

NAME

sh – shell (command interpreter)

SYNOPSIS

```
sh [ -t ] [ -c ] [ name [ arg1 ... [ arg9 ] ] ]
```

DESCRIPTION

Sh is the standard command interpreter. It is the program which reads and arranges the execution of the command lines typed by most users. It may itself be called as a command to interpret files of commands. Before discussing the arguments to the Shell used as a command, the structure of command lines themselves will be given.

Commands. Each command is a sequence of non-blank command arguments separated by blanks. The first argument specifies the name of a command to be executed. Except for certain types of special arguments discussed below, the arguments other than the command name are passed without interpretation to the invoked command.

If the first argument is the name of an executable file, it is invoked; otherwise the string `‘/bin/’` is prepended to the argument. (In this way most standard commands, which reside in `‘/bin/’`, are found.) If no such command is found, the string `‘/usr/’` is further prepended (to give `‘/usr/bin/command’`) and another attempt is made to execute the resulting file. (Certain lesser-used commands live in `‘/usr/bin/’`.)

If a non-directory file has executable mode, but not the form of an executable program (does not begin with the proper magic number) then it is assumed to be an ASCII file of commands and a new Shell is created to execute it. See “Argument passing” below.

If the file cannot be found, a diagnostic is printed.

Command lines. One or more commands separated by `‘|’` or `‘^’` constitute a chain of *filters*. The standard output of each command but the last is taken as the standard input of the next command. Each command is run as a separate process, connected by pipes (see *pipe(II)*) to its neighbors. A command line contained in parentheses `‘()’` may appear in place of a simple command as a filter.

A *command line* consists of one or more pipelines separated, and perhaps terminated by `‘;’` or `‘&’`. The semicolon designates sequential execution. The ampersand causes the preceding pipeline to be executed without waiting for it to finish. The process id of such a pipeline is reported, so that it may be used if necessary for a subsequent *kill*.

Termination Reporting. If a command (not followed by `‘&’`) terminates abnormally, a message is printed. (All terminations other than `exit` and `interrupt` are considered abnormal.) Termination reports for commands followed by `‘&’` are given upon receipt of the first command subsequent to the termination of the command, or when a `wait` is executed. The following is a list of the abnormal termination messages:

- Bus error
- Trace/BPT trap
- Illegal instruction
- IOT trap
- EMT trap
- Bad system call
- Quit
- Floating exception
- Memory violation
- Killed
- Broken Pipe

If a core image is produced, `‘ --- Core dumped’` is appended to the appropriate message.

Redirection of I/O. There are three character sequences that cause the immediately following string to be interpreted as a special argument to the Shell itself. Such an argument may appear anywhere among the arguments of a simple command, or before or after a parenthesized command list, and is associated with that command or command list.

An argument of the form '<arg' causes the file 'arg' to be used as the standard input (file descriptor 0) of the associated command.

An argument of the form '>arg' causes file 'arg' to be used as the standard output (file descriptor 1) for the associated command. 'Arg' is created if it did not exist, and in any case is truncated at the outset.

An argument of the form '>>arg' causes file 'arg' to be used as the standard output for the associated command. If 'arg' did not exist, it is created; if it did exist, the command output is appended to the file.

For example, either of the command lines

```
ls >junk; cat tail >>junk
( ls; cat tail ) >junk
```

creates, on file 'junk', a listing of the working directory, followed immediately by the contents of file 'tail'.

Either of the constructs '>arg' or '>>arg' associated with any but the last command of a pipeline is ineffectual, as is '<arg' in any but the first.

In commands called by the Shell, file descriptor 2 refers to the standard output of the Shell before any redirection. Thus filters may write diagnostics to a location where they have a chance to be seen.

Generation of argument lists. If any argument contains any of the characters '?', '*', or '[', it is treated specially as follows. The current directory is searched for files which *match* the given argument.

The character '*' in an argument matches any string of characters in a file name (including the null string).

The character '?' matches any single character in a file name.

Square brackets '[...]' specify a class of characters which matches any single file-name character in the class. Within the brackets, each ordinary character is taken to be a member of the class. A pair of characters separated by '-' places in the class each character lexically greater than or equal to the first and less than or equal to the second member of the pair.

Other characters match only the same character in the file name.

For example, '*' matches all file names; '?' matches all one-character file names; '[ab]*.s' matches all file names beginning with 'a' or 'b' and ending with '.s'; '?[zi-m]' matches all two-character file names ending with 'z' or the letters 'i' through 'm'.

If the argument with '*' or '?' also contains a '/', a slightly different procedure is used: instead of the current directory, the directory used is the one obtained by taking the argument up to the last '/' before a '*' or '?'. The matching process matches the remainder of the argument after this '/' against the files in the derived directory. For example: '/usr/dmr/a*.s' matches all files in directory '/usr/dmr' which begin with 'a' and end with '.s'.

In any event, a list of names is obtained which match the argument. This list is sorted into alphabetical order, and the resulting sequence of arguments replaces the single argument containing the '*', '[', or '?'. The same process is carried out for each argument (the resulting lists are *not* merged) and finally the command is called with the resulting list of arguments.

Quoting. The character '\' causes the immediately following character to lose any special meaning it may have to the Shell; in this way '<', '>', and other characters meaningful to the Shell may be passed as part of arguments. A special case of this feature allows the continuation of commands onto more than one line: a new-line preceded by '\' is translated into a blank.

Sequences of characters enclosed in double (") or single (') quotes are also taken literally. For example:

```
ls | pr -h "My directory"
```

causes a directory listing to be produced by *ls*, and passed on to *pr* to be printed with the heading 'My directory'. Quotes permit the inclusion of blanks in the heading, which is a single argument to *pr*.

Argument passing. When the Shell is invoked as a command, it has additional string processing capabilities. Recall that the form in which the Shell is invoked is

```
sh [ name [ arg1 ... [ arg9 ] ] ]
```

The *name* is the name of a file which is read and interpreted. If not given, this subinstance of the Shell continues to read the standard input file.

In command lines in the file (not in command input), character sequences of the form '\$n', where *n* is a digit, are replaced by the *n*th argument to the invocation of the Shell (argn). '\$0' is replaced by *name*.

The argument '-t', used alone, causes *sh* to read the standard input for a single line, execute it as a command, and then exit. This facility replaces the older 'mini-shell'. It is useful for interactive programs which allow users to execute system commands.

The argument '-c' (used with one following argument) causes the next argument to be taken as a command line and executed. No new-line need be present, but new-line characters are treated appropriately. This facility is useful as an alternative to '-t' where the caller has already read some of the characters of the command to be executed.

End of file. An end-of-file in the Shell's input causes it to exit. A side effect of this fact means that the way to log out from UNIX is to type an EOT.

Special commands. The following commands are treated specially by the Shell.

chdir is done without spawning a new process by executing *chdir*(II).

login is done by executing /bin/login without creating a new process.

wait is done without spawning a new process by executing *wait*(II).

shift is done by manipulating the arguments to the Shell.

: is simply ignored.

Command file errors; interrupts. Any Shell-detected error, or an interrupt signal, during the execution of a command file causes the Shell to cease execution of that file.

Processes that are created with '&' ignore interrupts. Also if such a process has not redirected its input with a '<', its input is automatically redirected to the zero length file /dev/null.

FILES

/etc/glob, which interprets '*', '?', and '['.

/dev/null as a source of end-of-file.

SEE ALSO

'The UNIX Time-Sharing System', CACM, July, 1974, which gives the theory of operation of the Shell.
chdir(I), login(I), shift(I), wait(I)

BUGS

There is no way to redirect the diagnostic output.