NAME

wish − command interpreter and programming shell

SYNOPSIS

wish [options] [script-file]

DESCRIPTION

Wish is a shell (command interpreter) which reads commands from either a file or the terminal and executes them. It executes system commands as well as builtin commands (see below for description). Main features include: a history mechanism, enabling previous commands to be easily redone; a comprehensive command line editor for ease of use from terminal; word completion and option listing on the command line; full input and output redirection and aliases. These features are described in detail below.

This manual describes internal version 41 of Wish 2.0.

Wish is derived from the Clam shell, written by Callum Gibson, and is in some sense a partial rewrite of Clam with some new features added and some old features removed. Many thanks to Callum for allowing me to use his source, and for his suggestions and criticisms during Wish's development.

Basic Command Line Parsing

The command line, whether read from stdin or file, is broken up into *words.* Words are sequences of characters which are separated by word-delimiters (the most obvious example is a *space).* The following characters have effects on the words of the command line and act as word delimiters:

< *filename.* Standard input redirection. Instead of reading input from the terminal, input is read from the named file (the following word). e.g.

```
% prog < inputfile
```

> *filename.* Standard output redirection. Instead of outputting to the terminal screen, output is redirected to the named file. e.g.

```
% prog > outputfile
```

n> *filename.* Other output redirection. Instead of outputting to the terminal screen, output from the given file descriptor is redirected to the named file. e.g.

```
% prog 5> outputfile
```

2> Standard error redirection. The default is the terminal screen, but instead standard error output is directed to the named file. e.g.

```
% program 2> errfile
```

[n]>> *filename.* As above but the named file is appended to, not erased first.

| Pipe output of one program to input of another. A series of programs joined by pipes is called a *pipeline.* A pipe joins the standard output of the first program to the standard input of the second program. e.g.

```
% prog1 | prog2
```

& background. When this symbol appears, it runs the previous command in the background. A number indicating the process id and Wish job number (job control only) is returned and the prompt is given immediately for the next command.

; semicolon is used to separate pipelines. A semicolon can be typed instead of a newline so that a new command or pipeline can begin. The effect is exactly the same as a newline and thus it has the highest priority of these metacharacters.

There are valid combinations of these, too, although it is illegal to redirect output to two places, for example. Some other characters also have special meaning and these may cause the words to be interpreted in different ways. Ordinarily the words become the arguments of an executing program. Most of these "metacharacters" will also be ignored inside double quotes ‘"’ and all (except history substitution commands using !) are ignored inside single quotes ‘’’. (See the section titled "Quoting")

Job Control

If job control is included, Wish allows the user to have control over their processes or *jobs.* Whenever a process is run it is given a job number. This applies to all background jobs and the current foreground one if it exists. This number can then be used to identify a process for use with bg or fg. The list of all currently active jobs can be displayed with the *jobs* command. The *bg* and *fg* commands allow jobs to be placed into the background (where the prompt is returned but the command still runs) or the foreground (where the user is in an interactive state with the process). The stop signal can be generated by typing '^Z' while a foreground job is running. This has the effect of stopping that process immediately, which may be restarted with either the bg or fg command. Note that bg and fg always refer to the current (most recent) job by default. For further explanation, see the "Builtin Commands" section below.

Background jobs are allowed to write to the terminal by default but not read from it. (The shell places a job in the background by disassociating it with the control terminal's process group. See *setpgrp(2)).*

The *stty(1)* "tostop" mode can be used if output from background jobs is to be disallowed. This mode will cause background jobs to stop if a write to the terminal is attempted. Reads from the background are always disallowed. The user is notified of a processes status after return is pressed and before the next prompt appears. This gives information about completion or stoppage of background jobs.

Those people using **Minix** should use *stty(1)* to redefine their 'quit' key from ^\ to ^Z, as there is a primitive form of job control available when *Wish* is compiled with #define V7JOB. This doesn't always work, but most of the time you can ^Z, bg and fg.

Job control under Posix systems is still slightly broken; I haven't had the time to sit down and grok the full implications of the Posix rationale for job control.

Command Line Editing (CLE).

This feature allows the use of control keys and meta-character key sequences to perform editing operations upon the current line. This is done by using "cbreak" terminal mode (see *stty(1)*) which causes an interrupt to be generated after each character is typed and hence available for input, rather than just at the end of a line when return is typed. Wish also turns echoing off so that it can position the characters properly and effect autowrap, control character representation, etc. These two modes are reset when a foreground job is executed. Other modes are settable by the user and not changed by Wish.

European users will be happy to know that the CLE is now 8-bit transparent. For users with 7-bit keyboards, the CLE now has a command to produce 8-bit keystrokes. When 8-bit characters cannot be displayed on the screen, turn the Wish variable 'MSB' on; 8-bit characters will then be displayed in bold.

The Command Line Editor provides many editing commands. The following table lists the possible commands. The three columns hold: the default key binding for the command, the internal key value for the command (usually the same as the key binding), and a brief description of the action. Most actions are obvious; the complex ones are described below.

KEY	VALUE	ACTION
===	=====	======
^@	\000	save current position
^A	\001	goto start of line
^B	\002	go backwards one character
^C	\003	interrupt - discard line and start again
^D	\004	delete current char, list options, or exit shell
^E	\005	goto end of line
^F	\006	go forward one character
^G	\017	kill whole line
^H	\010	back-delete one character
^I TAB	\011	complete current word
^J LF	\012	finish line (same as return)
^K	\013	kill from cursor to end of line
^L FF	\014	clear screen and redisplay line
^M CR	\015	finish line (same as linefeed)

ˆN	\016	step forward in history
ˆO	\017	toggle insert/overwrite mode
ˆP	\020	step backward in history
ˆQ	\021	resume tty output
ˆR	\022	redisplay the line
ˆS	\023	stop tty output
ˆT	\024	transpose the current and previous chars
ˆU	\025	accepts number and key to perform num times
ˆV	\026	next character to be read literally
ˆW	\027	delete word backwards
ˆX	\030	goto saved position
ˆY	\031	yank the previous word into a buffer
ˆZ		suspend process (no effect upon CLE)
ˆ[ESC		prefix for meta-commands
ˆ\	\034	beep
ˆ]	\035	turn CLE mode bit on (see bindings below)
ˆˆ	\036	turn CLE mode bit off (see bindings below)
ˆ_	\037	turn the following character’s msb on
SPC../		insert that character
0..9		insert the character or read digit (ˆU)
:..˜		insert that character
ˆ? DEL		back-delete character (same as ˆH)
ESC-ˆP	\201	match previous partial command
ESC-b		
ESC-B	\202	skip backwards one word
ESC-d		
ESC-D	\203	delete word forwards
ESC-f		
ESC-F	\204	skip forward one word
ESC-h		
ESC-H	\205	get help on word
ESC-p		
ESC-P	\206	insert buffer onto the line
ESC-y		
ESC-Y	\207	yank the next word into a buffer
ESC-/	\210	search forward for next typed character
ESC-?	\211	search backwards for next typed character

Note that some keys which normally generate signals (e.g. ˆZ, ˆY, ˆ\) do not do so whilst a line is being entered/edited. Their signal generating status is returned whenever a foreground job is in progress.

Complex CLE Functions

Some of the CLE commands are complicated. Their full description is given here.

ˆD Delete char, list options, leave shell

This command has three functions. If you use the command at the beginning of a line, you will exit the shell. If your cursor is on top of a character, that character will be deleted. Finally, if your cursor is at the end of the line, a list of ‘expansions’ will be shown. This command will show files, programs you can run, user’s home directories and variables, depending upon how you use it.

For example, to see a list of files in your current directory, you can do

```
% word ˆD
```

To see all the files starting with ‘s’, do

```
% word word sˆD
```

The word can be a relative or absolute path name; to see the files starting with ‘std’ in the include directory, and the parent directory, do

```
% word word /usr/include/std^D
% word word ../std^D
```

If the word is the first on the line, aliases, builtins and programs in your PATH are listed. To see all the commands, and all the commands starting with ‘cl’ do

```
% <space>^D
% cl^D
```

If the word begins with a ‘$’, the command will give a list of variable names; to see all the variable names, and all starting with ‘H’, do

```
% word word $^D
% word word $H^D
```

If the word begins with a ‘~’, the command gives a list of users. Examples:

```
% word word ~^D
% word word ~ro^D
```

Words with ‘/’ signs in them will be expanded internally to their pathname equivalents. The word ‘~#/’ means the list of aliases and builtins. For example, to list the files in fred’s directory, in $HOME, and the aliases and builtins, do

```
% word word ~fred/^D
% word word $HOME/^D
% word word ~#/^D
```

<TAB> Filename Completion

Filename completion is closely associated with the list options function. If you use <TAB> instead of ^D, the word on the command line is completed up to the first duplicate letter. For example, if you have two files ’bigfile.c’ and ‘bigfile.o’ in the current directory, then

```
% ls bi<TAB>        becomes
% ls bigfile.
```

Words with only one match are fully completed. This function understands the same words as the list options functions, so you can complete variables, usernames, programs, and files in directories like

```
% ls $HOME/.cl<TAB>     becomes
% ls $HOME/.wishrc
```

Quoting Commands

The CLE provides commands to quote characters instead of them performing functions. The first is ^V. For example, to insert a ^A on the line without ^A doing its usual action, do

```
% word word ^V^A
```

The ^_ command works exactly the same, but also turns the character’s most significant bit on, so 8-bit keystrokes can be entered from a 7-bit keyboard.

^U Repeat Keystroke

^U is followed by a decimal number (which isn’t displayed), and one character. That character is repeated number times.

ESC-^P Match Partial Command

This command only works for the first word. When the cursor is at the end of the first word, and this command is done, the last command that matched the word is added to the line. For example:

```
% ls -l
% wc file
% echo hello
% lESC-ˆP          becomes
% ls -l
```

Word Commands

The word commands are affected by the Wish variable wordterm. This can be set to holds the characters that ‘end’ a word. If this variable isn’t defined, the CLE assumes it holds the following characters: space, tab, > < | / ; = & and ‘.

Bindings

Wish now distinguishes between the commands available in the CLE and the keystrokes that cause the commands to occur. This allows Wish to have different editor bindings. For example, Wish defaults to an emacs-like binding, as noted above, but can be given vi-like bindings by sourcing the file Vibind.

The command characters given in the above table are used by the Wish **bind** builtin command, which can bind a keystroke sequence to another keystroke sequence. With the default emacs-like style, most of the emacs keys are identical to the internal CLE commands, and only the ESC commands are internally bound by Wish. Setting up another editor style involves the use of the ‘bind’ and ‘unbind’ builtins described in the Builtin Commands section.

The CLE also offers a boolean ‘mode’. When the mode is on, all current key bindings apply. However, when the mode is off, key bindings marked as ‘mode only’ are disabled. This allows the vi-style set of bindings to be used.

The help command calls the system manual command with the word immediately preceding the cursor as the argument.

Variables.

The shell variable facility relies on the following metacharacters:

$ denoting a variable. Wish will interpret $ as a reference to a shell variable. The word immediately adjacent to the ‘$’ is taken as the variable name. This word is delimited by any of the metacharacters above, by quotes, by another ‘$’, or by an array subscript. e.g.

```
% echo $fred
```

This will echo the contents of the variable fred.

= assignment of variables. Variables are assigned simply by stating the name(s) of the variable(s), followed by the equal ‘=’ sign, and then the value(s).

Wish is responsible for two separate sets of variables. The examples given above refer to *internal* shell variables. These variables are immediately accessible to the user and also act as flags for some Wish functions (e.g. HASH,ignoreeof,UNIV). The other set is known as *environment* variables. Environment variables are passed to any new process by the execve system call. When the shell is started all environment variables are loaded into the shell’s internal variable storage space. When this initialisation occurs, these variables are automatically marked for *export* , meaning that when execution of a command takes place, the variables marked for export form the environment for the executed command. It is also possible to mark any variable for export by using the *export* command described below.

Wish also has several inbuilt variables:

$$ - The process-id of the process
$? - The exit status of the last process
$# - The number of arguments to the process
$n - (where n is a digit) The n’th argument to the process.
$* - All the arguments to the process (except $0)

When used interactively, these refer to the Wish itself. However, in aliases and Wish scripts, these variables refer to the alias or script. For example, if ‘prog’ is a Wish script, and you type

```
% prog foo bar
```

then from within "prog", $0 will be "prog", $1 will be "foo" and $2 will be "bar". Similarly, an alias such as

```
% alias l ’/bin/ls -lg $*’
```

will do ’/bin/ls -lg’ on all of its arguments.

History and History Substitution.

The history mechanism enables every command typed to be stored and accessed at a later time. To get a list of previous commands use the *history* command. Wish does not enter the *history* command into the history, nor does it store single letter lines since these can be more easily retyped than by using history substitution (described below). The number of commands remembered by Wish is set by the internal "history" variable.

It is possible to recall a previous line(s) from the history by using the metacharacter ‘!’, followed by the absolute history number, a relative history number, a string which matches the most recent command starting with the same string, or another ‘!’ which refers to the previous command. e.g.

```
% !12 fred
```

This will substitute the history event numbered 12 onto the line at that position and the argument "fred" will remain as an argument to that command.

```
% !-3
```

This will substitute the event from three commands ago. It is still possible to have an argument or a different substitution on the same line.

```
% !fo
```

This will substitute the most recent command starting with "fo".

You can also append characters onto a previous history command rather than adding a new word. If we issued a command "ls -l" and then used "!la" the shell would search for a command starting with "la" instead of finding "ls -l" and appending the "a". So, to achieve this, simply quote the characters that are to be appended (actually you only need quote the first). Hence,

```
% !l"a"
```

will give the desired substitution of "ls -la". Spaces may be included in the string by using backslash or quotes.

```
% !"v d"
```

Filename Substitution.

Rather than type a lot of filenames, sometimes it is possible to specify a group of files using the "globbing" technique. This means that using the following metacharacters, you can specify files in any directory using a sort of shorthand notation. The shell then performs what is called "pattern matching" to produce the file names wanted.

* represents any number of characters including zero.
? represents any one character.
[] encloses a list of characters that can match. Two characters may also have a hyphen separating them meaning "up to". So [A-Za-z] includes the letters ’A’ to ’Z’ and ’a’ to ’z’.

e.g. ab* matches all files in current directory starting with "ab".
/usr/bin/lo*.? matches all files in /usr/bin starting with "lo" then ending with a "." and any other character.
[abc]*.c matches all the C-files in the current directory starting with an "a", "b", or "c".

It is also possible to specify *subdirectories* using pattern matching e.g. */*.c matches all C-files in any subdirectory.

A further abbreviation method is available for referring to the home directories of users. This is done by typing a ‘˜’ followed by a *username* or if no username is given then the home directory of the user running the shell is used (i.e. ˜ refers to your directory).
e.g. "˜cgibson" will be replaced by "/u1/hons/cgibson"
"˜mike/work" will be replaced by "/u1/staff/mike/work".

Wish now fully copes with systems that uses yypasswd.

Quoting.

There are three types of quotes (not including backquotes) which may be used to stop the above mentioned metacharacters from being interpreted. Single quotes '' will stop all interpretation except history substitution (!). The last set of quotes is double quotes. They allow history and variable substitutions but disallow pattern matching. Note that quotes may be nested but only the outer set have any effect.

Aliases - User defined Command Sequences

It is possible for the user to define their own set of aliases. Unlike *Csh(1)* where the first word of any command is checked to see if it's an alias, and then the alias is substituted onto the line, Wish allows full shell scripts to be defined which are then executed by the shell after parsing the line. These aliases are stored internally and executed in a similar way as a Wish script. Thus it is possible to pass arguments as $1 $2 etc. or $* for all arguments. In fact, the aliases should behave exactly as a script stored in the user's directory but without taking up directory space. Remember that the more aliases that exist, the more slowly commands will run. Because of the way that these aliases are implemented, it is easy to call builtins, system commands, and even other aliases, from within an alias. You should be careful not to call an alias from within itself since this will cause an error. Aliases can be created using the builtin command *alias* , see below in Builtin Commands section.

Builtin Commands

This section describes the commands that are builtin to the shell. Builtins are only executed in a subshell if they are at the start or in the middle of a pipeline or run in the background.

This section is divided up into several subsections, because some builtins can be left out if they cannot be used or are not needed. To find out which builtins you have, do

```
% ˜#/ˆD
```

This gives you a list of aliases and builtins.

1. Standard Builtins

alias [[-e -s] <alias-name> [<alias-defn>]]

When invoked with no argument this command lists the aliases currently defined. The -l flag instructs Wish to load up the alias from the named file (the alias is given the same name). In the same way -s causes Wish to save the alias to file. If none of these flags appear, then the first argument is taken as the alias name, and the remaining arguments as the alias definition. If there are no other arguments, that alias's definition is printed.

bind [[-m] keysequence [newkeysequence]]

This command allows the user to bind a series of keystrokes to a new series of keystrokes. Note that most ASCII control characters (\000 to \037), and some other control characters (\201 to \211) cause the CLE to perform a function, so most bindings are to one or more of these characters. With no arguments, bind shows the current set of key bindings. With a keysequence argument, the binding of the given sequence is shown. With a keysequence and a value, the sequence is bound to the action. For example:

```
% bind 'ˆ[[A' ˆP
```

binds the up-arrow key to the 'step backward in history' action. Bind also understands C-like octal sequences, so that

```
% bind 'ˆ[[A'   01hello  05
```

will cause the up-arrow to insert the word 'hello' at the beginning of the line.

If a keysequence is bound using the '-m' option, then the binding is marked 'mode-only', and is only invoked when the CLE is in MODEON mode. Otherwise it is ignored.

Note that the CLE uses the first match found when parsing keyboard input. For example, if the current bindings are:

```
% bind
^[[A   is bound to ^P
^[[B   is bound to ^N
^[[Aa  is bound to ^A
^[[B   is bound to ^E
```

then the keyboard input ‘^[[Aa’ will do the command ^P (i.e. go back in history), and then add an ‘a’ to the command line; similarly, the input ’^[[B’ will do the command ^E (i.e. go to the end of the line).

Key bindings can be recursive. A large error will occur if you do

```
% bind a aa
```

as the next time you type ‘a’, it will become an infinite number of a’s, up to the size of the CLE line buffer.

cd [<directory>] or chdir [<directory>]
: This command changes the current working directory of the current shell process. Thus, if cd occurs inside a child process, when that process dies, the parent will not have changed directory. Cd takes a single argument which is the name of the directory to change to. The directory name may be a full path name starting from the root directory or a path name relative to the current directory. The ‘˜’ notation can also be used to refer to the home directory of users.

echo [-n] <arguments>
: Echo writes the given arguments out on a single line on standard output. A newline is appended unless the -n option is given.

export [[-] <variable>]
: mark a variable for export, or if ‘-’ flag given, unmark export ability for that variable. When a variable is marked for export, it means that the internal copy of the variable is transferred to the environment whenever a program is executed, making the new copy available to the new process.

history [m[-n]]
: Lists the previously executed commands remembered by the shell. An optional range can be specified which will list the commands numbered from *m* to *n.* A single number argument will display the m most recent commands.

list [env]
: List the internal variables defined in Wish. With the "env" option, the environment variables are listed (like *printenv(1)).*

source <script-name> [<argument-list>]
: Cause the Wish script to be executed in the current shell, as if the contents of the file had been typed from the keyboard. This is useful if the environment variables are to be set for the shell. (e.g. if you have a script that changes your TERMCAP or PRINTER variable etc.) This script could also be loaded as an alias with "alias -l".

tilde [-l], tilde [[shorthand] [dirname]]
: Wish keeps an internal list of shorthand names for directories. Tilde adds the given shorthand/dirname pair to the list. From then on, you can use ˜shorthand to mean the dirname. If just shorthand is given, the expansion for that shorthand is given. With no arguments, all the internal pairs are given. The -l argument will also show the username/directory pairs from the /etc/passwd file.

unalias <alias-name> ...
: Delete the named alias if it exists. If not an error message is returned.

unbind keysequence
: Unbinds the given key sequence, if that sequence has been bound.

untilde <shorthand>
: Removes the shorthand name/directory pair from the internal list of directory shorthands. unset <variable> ... Delete the internal variable named. If an environment variable of the same name exists, it will be deleted.

unsetenv <variable> ...

Delete the environment variable named. This command should be used with caution since some system programs use the environment variables. The shell with still have an non-exported copy of the variable.

2. Job Control Builtins

bg [%<job-no>] [<pid>]

This command has the effect of starting the current job (which will be stopped) in the background if given no arguments. If given arguments it starts the specified jobs in the background. The arguments are simply the job numbers which have been assigned by the shell. These can be obtained using the *jobs* command.

fg [%<job-no>] [<pid>]

Cause the current job to execute in the foreground whether stopped or executing in the background. If an argument is supplied it is taken as the job number (as described in *bg* and Job Control section).

jobs

Give a list of current jobs (processes) recognised by Wish. These jobs are the either running in the background or are stopped. The shell cannot keep track of processes that result from a *fork(2).* In the att universe, job control is severely restricted. (See Job Control section).

Special Variables.

This section contains a description of the variables which are used by the shell to indicate certain conditions or which actually affect the running of the shell in some way. The user may set these variables to suit their own personal tastes.

KEEPSTTY

This variable affects the state of the terminal under Wish. If not defined, you can change the terminal state using stty, and Wish will use the new state, with the exception of cbreak/cooked modes. with KEEPSTTY, Wish will always reset the terminal to the initial state. This is useful if you have a program with sometimes crashes, leaving the terminal in nl state or even changing the speed. Defining this variable will set the terminal back to a sane state when you return to Wish.

MSB

This variable affects the way characters with their msb on are printed. Wish defaults to printing characters as they are. However, if MSB is set to any value, characters above 127 are AND-ed back to 0-127, and are printed in bold.

beep

This variable can be set to any string of characters which will be output whenever a bell would normally sound. By default, this value is ASCII 7 (ˆG). If set to null, the shell will give no warning bell. (For terminals with inverse video, "ˆ[[?5hˆ[[?5l" will produce a flash).

cwd

Holds the value of the current working directory. This variable is automatically updated by the shell whenever a cd or chdir occurs. It generally should not be changed by the user.

prompt

Prompt contains the format description for the user prompt. By default this is a '%' sign. Prompt2 contains the format description for the second prompt, given when more information is needed, like in a while, case, or if statement being executed in interactive mode (i.e. from keyboard). The format description is simply a string of characters which are displayed, except that some special attributes may be entered using the '%' metacharacter.

%% a percent '%' sign.
%d current working directory.
%!
%h current history number.
%S start standout mode (see *termcap(4)).*
%s stop standout mode.
%@

%t time in 24-hour format with seconds.

If any other character follows the '%' then the '%' and then that character is printed. Other characters are just printed. e.g. "%d[%h] " could produce "/u1/staff/mike/work[58]" as the prompt. This prompt format is similar to that used by *tcsh(1).*

wordterm

This variable holds all the characters that delimit words, and is used in the word-oriented CLE functions. If undefined, the CLE will use the characters SPACE, TAB, > < | / ; = & and ‘.

AUTHORS

Callum Gibson- Honours project 1988. Warren Toomey.

FILES

˜/.wishrc	Read at beginning of execution of Wish.

LIMITATIONS

Maximum line length is 2048 (but redefinable). Maximum number of arguments is 512. Not all of the above features exist, yet, and some new ones may be added in future versions.

SEE ALSO

chmod(1), csh(1), sh(1), stty(1), tcsh(1), exec(2), fork(2), pipe(2), umask(2), wait(2), string(3), signal(3), termcap(4), tty(4), environ(7).