ACME@Kagi.com • wkw@futuris.net

Acme Script Widgets

ACME Script Widgets are a collection of AppleScript™ Scripting Additions which provide powerful text and list manipulation capabilities. AppleScripts will execute from 2 to 20 times faster and require fewer lines of code when using the Widgets.

ACME Script Widgets are provided on a 30 day evaluation license. The thirty day period begins from the day you first install them. After the evaluation period has expired, you are required to either purchase the appropriate license, or remove them from all computers where they are installed.

Support is available to all ASW customers, as well as people using the Widgets during the evaluation period. Send questions to ACME@kagi.com. The ACME World Wide Web site is currently being built at http://www.gz.com/acme/.

This software may not be uploaded to any online service without prior written permission.

Ordering Information

To order ACME Script Widgets, use the “Register” program included with this distribution, or contact Wayne Walrath at 203-857-0631 or by E-mail at ACME@kagi.com. Payment can be made by credit card, check, cash, or with a First Virtual account.

If you are ordering by E-mail, Register disguises your credit card information enough that it isn't obvious what the data is, but it is not guaranteed secure. For E-mail registration, use the "Copy..." button to get the registration information onto the clipboard, then paste it into an E-mail message and send it to shareware@kagi.com. If you are mailing your payment in, use the "Print..." button and a form will be printed which contains your registration information and special bar codes to automate the processing at Kagi. Register also supports printing an invoice which you can give to your purchasing department. To get more help on using Register, turn on Balloon Help after launching the program.

Licenses Fees

Single User License (one CPU)	US$ **29.00**
Site License (unlimited copies within same organization in a 100 mile (160 kilometer) radius)	US$ **200.00**
World-Wide License (unlimited copies within organization)	US$ **600.00**

For commercial bundling, solution provider redistribution or other arrangements, contact Wayne Walrath at acme@kagi.com, (203) 857-0631.

Mail or fax registrations to:

Kagi Shareware
1442-A Walnut Street #392-WW
Berkeley, CA 94709-1405
USA
shareware@kagi.com
fax +1 510 652 6589

Why Scripting Additions?

Scripting Additions (also known as osaxen or osax in the singular because ‘osax’ is the file type for Scripting Additions) are compiled C/C++ code resources which execute many times faster than pure AppleScript which is more or less an interpreted language. AppleScript is a general purpose, high-level language designed to control and tie together other applications. By supporting Apple events, other applications provide the powerful, special purpose features like graphics manipulation, or communications and database services. However, there are many occasions where it is quicker and simpler to just manipulate basic data from within an AppleScript and not have to request the services from another application. To support that, AppleScript can be extended with Scripting Additions.

The Acme Script Widgets were created to handle such common tasks as searching for text strings, or manipulating AppleScript list variables.

Ninety percent of the functionality the Script Widgets provide can be achieved with pure AppleScript statements alone, and if the amount of data you are working with is very small, this works just fine and nearly as fast. But even casual script writers will very soon reach the point where they want more speed out of the scripts they write, and Acme Script Widgets are guaranteed to provide that along with reducing the number of AppleScript statements required.

Here’s a simple example which illustrates the difference using the Script Widgets can make for a typical operation. Given a line of text, the goal is to remove all spaces and periods from both ends of the text.

```
set theString to ".....On a clear day you can see forever.      "

set currentChar to length of theString
-- remove periods and spaces from the end of the string first
repeat while ((character currentChar of theString = " ") or ¬
        (character currentChar of theString = " ")) ¬
        and (currentChar > 0)
        set currentChar to (currentChar - 1)
end repeat
set theString to characters 1 thru currentChar of theString as text

set currentChar to 1
-- now remove periods and spaces from the beginning of the string
repeat while ((character currentChar of theString = ".") or ¬
        (character currentChar of theString = ".")) ¬
        and (currentChar < length of theString)

        set currentChar to (currentChar + 1)
end repeat
set theString to characters currentChar thru -1 of theString as text
```

And now the same task but with the Acme Script Widget Trim command

```
-- reset the data string for the next example
set theString to ".....On a clear day you can see forever.      "

-- now do the same using the Trim command
set theString to trim {".", " "} off theString from both sides
```

Both of these AppleScript code snippets were timed using an osax called “the ticks” (from Jon’s Commands) which returns the Macintosh’s current tick count (there are sixty ticks to a second). The AppleScript-only solution ran in about 25 ticks, but the Trim version only needed about 5 ticks (clocked on a PPC 6100/66 in Script Debugger, your times will likely vary, but the relative difference should be similar). Even for this relatively short string of characters, Trim accomplished the task in fifth of the time it took to do with just AppleScript statements. And the AppleScript-only solution required ten lines of code to do what Trim did in only one.

Using the Widgets almost always results in fewer lines of code, however the actual performance will vary depending on many factors. Often times the difference is much more dramatic. On the next page are a few of the comments Acme Script Widget users have sent to us.

Praise for Acme Script Widgets

From: Dennis L. Whiteman, 72740,3146
To: Wayne Walrath, 73243,3303
Date: 24 Oct, 1994, 10:14
RE: Tokenize Scripting Addition: Final Version

I set up a droplet and database over the weekend that lets me drop e-mail saved from CompuServe and eWorld (either as text files or clippings) into a Filemaker correspondance database. I used both Tokenize and Join Lists and they worked great. Probably saved me over 50 lines of code each. Keep up the good work!

Dennis L. Whiteman
THE ULTIMATE FREELANCER

From: John Craig, 74660,2313
To: Wayne Walrath, 70233,3151
Date: Wed, 12 Oct, 1994, 17:52
RE: TokenizeII OSAX

Thank you for sending me the Scripting addition!

I've just ripped out about 100 lines of code that painfully parsed some data records from a 4th Dimension server application and sent them to Excel and Delta Graph. The whole process has been cut to about 25% of the original time it took to parse the data (1 hour to 13 minutes!).

Thanks again!

John Craig

[...] You should, however, check out Wayne Walrath's ACME script widgets which are fantastic. The script below uses his tokenize osax and is much shorter and never messes with the AS delimiters.

Andrew Olson
olson_andrew@bcgmac.bcgny.com

Praise for Acme Script Widgets

I've come across a lot of scripting additions, and Tokenize is one of the most genuinely useful. It was very insightful of you to capitalize upon the current limitations of AppleScript's text item delimiters. [...]

Regards,
David Jokinen
Ground Zero Software (eWorld Shortcut = ground zero)

Date: Sat, 3 Dec 1994 08:11:15 -0600
Sender: <MACSCRPT@DARTCMS1.DARTMOUTH.EDU>
From: Steve Alex <steve_alex@AIDT.EDU>
Subject: Thank you ACME Script Widgets!

[...] "Tokenize" allowed us to take a routine that was about 150 lines and get it down to about 30. I could not pass up that kind of speed and size savings, so I put a period in front of it (.tokenize -- we use the period to distinguish "third party" OSAX) and threw it in the scripting additions folder. I said "Thank you Wayne." to myself and went on solving the worlds problems.

[...] Now comes "Offset In List" (part of ACME Script Widgets 1.0) and he causes me to chunk a neat recursive script object I spend a couple days writing that found the offset of a string in a list of strings. What's worse is that he solved some problems I don't even thing he realized he solved! I just glanced at the read me, opened an example, said "Um, lets try this" [...]

Steve Alex (steve_alex@aidt.edu)

Acme Script Widgets Overview

Package contents and file locations:

/ACME Script Widgets 2.5
The core scripting additions. Install in your Scripting Additions folder, inside the Extensions folder. *Contents*

/Register (ACME SW)
An application for purchasing the software license(s).

/Others.../ACME parse args/
Scripting Additions for writing WebSTAR CGIs. These commands parse the post_args and search_args in a CGI script application.

/Others.../Balloon Help/
Scripting Addition for turning balloon help on or off, or determining the current state (on or off).

/Others.../Mouse/
Get mouse coordinates.

/Others.../Mama's Little Helper/
This is an unsupported experimental Scripting Addition for use in writing WebSTAR CGIs.

/Demo Scripts/
Scripts which demonstrate all the commands provided by the core Scripting Additions (those contained in the ACME Script Widgets 2.5 file).

/Documentation/
User guide.

QuickQuide to Acme Script Widgets

What follows is a quick guide to each of the main commands. The commands are explained in more detail later in the document, but one of the best ways to learn what each function does is to look at the corresponding demo AppleScript located in the “Demo Scripts” folder, and to open the dictionary for the Scripting Additions from within your script editor.

Tokenize	Similar to working with AppleScript™’s Text Item Delimiters, but much more powerful. Pulls “tokens” out of text strings based on any number of separators you provide.
Join List	The reverse of Tokenize. Build text strings from lists of element inserting any number of separators between the list items.
Acme Replace	Search for text strings and replace them with other strings.
Offsets of	A searching function which returns the offsets into a text string of another string.
Offset in List	Quickly search through a list for a string, returning the index where it was found.
Trim	Remove characters or strings from either end of a block of text. Supports AppleScript’s “ignoring” keyword so that a list of ignored patterns can be specified while trimming the text.
Acme Sort	Sort a list of items or a list of lists into ascending or descending order.
Combine Lists	Useful for FileMaker Pro scripters and anyone who has to manipulate nested lists. Given a list of lists, it regroups the items in the inner lists according to their indexes.
Acme Parse Args	**WebSTAR** CGI writers take note! This command parses all the input arguments WebSTAR sends to an AppleScript CGI or ACGI. Convert %XX encodings, split label/value pairs out, etc.
Acme Lookup Field	a complement to Acme Parse Args, let’s you quickly locate any HTML form field value after parsing the input data with Acme Parse Args.

ACME Script Widgets Change History

The ACME Script Widgets are continually being updated and expanded. There were many significant changes and one or two bug fixes between the two public releases (1.0 and 2.0), and most of these are documented below. Version 2.5 adds some new commands and implements some users' requests.

ACME Replace: (formerly called Replace)

Changed the terminology of the osax to reduce the chance of collisions in the terminology space, since many applications define the replace command. Implemented case sensitivity. NOTE: Using the case insensitive option causes Replace to run from a third to half again as long as when it isn't specified, and the memory usage nearly doubles. So don't use this option if you do not need it. Due to the increased memory demands, the osax attempts to allocate space in temporary memory first if it is available, and if not, then attempts to get enough in the local Heap zone.

Offsets Of:

Removed 255 char limit on size of search string. The next version will support passing a list of strings to tokenize instead of having to do it one at a time, and will remove the 255 character limit on the size of delimiter items.

Offset In List:

Totally rewrote the association lists demo and broke it out into its own script. The Association List is now a script object with methods for all the common operations. You can load it into your scripts as a property and get flexible association lists effortlessly.

Mouse:

New addition to satisfy a user's request. Donald Olsen was the first to create this command I believe, and this version doesn't do anything that his doesn't.

Combine Lists:

New to version 2.0. Useful for working lists of records and repeating fields from FileMaker Pro.

Trim:

New to version 2.0. Trims any number of strings or characters off either end of the target strings, with the option of ignoring any number of patterns. This is going to be one of the hot-ones!!

Tokenize:

Version 1.2 was rewritten to return null tokens. This makes the behavior exactly the same as using AppleScript's text item delimiters property, and means that running a string first through Tokenize, then the output through Join List will produce the original input string exactly. If you were using a version prior to 1.2 you'll need to use the optional parameter "null tokens false" in order to get exactly the same behavior in your scripts as with 1.1.

Acme Sort:

New in version 2.5.

Acme Parse Args/Acme Lookup Field :

New in version 2.5.

Installation

The Scripting Additions must be copied to the Scripting Additions folder which resides inside the Extensions folder of the System folder on English language versions of the Macintosh operating system. For non-English versions, please refer to the documentation which came with your Macintosh if you are unsure where they belong.

Installing ACME Script Widgets:

The file ACME Script Widgets 2.5 contains all of the core Widgets. Other special purpose Widgets are provided in separate files so they only need to be installed if required. As a general rule, you should not let your Scripting Additions folder fill up with many commands which you almost never use because it takes longer for your scripts to compile. The difference is not extreme, so don't remove commands which you do use on a semi-regular basis, but it is a good idea to review which commands are in that folder from time to time and move unused additions to a different location until you need them again. You'll certainly want to keep the Acme Script Widgets file in there at all times since they are useful for so many purposes!

ACME Replace

ACME Replace is a general purpose search and replace tool for AppleScript™. You can specify one or more strings to search for in one or more target strings, and each occurrence will be replaced by the specified replacement text. Replace has been renamed to ACME Replace to avoid terminology conflicts with other Scripting Additions and applications.

USAGE:

ACME replace: replaces occurrences of a string with something else.

```
ACME replace  anything  -- string or list of strings to search for.
        in  anything  -- the string or list of strings to search in.
        with  string  -- the replace string.
        [all occurrences  boolean]  -- replace all occurrences?  (default=TRUE)
        [case  sensitive/insensitive]  -- Consider case? (Default is sensitive.)

Result:   anything  -- original text item(s) with replacements.
```

There are no limits on the size of the parameters (other than available memory), and anything that AppleScript™ can coerce to a string is legal (e.g., integers, floats, etc.).

If you pass a list of strings to search for, they are replaced one at a time in the order they are passed. If you specify that only the first occurrence should be replaced, each one of the search strings will be replaced once. This might be useful if you wanted to replace the first X number of a certain string. For example:

```
ACME replace {1, 1, 1} in "12121212" with "_" without all occurrences

=>     "_2_2_212"
```

Three of the ones (1) were replaced with "_", and the rest left untouched.

NOTE: Using the case insensitive option causes Replace to run slower than when it isn't specified, and the memory usage nearly doubles. So don't use this option if you do not need it.

Acme Replace (cont.)

NOTE 2 (important!): avoid specifying ‘all occurrences true’ as an option when using Acme replace. You never need to specify this since the default behavior **is** to replace all occurrences (i.e., true), but more importantly, your script will not recompile properly. The problem lies in the way AppleScript converts boolean parameters to “with / without” syntax, and due to terminology conflicts ‘with all occurrences’ does not properly compile.

Very Important! Acme replace does not change the text in a variable. In other words, the following lines of AppleScript will not change the value stored in myVar:

```
set myVar to “555-1212”
Acme replace “555” in myVar with “*”
```

AppleScript never makes “in place” changes to variables like this. You must set a variable to a new value:

```
set myVar to Acme replace “555” in myVar with “*”
```

now myVar has been changed. This is one of the biggest stumbling blocks for first time users of Acme replace.

Change Case

Change Case transforms the case of the text passed to it in a variety of ways. This Scripting Addition is most likely ONLY useful for working with English language text, since other languages have their own rules for capitalization.

USAGE:

change case: changes the case of (Roman) text.

change case
of anything -- text to modify. (string or list)
[to upper/lower/title case/sentence case/toggle case/who cares]

Result: anything -- original text with case changed.

The upper and lower case transforms do as their name implies.
Title case makes the first alpha character after a non-alpha character upper case.

Sentence case searches for the first alpha character in the string and capitalizes it, then continues on searching for one of /! / . / ? / and capitalizes the next alpha character after seeing one of the punctuation marks (all other characters are left untouched).

Toggle just switches the case of each character from whatever it was.

The "who cares" option was added mostly for fun (well, totally for fun) and randomly changes the case of each letter (if anyone finds a use for this option other than creating electronic ransom notes, please let me know...).

Combine Lists

Combine Lists reorganizes the items in sublists, grouping the Nth item from each of the inner lists together into a new list. For instance, the list {{1,2},{a,b}} becomes {{1,a},{2,b}} after processing. This Widget is quite handy to have around when scripting FileMaker Pro. FileMaker Pro returns multiple records as a list of lists, where each inner list contains all the fields for a single record. Combine Lists will reorganize the records so that all data from each field is grouped together; in other words, all the values for the first field are together in one list, all the values for the second field in another, etc. Combine Lists also has other applications outside of working with FileMaker Pro records.

USAGE:

combine lists: Groups the Nth item from each sublist together.
combine lists list -- List of lists.
Result: list -- Returns list of lists with items reorganized.

given the following list (which could represent three records from a FileMaker Pro database with fields: name, age, and house address):

{ {"Jon", 10, "10 Main St."**}**, **{**"Mary", 20, "301 Washington Ave."**}**, **{**"Sue", 15, "100 South Water St."**} }**

Combine Lists returns:

{ { "Jon", "Mary", "Sue" **}**, {10,20,15 **}**, {"10 Main St.","301 Washington Ave.","100 South Water St." **} }**

The Nth item from each sublist is grouped together into a new sublist.

Running the output back through Combine Lists a second time will produce the original list. Combine Lists requires that all sublists have the same number of elements or it returns an error.

Join List

Join List forms a string from a list of items, inserting delimiters (in a repeating fashion if their are more than one) between the strings. Join list is the reverse of the Tokenize operation.

USAGE:

join list: form string from list items with the delimiters inserted in between.

```
join list  list  -- list of text items to join.
        with delimiters  list  -- list of delimiter items.

Result:   string  -- Returns the joined list as a string.
```

The first parameter is a list of zero or more strings to be joined into a string. If no delimiters are specified, all items from the list will be concatenated together back to back. There is no limit on the size of the items other than available memory.

The second parameter specifies one or more strings to insert between the joined items.

Delimiters are inserted between items of the first list sequentially; when the end of the delimiter list is reached, the Scripting Addition begins again with the first delimiter (they are inserted in a rotating fashion).

EXAMPLES:

```
join list {"Join", "List", "Ver1.0"} with delimiters {space}
        => "Join List Ver1.0"
```

The _space_ character is repeatedly inserted between the items in the first list. Here's a more complex example:

```
set jList to {"One", "Two", "Three", "Four", "Five"}
set dList to {"$", "#"}
join list jList with delimiters dList
        => "One$Two#Three$Four#Five"
```

The "$" is inserted between the first two items and the "#" between the second and third items, then because the end of the delimiter list was reached, it starts over at the beginning of the delimiters again with "$", continuing on in this manner until all of the strings in jList have been added.

If an empty list of delimiters is specified ({}), the command behaves exactly as if you had used AppleScript™ to coerce the list of strings to a single string.

```
set jList to {"One", "Two", "Three", "Four", "Five"}
join list jList with delimiters {}
        => "OneTwoThreeFourFive"
```

If the list of strings to be joined contains only a single element, join list returns just that element with none of the tokens appended.

```
join list "one" with delimiters {"&", "*"}
        => "one"
```

If you want any of the delimiters on the front or end of the returned string, use AppleScript's built-in capabilities for this.

```
set jList to {"One", "Two", "Three", "Four"}
set myString to "|" & (join list jList with delimiters "|") & "|"
        => "|One|Two|Three|Four|"
```

Offset In List

Offset In List searches through a list of items for a string. It searches for the pattern in each item of the list, or optionally only even or odd numbered items, and there are options for performing exact match and case sensitive searches.

USAGE:

offset in list: search list items, returning found string's index or next item.

offset in list list -- list of items to search.
        of string -- the search string.
        [returning next item boolean] -- (index returned by default)
        [searching even items/odd items/all items] -- Default: check every item.
        [case sensitive/insensitive] -- ignore case? (Default: case is significant.)
        [exact match boolean] -- whole word matching only? (Default = true.)

Result: anything -- item number of target, or (optionally) the next item after the target.

Considering the parameters in order, here's what they are for:

The direct parameter <list> is the list of items to search through.

<string> is the target to search for.

Returning... determines whether you want the item number where the target is found to be returned, or instead the next item *after* the item where target is found. Returning the offset is the default. The intended use for this option is in working with association lists (poor-man's records), where you have <key, value> pairs in a list (see the demo Association lists AppleScript in the Demo Scripts folder).

Searching ... determines whether every item is searched or only even or odd items. If you are working with <key, value> pairs, you would only want to search odd items to avoid finding find the target in a value item instead of in a key item. Default is to search every item.

Case ... determines whether case is considered in the search. Sensitive is default.

Exact match... if this is set to true (default) the match must be exact (however, using the _case_ option modifies this behavior), if set to false, it will look for substring matches.

Offset In List (cont.)

The primary use for Offset In List is to easily work with association lists. Association lists are key / value pairs which function more or less like AppleScript™ records. With Offset In List, you can store key / value pairs in a regular AppleScript™ list then use Offset In List to search the keys and return the value for a matched key. Consider the following contrived example of a list of names and phone numbers.

```
set people to {"Stephan", "455-1234", "Renee", "455-4444", "Laura", "433-2345"}
```

When we want to lookup a person's number, we search the keys in the list (the names). Notice that the keys are the odd numbered items of the list: 1, 3, and 5. Here's how to do this with Offset In List:

```
offset in list people of "renee" case insensitive searching odd items with returning next item
```

This statement would locate the key "Renee", and return the next item in the list which is "455-4444". We tell Offset In List to search only odd numbered items, and to ignore the case of the key.

It's a little more elegant to use AppleScript's records for this task, but unfortunately, with AppleScript™ you have to define the record and labels at the script's compile time. Using Offset In List, you don't need to.

An entire Association List script object has been provided in the demos. It contains operations for adding, deleting, and looking up data, plus several more.

Offsets Of

Offsets Of searches through a string or list of strings for a search pattern and returns a list of offsets (indexes into the string) where the string was found.

USAGE:

offsets of: returns a list of offsets of one string in another.

offsets of string -- Search string.
in anything -- string or list of strings to search.
[case sensitive/insensitive] -- defaults to case sensitive.

Result: list -- list of offsets where search string was found.

If the case parameter is not specified it defaults to sensitive; that is, case **is** significant, so "and" is not the same as "AND".

If you pass a single string to search in, the result is a list of the offsets, or an empty list if no matches were found. If you pass a list of strings to be searched, a list of lists of offsets is returned, with some of the lists possibly empty if no matches were made. There are no size limits on the parameters other than available memory.

Tokenize

Tokenize was designed to make it easier to split text into elements based on a set of delimiters. The demo AppleScript™ illustrates several novel uses for Tokenize which may not be obvious at first glance.

USAGE:

tokenize: split text into a list of items based on list of delimiters.

```
tokenize  string  -- the string to tokenize.
        with delimiters  anything  -- the delimiter string(s).
        [null tokens boolean] -- return null tokens (default is true, return them)

Result:   list  -- list of token strings.
```

the direct parameter to tokenize is a string, and the second (required) parameter is a list of strings (each string being one or more bytes in length) to use in tokenizing the direct parameter.

If you are only tokenizing with one delimiter you need not pass it as a list since AppleScript™ will handle the coercion for you. For example, the following is legal:

```
tokenize "My Name Is" with delimiters " Name "
        => {"My", "Is"}
```

Returning null tokens (the optional parameter) means where two or more delimiters are found next to one another, an empty string will be returned indicating there was no token in that location. For example:

```
tokenize “http://www.acme.com/” with delimiters “/”
        =>{"http:", "", "www.acme.com", ""}
```

The second and last items in the result are null tokens. In the first case, Tokenize found two delimiters next to each other, but nothing between them; in the second case no token was found after the delimiter. If you do not want null tokens returned, specify “without null tokens” when calling Tokenize.

Tokenize (cont.)

Some text processing tasks require more than one call to Tokenize to perform. For example, if the variable myText contained a number of lines separated by return characters, and you wanted to retrieve the words from line five, you could write the following AppleScript™ commands:

```
tokenize myText with delimiters {return}
tokenize (item 5 of result) with delimiters {space}
        => [result is a list with all the words from line five of the text]
```

What are tokens? Tokens can be anything which has some particular meaning within a context. The words in this sentence can be considered tokens. Each word has some meaning in the English language. The spaces between the words have no special meaning (for this discussion) except to delimit where the tokens start and stop. If the first sentence of this paragraph is run through Tokenize with a delimiter list consisting of a single space character, and the punctuation mark "?", it would return a list of three items (words): {"What", "are", "tokens"}. This is the process of tokenization.

Here's what that would look like in AppleScript™:

```
tokenize "What are tokens?" with delimiters {space, "?"}
        => {"What", "are", "tokens"}
```

Tokenize lets you specify which patterns to use as token delimiters, then it searches through a piece of text pulling out all the sequences of characters found between the specified tokens. ("space" is an AppleScript™ constant which translates to the space character [' ' or ASCII 32]).

Trim

Trim implements a common text processing command for removing characters or strings from the ends of a block of text. Acme Replace will also remove characters by searching for them and replacing them with nothing, but often times only characters need to be removed from the start of a line or piece of text, or alternately from the end. Using Trim simplifies this process.

USAGE:

```
trim: trim patterns of text off the ends of strings.

trim  [anything]  -- the patterns to trim. (string or list)
        off  anything  -- the items to trim.
        [from  front side/back side/both sides]  -- location to trim from.
        [ignoring  anything]  -- string(s) to ignore.

Result:  anything  -- the trimmed text.
```

Trim accepts a string or list of strings to be trimmed, a string or list of strings specifying what to trim, and optionally lets you specify where to trim from (from the front, back or both sides of the strings), and which pattern(s) should be ignored (a string or list of strings).

There are no size limits on the size of the parameters (except for available memory). If no items are specified for trimming, Trim defaults to removing spaces from *only* the end of the text items, ignoring nothing. Likewise, if the from parameter is not specified, only the end of the string(s) will be trimmed.

To summarize Trim's defaults, only the second parameter is required (the items to trim), and Trim will remove single spaces from only the end of the strings, ignoring nothing.

EXAMPLES:

```
Trim {space} off "    On a clear day you can see forever...    "
   => "    On a clear day you can see forever..."
```

The extra spaces are removed from the end of the string. If however, the string ends with a return character, then we should use the following form:

Trim (cont.)

```
Trim {space} off "  On a clear day you can see forever...     " & return
    => "  On a clear day you can see forever...\r"   -- "\r" is the return character
```

The spaces at the front of the string are left alone because default is to only trim from the end of the string. To also remove leading spaces, use the following command:

```
Trim {space} off "  On a clear day you can see forever...     " & return from both sides
    => "On a clear day you can see forever...\r"   ---- "\r" is the return character
```

The “ignoring” parameter provides a powerful feature most other implementations of Trim do not support. It forces Trim to keep searching for the patterns to trim even though other patterns might be in the way.

The results of specifying the same pattern as a trim argument and also an ignoring argument is undefined.

ACME Parse Args
ACME Lookup Field

ACME Parse Args and Lookup Field are a pair of commands only useful to people writing WWW server CGIs (as this documentation was being prepared, StarNine Technologies' WebSTAR product is the only shipping Macintosh web server). A certain level of familiarity in writing CGIs is assumed for the purposes of explaining the usage. For tutorials or further information on writing WebSTAR CGIs in AppleScript, check Jon Wiederspan's web site at `http://www.comvista.com/`, or StarNine's web site at `http://www.starnine.com/`.

ACME Parse Args and Field Lookup handle the tedious task of converting and parsing the arguments passed to a CGI script. The Parse command converts all "+" characters to a space, and translates sequences of %XX (where XX are hex digits) to the proper character. Additionally, Parse will combine the values for all duplicate field names into a single list.

The ACME Lookup Field command provides a very fast way to access specific field values in the parsed argument list.

USAGE:

ACME parse args string -- The post arguments.
[duplicates combined boolean] -- Combine values of duplicate field labels into a list.
[case sensitive/insensitive] -- Perform case sensitive search? (default is sensitive; i.e., case matters).

Result: list -- List of parsed CGI arguments as field/value pair lists.

ACME lookup field string -- Field name to lookup.
in list of list -- Parsed CGI arguments.
[default value anything] -- The value to return if the label isn't found or the value is empty.
[case sensitive/insensitive] -- Perform case sensitive search? (default is sensitive; i.e., case matters).

Result: anything -- Value(s) of field.

ACME Parse Args
ACME Lookup Field (cont.)

When a CGI or ACGI is called to process an html form, the script is passed the values the user entered on the form in the post_args variable (this is specific to WebSTAR). The data is encoded by translating certain special characters to the pattern %XX, where XX is a hexadecimal number representing the ASCII value of the character, by converting spaces to either "+" or %20 (depending on the browser), and combining all the field names and values from the form into one long string. It could look something like this string:

"name=Jon&age=27&comments=I+take+the+fifth"

Acme Parse Args decodes the input and groups the field names and values together.

```
Acme Parse Args post_args  -- post_args contains form data...
        => { {"name", "Jon"}, {"age", "27"}, {"comments", "I take the fifth"} }
```

Once the input data has been decoded and parsed, you can step through each of the inner lists and process the data, or get the value for a specific field by calling Acme Lookup Field. To get the value the user entered in the "age" field, make the following call:

```
Acme lookup field "age" in theParsedData
        => "27"
```

ACME Parse Args
ACME Lookup Field (cont.)

Combining duplicate fields

It's legal to use the same name for more than one field on an html form, and Acme Parse Args has special features which let you decide how you want the values returned when there are duplicate field names. If you specify the option "duplicates combined true", all values from duplicate label names are grouped together in a list. Using the example above but with the addition of an extra name input field:

"name=Jon&name=Johnson&age=27&comments=I+take+the+fifth"

Acme Parse Args post_args with duplicates combined -- post_args contains form data...
=> **{ {**"name", **{**"Jon", "Johnson"**}}**, **{**"age", "27"**}**, **{**"comments", "I take the fifth"**} }**

Without the duplicates combined parameter, the result would look instead like this:

Acme Parse Args post_args with duplicates combined -- post_args contains form data...
=> **{ {**"name", "Jon"**}**, **{**"name","Johnson"**}** ,**{**"age", "27"**}**, **{**"comments", "I take the fifth"**} }**

Case sensitive/insensitive

When matching label names in either the parse phase, or the lookup phase, case is normally not considered unless you specify the "case sensitive" parameter.

Using Acme Lookup Field

As an alternative to stepping through each of the label/value lists returned from Acme Parse Args, you can quickly get the value of a specific field by passing that field name to Acme Lookup Field (ALF). ALF will return either a single value, or a list of values if 1) there were duplicate field names on the form and, 2) you specified that duplicates should be combined in the same list when parsing the input data.

Acme Sort

Sort a list of items into ascending or descending order, or sort a list of lists using any one of the items in the inner lists as a sort key. Honors AppleScript native data types for text, integers, floating points numbers, and dates.

USAGE:

```
ACME sort list of anything  -- List of things to sort.
        [by item number small integer] -- 1-based index of item to sort
                                        on (1 is default)
        [into ascending order/descending order] -- sort order. Ascending is default.

Result: list  -- sorted list
```

Acme sort can sort either a simple list of items or a list of lists of other items. It's very flexible in allowing different sized lists to be sorted or different data types.

When sorting a list of lists as in {{"Florida", "Georgia", "Texas"}, {"Alaska", "Washington", "Hawaii"}, {"Montana", "Wisconsin", "Nevada", "Arizona"}}, the lists are sorted relative to one another based on the sort key you specify—but each inner list is **not** sorted at the same time, you must perform this explicitly in separate steps (see the Acme sort demo script for an example of sorting the inner lists first). By default, each list is sorted using the first item as the sort key. Using the example list of lists of states above, the second list would be sorted first in the output since "Alaska" sorts before "Florida" and "Montana" (assuming you are sorting into ascending order). To specify a different element as the sort key, use the optional parameter "by item number X" where "X" is the 1-based index of the sort key.

Acme sort can sort lists of lists even when the inner lists contain a different number of elements. In this case, if the sort key specified can not be found for a list, it will sort in front of all the others, as in:

```
ACME sort {{"Sunday", "wednesday"}, {"Tuesday"}} by item number 2
        =>{{"Tuesday"}, {"Sunday", "wednesday"}}
```

The second lists was moved to the front in the output because there did not exist an item in the list whose index was 2.

Acme Sort (cont.)

Acme sort will maintain the data type of list elements whenever it can, however comparing mixed data types (such as strings and integers) is accomplished by coercing the elements to strings. Look at the following example:

```
ACME sort {"foo", 5, "bar", "1"}
```

the second element in the list is the number 5, but the last element is a string representing the number 1. Acme sort will coerce the 5 to a string for the purpose of comparing it to the other elements, however the resulting list will still contain the actual integer 5, and not the string representing 5 (“5”). Since the ASCII characters representing integers sort have lower values then the Alpha characters, the numbers will sort ahead of the strings in the list above. The results of that sort look like this:

{"1", 5, "bar", "foo"}

The 5 retained its data type of integer, but the numbers sorted into correct order, and in front of the alpha strings.

Here’s a slightly more confusing example.

```
ACME sort {455,"454e", 600}
        => returns the error “Can’t make some data into the expected type”

ACME sort {"454e", 455, 600}
        =>{"454e", 455, 600}
```

Using the same list, but with the items in a different ordering, the first example returns an error, while the second returns a result. In this case, Acme sort is using the data type of the first sort key and trying to coerce the other keys to that type. Since “454e” cannot be coerced to an integer, and error is returned. In the second case, the first key is a string, and all the other elements in the list successfully coerce to a string.

If you will be sorting lists using mixed data types, you may want to run a few experiments with sample data to verify exactly how the mixed types will sort relative to one another.

Legal Stuff

THIS SOFTWARE AND DOCUMENTATION IS PROVIDED "AS IS" AND WITHOUT WARRANTIES AS TO PERFORMANCE OR MERCHANTABILITY. THIS PROGRAM IS SOLD WITHOUT ANY EXPRESS OR IMPLIED WARRANTIES WHATSOEVER. BECAUSE OF THE DIVERSITY OF CONDITIONS AND HARDWARE UNDER WHICH THIS PROGRAM MAY BE USED, NO WARRANTY OF FITNESS FOR A PARTICULAR PURPOSE IS OFFERED. THE USER MUST ASSUME THE ENTIRE RISK OF USING THE PROGRAM. ANY LIABILITY OF SELLER OR AUTHOR WILL BE LIMITED EXCLUSIVELY TO REPLACEMENT OR REFUND OF THE PURCHASE PRICE.

Government End Users: If you are acquiring the Software on behalf of any unit or agency of the United States Government, the following provisions apply. The Government agrees:

(i) if the Software is supplied to the Department of Defense (DoD), the Software is classified as "Commercial Computer Software" and the Government is acquiring only "restricted rights" in the Software, and its documentation as that term is defined in Clause 252.227-7013(c)(1) of the DFARS;

and (ii) if the Software is supplied to any unit or agency of the United States Government other than DoD, the Government's rights in the Software, and its documentation will be as defined in Clause 52.227-19(c)(2) of the FAR or, in the case of NASA, in Clause 18-52.227-86(d) of the NASA Supplement to the FAR.

Acme Technologies

Acme Technologies builds innovative tools for AppleScript and Internet servers. We are also available on a contract basis to design and implement custom scripting tools, or to help your engineers support AppleEvents in your products. Write for further information and rates.

Thank you for supporting Acme products!

Acme@kagi.com