results. The intersection of two arrays is defined as the set of items that are in both *array1* and *array2*.

Notes

Subscripts may be used to limit the range of items selected from array1 and array2, but any subscript used with array3 is ignored. Each range of items referred to by the subscripts specified for array1 and array2 should be in sorted order, and array1 and array2 should use the same comparison rules (e.g., both should use exact, ignorecase, international, or numeric). If either list isn't sorted then the operation will not place the correct items into array3.

Array3 will be very close to sorted order; if you want to sort it you should use the ShellSort function. Also, array3 will have the default array attributes.

Examples

Following is a short script which puts "3,4" into the message box.

```
on demoIntersection
      get alist(new, array1)
      get alist(new, array2)
      get alist(insert, array1, "0;3;4;5;6;7", ";")
      get alist(insert, array2, "1;2;3;4", ";")
      -- array1 = "0,3,4,5,6,7" and array2 = "1,2,3,4"
      get alist(intersection, array1, array2, array3)
      put alist(get, array3)
end demoIntersection
```

New

Syntax

`error alist(New,` *array*`)`

Description

Creates a new array. This function must be called before an array can be used. If no subscript is specified then the array is empty; otherwise, the array's dimensions are set to those specified in the subscript.

Notes

If no upper limit is specified for the size of an array, then some functions, such as insert, add, and set, may expand the array's size. If an upper limit is specified, then no expansion is possible without redimensioning the array.

Multi-dimensional arrays are not yet supported. When they are, the array's size will be set to the product of the dimensions.

Examples

get alist(new, array)
-- creates an empty one-dimensional array, and all
-- subscripts must be from 1 through the current

```
                              -- size of the array.
get alist(new, array[4])
      -- new array has 4 empty items, and all subscripts
                              -- must be from 1 through 4.
```

Pop

Syntax

`string alist(Pop,` *array*`[,` *count,* `[`*separator*`]])`

Description

Pops and returns the top element of the stack formed by *array*. If *count* is given, then Pop is called repeatedly for *count* items. Items are separated by commas (,) unless *separator* is given, in which case that character is used to separate items.

Notes

This function only works with one-dimensional arrays.

Examples

-- Initially array is "one,two,three,four"
get alist(pop, array)
-- Returns "four"; array="one,two,three"
get alist(pop, array, 2, ":")

```
                              -- Returns "three:two"; array="one"
```

Push

Syntax

`error alist(Push,` *array, string*`[,` *separator*`])`

Description

Pushes *string* onto the stack formed by *array*. If *separator* is given then every substring in the string delimited by the separator character is pushed in turn onto *array*. If the sorted attribute is true and the operation would result in a violation of sorted order then the sorted attribute is set to false.

Notes

This function only works with one-dimensional arrays.

Examples

```
-- Array is initially empty
get alist(push, array, "one")
            -- array="one"
get alist(push, array, "two;three", ";")
            -- array="one,two,three"
```

QuickSort

Syntax

`error alist(QuickSort, `*array*`)`

Description

Uses the quick sort algorithm to sort *array*, regardless of the state of the sorted attribute.

The average case performance of quick sort is O(NlgN). This version of quick sort utilizes several enhancements over a naive implementation, which save between 20% and 30% of the running time, in addition to making worst case performance of O(N^2) — which in a naive implementation could occur when sorting an already sorted array — very unlikely to occur. Despite these enhancements, the shell sort algorithm tends to be faster for presorted data.

Notes

The order in which items are sorted depends on the value of the compare attribute.

The sorted attribute is set to true only if the entire array is specified to the QuickSort function.

Examples

get alist(quicksort, array)
-- sorts the entire array

```
                              -- sorted attribute is set to true
get alist(quicksort, array[3…10])
      -- sorts items 3 through 10

                              -- sorted attribute is unchanged
```

Search

Syntax

`index alist(Search,` *array*, *string*`)`

Description

Searches *array* for an item matching *string*, and returns an index describing the location of the item in the array, or empty if *string* isn't found. If no subscript is given then the entire array is searched, otherwise only the subscripted range is searched. If the sorted attribute is true then Search uses binary search to locate the item, otherwise it uses sequential search.

Notes

THE FORMAT OF THE DATA RETURNED BY THIS FUNCTION HAS NOT BEEN FINALIZED. The subscript returned is in the form of a list of items, where the first item corresponds to the first dimension, the second item to the second dimension, etc. Thus, a search in a one dimensional array will always return a single number, a search in a two dimensional array will return two numbers, etc.

Examples

```
-- Array contains "Item1,Item2,Item3,Item4"

get alist(search, array, "Item2") -- Returns 2

get alist(search, array, "Item7") -- Returns empty
get alist(search, array[1…3], "Item4") -- Returns empty
```

SeqSearch

Syntax

index alist(SeqSearch, *array*, *string*)

Description

Uses the sequential search algorithm to search *array* for an item matching *string*.

The sequential search algorithm can locate an item in an array in O(N) time, and typically locates items in O(N/2).

See description of the Search command for more details.

Set

Syntax

`error alist(Set,` *array*`,` *string*`[,` *separator*`])`

Description

Replaces the value of the subscripted item in *array* with *string*. If *separator* is given, then Set is called on all substrings delimited by the separator in the string, starting at the subscripted item and advancing the subscript by one for each substring. If the sorted attribute is true and the operation would result in a violation of sorted order, then the sorted attribute is set to false.

Examples

-- Array initially contains "Item1,Item3".
get alist(set, array[1], "ItemX")
-- array="ItemX,Item3"
get alist(set, array[1], "ItemY:ItemZ", ":")

```
    -- array="ItemY,ItemZ"
```

SetAttribute

Syntax

`string alist(SetAttribute,` *array*`,` *attribute*`,` *value*`)`

Description

Sets the value of the named attribute. See descriptions of array attributes for more details.

Notes

The type of the value parameter depends on the type of the attribute.

Examples

```
get alist(setattribute, array, sorted, false)

get alist(setattribute, array, compare, numeric)
```

SetDimension

Syntax

```
error alist(SetDimension, array)
```

Description

Sets the dimensions of *array* to correspond with the given subscripts. If the total size of *array* is reduced then extra items are discarded, while if *array* expands then empty items are appended. This command is similar to the New command, except that it works on an existing array.

ShellSort

Syntax

```
error alist(ShellSort, array)
```

Description

Uses the shell sort algorithm to sort the array, regardless of the state of the sorted attribute.

The shell sort algorithm's expected average performance is O(N$^{3/2}$). Shell sort will provide fairly good average case performance, though it is almost always slower than quick sort when applied to random data, whose average performance is O(NlgN). When applied to presorted data shell sort may be faster than quick sort.

Notes

The order in which items are sorted depends on the value of the compare attribute.

The sorted attribute is set to true only if the entire array is specified to the ShellSort function.

Examples

get alist(shellsort, array)
-- sorts the entire array

```
                              -- sorted attribute is set to true
get alist(shellsort, array[3…10])
      -- sorts items 3 through 10

                              -- sorted attribute is unchanged
```

Size

Syntax

```
integer alist(Size, array)
```

Description

Returns the number of items in *array*. An empty array has zero items, and an array with dimensions 3 by 5 has 15 items.

Examples

```
get alist(size, array)
```

Sort

Syntax

```
empty alist(Sort, array)
```

Description

Sorts the array. If the entire array is specified to the sort command, and if the sorted attribute is false, then the quick sort algorithm is used, otherwise the shell sort algorithm is used.

Notes

The order in which items are sorted depends on the value of the compare attribute.

The sorted attribute is set to true only if the entire array is specified to the Sort function.

Examples

get alist(sort, array)
-- sorts the entire array

```
                              -- sorted attribute is set to true
get alist(sort, array[3…10])
      -- sorts items 3 through 10
                              -- sorted attribute is unchanged
```

Top

Syntax

`string alist(Top,` *array*`[,` *count*, `[`*separator*`]]`

Description

Returns the top element of the stack formed by *array*. If *count* is given, then Top is called repeatedly for *count* items. Items are separated by commas (,) unless *separator* is given, in which case that character is used to separate items.

Notes

Can be used to get the items of an array in reverse order by using the following statement:

```
get alist(top, list, alist(size, array))
```

It is an error to request the top item of an empty array.

This function only works with one-dimensional arrays.

Examples

```
                        -- List is "one,two,three,four"
get alist(top, array)
                              -- Returns "four"
get alist(top, array, 3, ":")
                              -- Returns "four:three:two"
```

Union

Syntax

`error alist(Union,` *array1*, *array2*, *array3*`)`

Description

Places the union of *array1* and *array2* into *array3*. Both *array1* and *array2* must exist; *array3* is created to hold the results. The union of two arrays is defined as the set of items that are either in *array1* or *array2* or both.

Notes

Subscripts may be used to limit the range of items selected from array1 and array2, but any subscript used with array3 is ignored. Each range of items referred to by the subscripts specified for array1 and array2 should be in sorted order, and array1 and array2 should use the same comparison rules (e.g., both should use exact, ignorecase, international, or numeric). If either list isn't sorted then the operation will not place the correct items into array3.

Array3 will be very close to sorted order; if you want to sort it you should use the ShellSort function. Also, array3 will have the default array attributes.

Examples

Following is a short script which puts "0,1,2,3,4,5,6,7" into the message box.

```
on demoUnion
      get alist(new, array1)
      get alist(new, array2)
      get alist(insert, array1, "0;3;4;5;6;7", ";")
      get alist(insert, array2, "1;2;3;4", ";")
      -- array1 = "0,3,4,5,6,7" and array2 = "1,2,3,4"
      get alist(intersection, array1, array2, array3)
      put alist(get, array3)
end demoUnion
```

Limitations and bugs

This section describes any limitations on the size and number of data that the program may manipulate. Also discussed are any known bugs, with suggested ways to work around them.

Limitations

This section lists various minimum and maximum sizes for arrays. All limits may be smaller depending on the availability of memory and other computer resources. It is unlikely ArrayList will actually encounter an error associated with the exhaustion of available memory since HyperCard is more likely to quit first.

In the following table, the value represented by Integer is 32,767 and the value represented by LongInt is 2,147,483,647.

Table 4. **ArrayList limits**

Item	Limit
Number of arrays	LongInt
Items in an array	LongInt
Length of an item	LongInt

Known bugs

This section is included for updates on possible and real bugs, and for the dissemination of temporary solutions. Pseudo-bugs will also be reported here (a pseudo-bug is defined as “weird behavior deriving from the correct definition of the software”).

- Since this program is still under development, not all of the features described in this manual are fully implemented or defined. This will mostly affect the operation of subscripts and the treatment of multi-dimensional arrays.

Version information

This section describes features that have changed from previous versions. Also discussed are plans for the future of this program.

Changes from earlier versions

This is the first version, so there have been no changes.

Future plans

I intend to add a capability for multi-dimensional arrays. This will be a simple extension of the features already implemented.

Also in the works is a method for maintaining a stack of groups of arrays. This will facilitate local arrays within handlers, where one push is done at the start of a handler and one pop is done before the same handler exits. This scheme will also support global arrays, so that it will most likely be backwards compatible with the current version of ArrayList.

I may add another value to the `compare` attribute for comparing text. This attribute would ignore punctuation, and other non-text characters. I may also implement multi-field comparisons, so that you can sort and search on more than one field within a single item.

About the program

«SECTION NOT YET AVAILABLE»

Appendix A. Functions (by operation)

This section lists the functions implemented according to the operations they perform.

Getting information about ArrayList

```
string      alist("!")
string      alist("?")
```

Getting errors

```
error       alist(Error)
```

Creating and disposing of arrays

```
error       alist(New, array)
error       alist(Dispose, array)
```

Inserting and deleting items

```
error       alist(Insert, array, string[, separator])
error       alist(Delete, array)
```

Adding items

```
error       alist(Add, array, string[, separator])
```

Getting and setting items

```
string      alist(Get, array[, separator])
error       alist(Set, array, string[, separator])
```

Getting size of arrays

```
integer     alist(Size, array)
```

Getting and setting dimensions

```
error       alist(GetDimension, array)
error       alist(SetDimension, array)
```

Getting and setting array attributes

```
string      alist(GetAttribute, array, attribute)
error       alist(SetAttribute, array, attribute, value)
```

Searching

```
index       alist(Search, array, string)
index       alist(BinSearch, array, string)
index       alist(SeqSearch, array, string)
```

Sorting

```
empty       alist(Sort, array)
empty       alist(QuickSort, array)
empty       alist(ShellSort, array)
```

Set operations

error   alist(Union, *array1*, *array2*, *array3*)

`error        alist(Difference,` *array1*, *array2*, *array3*`)`
`error        alist(Intersection,` *array1*, *array2*, *array3*`)`

Stack operations

`error        alist(Push,` *array*, *string*, `[`*separator*`])`
`string       alist(Pop,` *array*, `[`*count*, `[`*separator*`]])`
`string       alist(Top,` *array*, `[`*count*, `[`*separator*`]])`

Appendix B. Function quick reference

The following table is an alphabetic list of all of the functions implemented, the types of data they return, and their syntax.

Table 5. **Function quick reference**

Returns	Syntax
`string`	`alist("!")`
`string`	`alist("?")`
`error`	`alist(Add,` *array*`,` *string*`[,` *separator*`])`
`index`	`alist(BinSearch,` *array*`,` *string*`)`
`error`	`alist(Delete,` *array*`)`
`error`	`alist(Difference,` *array1*`,` *array2*`,` *array3*`)`
`error`	`alist(Dispose,` *array*`)`
`error`	`alist(Error)`
`string`	`alist(Get,` *array*`[,` *separator*`]])`
`string`	`alist(GetAttribute,` *array, attribute*`)`
`error`	`alist(GetDimension,` *array*`)`
`error`	`alist(Insert,` *array*`,` *string*`[,` *separator*`])`
`error`	`alist(Intersection,` *array1*`,` *array2*`,` *array3*`)`
`error`	`alist(New,` *array*`)`
`string`	`alist(Pop,` *array*`, [`*count*`, [`*separator*`]])`
`error`	`alist(Push,` *array*`,` *string*`, [`*separator*`])`
`error`	`alist(QuickSort,` *array*`)`
`index`	`alist(Search,` *array*`,` *string*`)`
`index`	`alist(SeqSearch,` *array*`,` *string*`)`
`error`	`alist(Set,` *array*`,` *index*`,` *string*`[,` *separator*`])`
`error`	`alist(SetAttribute,` *array*`,` *attribute, value*`)`
`error`	`alist(SetDimension,` *array*`)`
`empty`	`alist(ShellSort,` *array*`)`
`integer`	`alist(Size,` *array*`)`
`empty`	`alist(Sort,` *array*`)`

`string`	`alist(Top,` *array*`, [`*count*`, [`*separator*`]])`
`error`	`alist(Union,` *array1*`,` *array2*`,` *array3*`)`

Appendix C. Resources used

This appendix gives a complete list of resources needed by this program. These resources must be installed in your stack for the program to work (see the section on installation in the common manual).

The following table lists the resources with their default IDs and names, along with a short description of the data contained in each resource and how the resource is used by the program. The resource of type TABL is described in the common manual.

Table 6. **Resources used**

Type	Name	Description
XFCN	alist	The little XFCN whose purpose it is to load, lock, and call the PROC resource.
PROC	ArrayList	The resource containing the executable code.
STR#	ArrayList:ResourceMap	Map of resources used by ArrayList.
STR#	ArrayList:Info	Version and usage information.
TABL	ArrayList:Functions	Names of the functions.
TABL	ArrayList:Attributes	Names of the attributes.
TABL	ArrayList:Compares	Names of comparison methods.

Appendix D. Revision history

This section is to be used for recording any changes made to this manual. This is necessary since I do not want inconsistencies or mistakes introduced by others to reflect on my reputation, and, if the revisions improve this product, then the person who made the improvements should receive full credit. For consistency, please enter dates as Year-Month-Day.

Table 7. **Revision history**

Date	Name	Comments
90-07-18	Ari Halberstadt	This is an example entry
90-07-11	Ari Halberstadt	Version 0.9