NAME

ash – an "AppleScript Shell" for interactive execution of AppleScript commands

SYNOPSIS

ash [<options>] [*filename(s)*]

DESCRIPTION

This ‘‘AppleScript Shell’’ program is intended for interactive use in a manner similar to that of the standard Unix shells. You can execute simple one-line AppleScript commands by just typing them and hitting ’return’. The AppleScript command will be executed immediately.

But many AppleScript commands are multiple lines (e.g. ’tell’ or ’repeat’ commands). For these, the ‘‘AppleScript Shell’’ will go into a mode where it stores your commands until you enter the corresponding ’end’ command at which point your multiple-line AppleScript command will be executed. The prompt will show you what mode you are in.

You can exit this shell at any time by entering the command ‘‘–exit’’. If you want to abort a pending multi-line command without exiting from the shell, use the ‘‘–clear’’ command.

Results of AppleScript commands

The result of the AppleScript command that was last executed will appear in the Terminal window without you having to do anything special. But if you want to output the value of an intermediate AppleScript expression, you can use the ‘‘–echo’’ command. For example:

```
tell application ‘‘Finder’’
    set theSelection to selection
    set n to number of items in theSelection
    –echo ‘‘number of items selected: ’’ & n
    repeat with i from 1 to n
        set theItem to item i of theSelection as alias
        –echo ‘‘item ’’ & i & ‘‘ is ’’ & theItem
    end repeat
end tell
```

This is especially useful when debugging an AppleScript. The ‘‘–echo’’ command is implemented by means of an AppleScript subroutine ‘‘ashEchoSub’’ which is included in the AppleScript before it is executed.

Subroutines and Script Objects

When you define a subroutine (starting with ‘‘on’’ or ‘‘to’’) or a script object (starting with ‘‘script’’), these pieces of code remain active and are available for use in all subsequent commands. I.e. all subroutines and script objects defined in the current session are present in the AppleScript that is executed. The ‘‘–show’’ command will show you what subroutines and script objects have been defined so far.

Since subroutines and script objects don’t affect anything unless they are invoked, you can generally just forget about the extraneous ones. But if you want to clean house just to make things neater, there are a few commands provided for this purpose. The ‘‘–clearSub’’ command will remove the subroutine specified as an argument. For example, if you had previously defined a subroutine named ‘‘foo’’, then ‘‘–clearSub foo’’ would remove it. The ‘‘–clearScript’’ command will remove the script object specified as an argument. For example, if you had previously defined a script object named ‘‘fred’’, then ‘‘–clearScript fred’’ would remove it. The ‘‘–clearAll’’ command will remove all previously defined subroutines, script objects, properties, and variables as well as clearing the current AppleScript.

Variables & Properties

There is only preliminary support for top-level variables or properties. If you define a top-level variable or property, it and its value will remain active and be available for use in all subsequent commands in the same way that subroutines and script objects persist after definition. The ‘‘–clearVar’’ command will remove the variable or property specified as an argument. It is often useful to use ‘‘batch mode’’ (via the ‘‘–batch’’

command) or to use a 'try' block when you are setting some variables at top level that you want to use in some later statement.

Running AppleScript commands in "batch mode"

In the usual mode of operation, each AppleScript command that you enter is executed immediately. (Multi–line commands (e.g. 'tell' or 'repeat') will execute when the corresponding 'end' is entered.) But sometimes you want to enter a bunch of AppleScript commands and then have them all executed at once. The "–batch" command allows you to do this – it starts "batch mode" operation. AppleScript commands issued in this mode are only executed when you leave batch mode via the "–end" command. Unlike the case in normal one-command-at-a-time operation, subroutines and script objects defined in batch mode do not remain active after the end of batch mode.

If you supply a filename argument to the "–batch" command, 'ash' will go into batch mode, source the specified file, then automatically exit from batch mode.

Comments

Any line starting with a hash (#) character is treated as a comment and thus is completely ignored by 'ash'. This parallels the commenting convention of the usual Unix shells like 'bash' and 'tcsh'. Of course the standard AppleScript commenting characters are also supported.

Startup

When 'ash' starts up, it executes the commands in the file ~/.ashrc in the same manner as if these commands had been typed interactively at the command prompt. (In other words, it automatically "sources" the ~/.ashrc file.) For example, if you had the following command in the .ashrc file:

```
say "Welcome to \"ash\" (AppleScript Shell)"
```

then you would get a spoken welcome when you started 'ash'. You can use the .ashrc file to store commonly used abbreviations or to set up AppleScript subroutines, etc. You can use the "–norc" command-line option to prevent the ".ashrc" file from being read at startup.

Getting input from the user

The usual way to get input from the user in an AppleScript is to put up a dialog of some sort. Since the AppleScripts run by 'ash' are in a different environment than usual, using something like "display dialog" would result in an error message saying "no user interaction allowed". In order to sidestep this problem, 'ash' redirects all such user interaction to the Terminal application by prefacing such commands with 'tell application "Terminal" to'. If you are running 'ash' in some other terminal-type application, you will need to change the Perl variable '$terminalAppName' to reflect the name of your app.

An alternative way to get input from the user when running scripts in 'ash' is to use the "–read" command. For example '–read n' will read characters from the keyboard and put them into an AppleScript variable named "n". If you are using this method of getting input from the user, you probably want to use the "–echo" command to display a prompt to tell the user what is expected.

Note on command-line editing

This script uses the Perl module "Term::ReadLine" which supplies facilities for interactive command–lines. The default Perl installation on OS X (as of Tiger) only includes a "stub" version of the facilities used by this module. If you install the module "Term::ReadLine::Perl" (e.g. via CPAN) then you will get command-line editing and command history (via the arrow keys).

OPTIONS

The following options can be specified on the command-line that is used to invoke 'ash':

–nogreeting

Disables the greeting message that is given when you start 'ash'

–quiet
Stops 'ash' from outputing status messages in response to commands. This option also disables the greeting message at startup.

–norc
Prevents the ~/.ashrc file from being read at startup. (This is only useful when running interactively since stand-alone scripts do not read the ~/.ashrc file at startup.)

–oneoff
Puts 'ash' into ''one off'' mode where 'ash' will automatically exit after executing one AppleScript command. The ~/.ashrc file will still be read at startup and AppleScript commands in that file don't count. It is often useful to combine this option with ''–quiet'' (and possibly with ''–norc'') to get a quick, clean way to run a single AppleScript command. This option is ignored when running 'ash' non–interactively.

–trace
Enables ''trace mode'' for the execution of AppleScripts. In this mode, the execution pauses after each AppleScript statement and the result from the previous statement is displayed. Each statement will pause for one second before continuing with the rest of the AppleScript. Pressing any key will stop it from pausing and so if you want it to run freely, just hold a key down. Trace mode is mostly useful when running scripts non–interactively.

–debug *level*
Sets the debugLevel to the specified integer. Higher values result in more debugging messages. Values higher than 1 will not likely be useful to anyone other than the developer. (Default is 0)

–timing *level*
Sets the timingLevel to the specified integer (should be either 0 or 1). If the timingLevel is greater than zero, 'ash' outputs info about the time taken to compile and execute the AppleScript. (Default is 0)

–osaMethod *method*
Specifies which method should be used to compile and execute the AppleScript. Possible values are: macosasimple, macperl, osascript
macosasimple: Uses the Perl module ''Mac::OSA::Simple''
macperl: Uses the Perl module ''Mac::Perl''
osascript: Uses the /usr/bin/osascript tool

Any of the above command-line options can be abbreviated as long as there is no ambiguity. For example, ''–osa'' can be used in place of ''–osaMethod'' since that is the only option that starts with ''–osa''.

If any filenames are specified on the command–line, 'ash' will execute the commands in those files non–interactively. I.e. supplying a file on the command-line is an alternative to inserting a ''shebang'' line and making the script file executable as described in the ''stand–alone scripts'' section.

COMMANDS

There are several special commands (starting with "**–**") that are interpreted by this shell. These commands can be entered at the **ash** prompt when running interactively, or inserted in a file that is run non–interactively. (The reason for the ''–'' at the start of each command name is to ensure that these commands don't collide with some AppleScript syntax.) Even though some of the command names include uppercase characters, the command processing is case–insensitive, so for example you could use ''–clearall'' instead of ''–clearAll''.

–help *[topic]*
If you supply one of the available topic names as an argument, the ''–help'' command will show the help text for that topic, otherwise it will show the ''intro'' section. To see the list of available topics, use ''–help topics''.

–exit
Exits the **ash** shell

–abbrev *[name [commandString]]*
Defines an abbreviation for a command string. For example:

```
-abbrev strack some track of playlist "Library"
```

defines 'strack' as an abbreviation for 'some track of playlist "Library"' so you could then issue the AppleScript command

```
tell application "iTunes" to play strack
```

in order to play a random song from your iTunes library.

You can remind yourself of the definition of the abbreviation named "strack" by entering the command "–abbrev strack". You can see the current list of abbreviations by entering the command **–abbrev** (with nothing following it).

–unabbrev *name*
Removes a previously defined abbreviation.

–echo *expression*
Echos the value of the specified AppleScript *expression*.

–read [*options*] [*varName*]
Reads from the keyboard in the same manner as the 'read' command in Bash. If *varName* is supplied, the characters read are stored in an AppleScript variable of that name. The *options* are the same format as those for the Bash 'read' command. E.g. "–n1" will read one single character (without the need to press Return), "–s" will disable echoing of characters, "–t5" will make it timeout after 5 seconds. Note that unless you use the "–t" option, it will wait indefinitely for the user to enter something.

–source *filename*
You can use the "–source" command to execute the commands that are in a specified file in the same manner as if these commands had been typed interactively at the command prompt. This is another way of saving typing. For example, if you have some commands in the file "~/MyStuff/do_something", you could run those commands via:
–source ~/MyStuff/do_something

Note in particular that any subroutines defined in that file will persist and be available for use in interactive commands. (See: –help subroutines) If you are using the "–source" command to bring in a whole script for executing, you probably want to go into "batch mode" first. As an alternative to going into batch mode, sourcing the script file, then exiting batch mode, you can supply the script filename as an argument to the "–batch" command (e.g. '–batch ~/MyStuff/do_something'). This will go into batch mode, source the specified file, then exit from batch mode automatically.

–batch [*filename*]
Starts "batch mode". If you supply a filename as an argument to the "–batch" command (e.g. '–batch ~/MyStuff/do_something'), 'ash' will go into batch mode, source the specified file, then exit from batch mode automatically.

–end
Ends "batch mode" and executes the pending AppleScript commands.

–show
Displays the text of the current AppleScript (i.e. the text of a partially completed multi-line command or that of the most recently executed command, plus any previously defined subroutines or script objects) This is useful when you want to copy & paste that AppleScript elsewhere, or just to review the commands you have entered and the existing subroutines and script objects.

–editor
Activates Apple's "Script Editor" and creates a new document with the text of the current script.

–rerun
Reruns the current AppleScript.

–clear
Clears the current AppleScript.

–clearSub *subName*
Clears the specified AppleScript subroutine. For example, if you had previously defined a subroutine named ‘‘foo’’, then ‘‘–clearSub foo’’ would remove it.

–clearScript *scriptName*
Clears the specified script object. For example, if you had previously defined a script object named ‘‘fred’’, then ‘‘–clearScript fred’’ would remove it.

–clearVar *varName*
Clears the specified variable or property. For example, if you had previously defined a variable or property named ‘‘x’’, then ‘‘–clearVar x’’ would remove it.

–clearAll
Clears all previously defined subroutines, script objects, variables, and properties as well as clearing the current AppleScript.

–cd *[dirName]*
Changes the current working directory to the directory specified. If no directory is specified, changes to the user’s home directory. If the ‘‘–f’’ option is used (‘‘–cd –f’’), it changes directory to the folder of the frontmost Finder window.

–pwd
Displays the current working directory. (This command is actually just an abbreviation for ‘‘–! pwd’’.)

–ls *[options] [filenames]*
Lists the files of the current directory. (This command is actually just an abbreviation for ‘‘–! ls’’ and so it takes all the usual command-line options for ’ls’.)

–! *command*
Passes the specified *command* to a standard Unix shell for execution. For example, the command ’–! ls’ does the same thing as the ‘‘–ls’’ command. (The ‘‘–ls’’ command is provided just as a convenience.)

–createMan
Creates a ’man’ page file named ‘‘ash.1’’ in the current directory. You will need to move this file to one of the directories in your MANPATH (e.g. move it to /usr/share/man/man1/)

STAND-ALONE SCRIPT FILES

It is also possible to use ’ash’ in a non-interactive way, by specifying it as the ‘‘shebang’’ interpreter in a script file. Using this mechanism, you can create stand-alone script files that can be run like usual Unix scripts. To do this, you save your AppleScript commands (and special ’ash’ commands) in a file and make the first line of that file be the following:

```
#!/usr/bin/env ash
```

(This assumes that ’ash’ is in your shell execution PATH – otherwise you should specify the full path to ’ash’ in that ‘‘shebang’’ line.) Then make the script file executable (using ’chmod +x’) and you will be able to run that script like any other Unix command. Technical note: the reason why you need to use ’/usr/bin/env’ in the ‘‘shebang’’ line is that ’ash’ is itself a script.

When running non–interactively, ’ash’ is effectively in ‘‘batch mode’’. All of the AppleScript commands are sent off for execution at one time.

Note that the ~/.ashrc file is *not* read when running non-interactively (i.e. when running a stand-alone script) and thus the ‘‘–norc’’ command-line option is redundant in this case. If you want to execute the commands from your ~/.ashrc file when running a stand-alone script, you can use the ‘‘–source’’ command to do so.

An alternative way to run script files non-interactively is to specify the filenames on the ’ash’ command–line. For example: ’ash file1 file2’ would non-interactively execute the commands in the files ‘‘file1’’

and ‘‘file2’’.

BUGS

* handling of top-level variables and properties is inadequate

AUTHOR

ash was written by Cameron Hayne (macdev@hayne.net). The initial version was in January 2002.

COPYRIGHT & LICENSE

VERSION

This man page was generated via the ‘‘–createMan’’ command using version 0.60 of **ash**. You can check what version you are using by issuing the ‘‘–help version’’ command. You can get the latest version of **ash** from the web site: http://hayne.net/MacDev/Ash/