NAME

ish — an interactive shell (command interpreter) with a syntax similar to **csh**.

SYNOPSIS

ish

DESCRIPTION

The interactive shell **ish** is a command interpreter similar to **csh**. It provides I/O redirection, pipes, command aliasing, and job control.

Initialization and Termination

When first started, **ish** performs commands from the file **~/.ishrc**, if there is such a file and it is readable. Typically, the **~/.ishrc** file contains commands to specify the terminal type and environment, such as the search path. (As described below, initially *no* environment variables are set for **ish**.)

Execution of **ish** is terminated by the **quit** built-in command or the TERM signal.

Interactive Operation

After startup processing, **ish** writes a **hostname%** prompt and begins reading commands from the terminal. The shell repeatedly performs the following actions: read a line of input, break the line into *words*, parse the sequence of words, execute each command in the current line, print any job control messages, then print the **hostname%** prompt. See below for details on these actions.

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 also form separate words: **>>**, **|&**, **>&**, and **>>&**.
- Special characters preceded by a backslash **** character are not treated as special characters.
- Strings enclosed in double quotes **"..."** or single quotes **'...'** form part or all of a single word. Special characters inside of strings do *not* form separate words.

A *simple command* is a sequence of words, the first of which specifies the command to be executed. You may assume that each simple command has the form "command arguments redirection" and that the syntax "command redirection arguments" is illegal.

A *pipeline* is a sequence of one or more simple commands separated by **|**. With **|** the standard output of the preceding command is redirected to the standard input of the next command. 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 have the 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 semicolon character.

& Causes asynchronous execution of the preceding pipeline. The shell does not wait for the pipeline to finish; instead it displays the job number and associated process IDs (as described under Job Control), and then begins processing the next pipeline.

I/O Redirection

The following special characters indicate that the subsequent word is the name of a file from which or to which to redirect the command's standard input, standard output, or standard error:

- **<** Redirect the standard input from a file.
- **>** Redirect the standard output to a file. If the file does not exist, it is created. If it does exist, it is overwritten.
- **>&** Same as **>**, but redirects both the standard output and the standard error to a file.
- **>>** Append the standard output to a file. Like **>** but places the output at the end of the file rather than overwriting the file.
- **>>&** Same as **>>** but appends both the standard output and standard error to a file.

Alias Substitution

After parsing the command line—but before executing it—**ish** substitutes all aliases that occur in the command line (see Built-In Commands). Only one level of substitution is performed; i.e., if an alias uses an alias, the second alias is ignored.

Command Execution

If the command is an **ish** shell built-in, the shell executes it directly. Otherwise, **ish** searches for a file by that name with execute access. If the command name starts with 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 by searching each directory in the PATH environment variable.

When a file is found that has the proper execute permissions, the shell forks a new process and passes the command, along with its arguments, to the OS using the **execve** 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 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. (Please use the *same* format as **csh** for these error messages.) If either of these errors occurs within any component of a pipeline, the entire pipeline is aborted, even if some of the elements of the pipeline have already been started.

A pipeline is completed and returns to the prompt only when all its components that are executing in the foreground have completed.

Environment Variables

Environment variables may be accessed by the **setenv** and **unsetenv** built-in commands. Initially no environment variables are set; i.e., **ish** does *not* inherit environment variables from its parent. (Your shell must maintain environment variables internally; it may not use the C library routines **putenv** or **getenv**.) When a program is exec'd, the environment variables are passed as parameters to **execve**. The only environment variable that **ish** needs to interpret is PATH; all others can be set and unset in **ish** using the **setenv** and **unsetenv** built-in commands, but they are not interpreted.

Signal Handling

Shells catch the TERM signal. They normally ignore QUIT signals (but commands probably catch them). Background jobs are immune to signals generated from the keyboard, including hangups (HUP). Other signals have the values that **ish** inherits from its environment.

Job Control

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

```
[1] 1234
```

To see the current list of jobs, use the **jobs** built-in command.

Jobs are manipulated using the built-in commands **bg**, **fg**, and **kill**. A reference to a job begins with **%**, as in **%1234**.

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

Status Reporting

The shell keeps track of the status of each job and reports when it has been placed in the background and when it has finished. Changes of status need only be reported prior to printing the prompt. This means that job status can be checked by using the **waitpid** system call. Do not use signal handlers to keep track of job status.

Built-In Commands

Built-in commands are executed directly in the shell. However, if a built-in command occurs in a pipeline as other than the last component, then it is executed in a subshell.

cd [*dir*]	Change the shell's working directory to directory *dir*. If no argument is given, change to the home directory of the user.
bg [**%***job*]	With no arguments, run the most recently stopped job, if any, in the background. With an argument, run the specified job in the background.
fg **%***job*	Bring the specified job to the foreground.
jobs	List the active jobs under job control.
kill **%***job* ...	Send the TERM (terminate) signal to the indicated job(s). To ensure termination, also send the CONT (continue) signal.
setenv [VAR [*word*]]	With no arguments, display all environment variables. With the VAR argument, set 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, set 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
alias [*word1* [*word2*]]	With no arguments, display all current aliases. With one argument, display the alias for *word1* (if any). With two arguments, establish an alias so that *word1* means *word2* in subsequent command lines.
unalias *word*	Remove the alias for *word*, if there is one.
quit	Terminate **ish**.

FILES

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

LIMITATIONS

Words can be no longer than 1024 characters.