# NAME

ish - a shell (command interpreter) with a csh-like syntax and advanced interactive features.

# SYNOPSIS

**ish**

# DESCRIPTION

*ish*, a shell, is a command interpreter with a syntax similar to csh.

## Initialization and Termination

When first started, *ish* performs commands from the file ~/*.ishrc*, provided that it is readable. Typically, the ~/*.ishrc* file contains commands to specify the terminal type and environment.

## Interactive Operation

After startup processing, an interactive ish shell begins reading commands from the terminal, prompting with *hostname*%. The shell then repeatedly performs the following actions: a line of command input is read and broken into *words*; this sequence of words is parsed (as described under **USAGE**); and the shell executes each command in the current line.

# USAGE

## Lexical Structure

The shell splits input lines into words separated by spaces or tabs, with the following exceptions:

- The special characters '&', '|', '<', '>', and ';' form separate words. The following sequences of special characters form single words: '>>', '|&', '>&', and '>>&'.
- Special characters preceded by a backslash '\'character prevents the shell from interpreting them as special characters.
- Strings enclosed in double quotes (") or quotes (') form part or all of a single word. Special characters inside of strings do not form separate words.

## Command Line Parsing

A *simple command* is a sequence of words, the first of which specifies the command to be executed. A *pipeline* is a sequence of one or more simple commands separated by | or |&. With '|', the standard output of the preceding command is redirected to the standard input of the command that follows. With '|&', both the standard error and the standard output are redirected through the pipeline.

A *list* is a sequence of one or more pipelines separated by ';' or '&'. These separators have the following meanings:

; Causes sequential execution of the preceding pipeline. The shell waits for the pipeline to finish. A newline character following a pipeline behaves the same as a ';'.

& Causes asynchronous execution of the preceding pipeline. The shell does not wait for the pipeline to finish; instead, it displays the job number (see **Job Control**) and associated process IDs, and begins processing the next pipeline (prompting if necessary).

## I/O Redirection

The following metacharacters indicate that the subsequent word is the name of a file to which the command's standard input, standard output, or standard error is redirected.

- < Redirect the standard input.
- >, >& Redirect the standard output to a file. If the file does not exist, it is created. If it does exist, it is overwritten; its previous contents are lost. The '&' form redirects both standard output and the standard error to the file.
- >>, >>& Append the standard output. Like '>', but places output at the end of the file rather than overwriting it. The '&' form appends both the standard error and standard output to the file.

## Command Execution

If the command is an *ish* shell built-in, the shell executes it directly. Otherwise, the shell searches for a file by that name with execute access. If the command-name contains a '/', the shell takes it as a pathname, and searches for it. If the command-name does not contain a '/', the shell attempts to resolve it to a pathname, searching each directory in the PATH variable for the command.

When a file is found that has proper execute permissions, the shell forks a new process and passes it, along with its arguments, to the OS using the `execve`(2V) system call (you must use this system call). The OS then attempts to overlay the new process with the desired program. If the file is an executable binary the OS succeeds, and begins executing the new process.

If the file does not have execute permissions, or if the pathname matches a directory, a "permission denied" message is displayed. If the pathname cannot be resolved a "command not found" message is displayed. If either of these errors occur with any component of a pipeline the entire pipeline is aborted, although some of the components of the pipeline may have already started running.

A pipeline is completed (i.e. returns to the prompt), only when all the commands that form a part of the pipeline and that are being executed in the foreground are completed.

## Environment Variables

Environment variables may be accessed via the **setenv** and **unsetenv** built-in commands. Initially no environment variables are set, i.e. *ish* does not inherit environment variables from its parent. *Ish* maintains environment variables internally, and must not use the C library routines `putenv` or `getenv`. When a program is exec'ed the environment variables are passed as parameters to `execve`. The only environment variable that *ish* interprets is PATH; all other environment variables can be set and unset in *ish* using the above builtin commands, but are not interpreted.

## Signal Handling

The shell normally ignores QUIT signals. Background jobs are immune to signals generated from the keyboard, including hangups (HUP). Other signals have the values that *ish* inherited from its environment. Shells catch the TERM signal.

## Job Control

The shell associates a numbered *job* with each command sequence, to keep track of those commands that are running in the background or have been stopped with TSTP signals (typically CTRL-Z). Jobs are put into the foreground using the `tcsetpgrp` system call. When a command is started in the background using the '&' metacharacter, the shell displays a line with the job number in brackets, and a list of associated process numbers; e.g.,

[1] 1234

To see the current list of jobs, use the **jobs** built-in command. The job most recently stopped (or put into the background if none are stopped) is referred to as the *current* job.

To manipulate jobs, refer to the built-in commands **bg**, **fg**, and **kill**.

A reference to a job begins with a '%'. Refer to job number *j* as in: 'kill %*j*'.

A job running in the background stops when it attempts to read from the terminal. Background jobs can normally produce output, but this can be suppressed using the 'stty tostop' command.

### Status Reporting

While running interactively, the shell tracks the status of each job and reports whenever it finishes or becomes blocked. Status only need be reported prior to printing the prompt; this means that job status may be polled via `wait`. Do not use signal handlers to track job status.

### Built-In Commands

Built-in commands are executed within *ish*. If a built-in command occurs as any component of a pipeline except the last, it is executed in a subshell.

| | |
|---|---|
| **bg** %*job* ... | Run the specified *job* in the background. |
| **cd** [ *dir* ] | Change the shell's working directory to directory *dir*. If no argument is given, change to the home directory of the user. |
| **exit** | Causes *ish* to exit. |
| **fg** %*job* | Bring the specified *job* into the foreground. |
| **jobs** | List the active jobs under job control. |
| **kill** %*job* ... | Send the TERM (terminate) signal to the *job* indicated. To insure termination, the job is also sent a CONT (continue) signal. |
| **setenv** [ *VAR* [ *word* ] ] | With no arguments, **setenv** displays all environment variables. With the *VAR* argument, it sets the environment variable *VAR* to have an empty (null) value. (By convention, environment variables are normally given upper-case names.) With both *VAR* and *word* arguments, setenv sets the named environment variable to the value *word*, which must be either a single word or a quoted string. |
| **unsetenv** *VAR* | Remove *VAR* from the environment. |

## FILES

| | |
|---|---|
| *~/.ishrc* | Read at beginning of execution by each shell. |

## LIMITATIONS

Words can be no longer than 1024 characters. The system limits argument lists to 1,048,576 characters.