NAME

sh − shell (command interpreter)

SYNOPSIS

sh [**−v**] [−] [**−ct**] [name [arg1 ...]]

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 when it is 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.

By default, 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 command name contains a ‘‘/’’, it is invoked as is, and no prepending ever occurs. This standard command search sequence may be changed by the user. See the description of the Shell variable ‘‘$p’’ below.

If a non-directory file exists that matches the command name and 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*, or a *pipeline.* 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 ‘‘&’’, or separated by ‘‘||’’ or ‘‘&&’’. The semicolon designates sequential execution. The ampersand causes the following pipeline to be executed without waiting for the preceding pipeline to finish. The process id of the preceding pipeline is reported, so that it may be used if necessary for a subsequent *wait* or *kill.* A pipeline following ‘‘&&’’ is executed only if the preceding pipeline completed successfully (exit code zero), while that following ‘‘||’’ is executed only if the preceding one did *not* execute successfully (exit code non-zero). The exit code tested is that of the last command in the pipeline. The ‘‘&&’’ operator has higher precedence.

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
Alarm clock
Terminated

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 follo wing 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 regardless of any redirection of standard output. Thus filters may write diagnostics to a location where they have a chance to be seen.

A redirection of the form ‘‘<−−’’ requests input from the standard input that existed when the instance of the Shell was created. This permits a command file to be treated as a filter. The procedure ‘‘lower’’ could be used in a pipeline to convert characters to lower case:

```
tr "[A−Z]" "[a−z]" <−−
```

A typical invocation might be:

```
reform −8 −c <prnt0 | lower >prnt0a
```

Generation of argument lists. If an y argument contains any of the characters ‘‘?’’, ‘‘*’’ or ‘‘[’’, it is treated specially as follows. The current directory is searched for files which*matc h* 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 non-null 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.

If an argument starts with ‘‘*’’, ‘‘?’’, or ‘‘[’’, that argument will not match any file name that starts with ‘‘**.**’’.

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’’. None of these examples match names that start with ‘‘.’’.

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 unmodified argument to the ‘‘/’’ preceding the first ‘‘*?[’’. 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.

If a command has one argument with ‘‘*’’, ‘‘?’’, or ‘‘[’’, a diagnostic is printed if no file names match that argument. If a command has several such arguments, a diagnostic is only printed if they *all* fail to match any files.

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.

A sequence of characters enclosed in single quotes (´) is taken literally, with no substitution or special processing whatsoever.

Sequences of characters enclosed in double quotes (") are also taken literally, except that ‘‘\’’, ‘‘"’’, and ‘‘$’’ are handled specially. The sequences ‘‘\"’’ and ‘‘\$’’ yield ‘‘"’’ and ‘‘$’’, respectively. The sequence ‘‘\x’’, where ‘‘x’’ is any character except ‘‘"’’ or ‘‘$’’, yields ‘‘\x’’. A ‘‘$’’ within a quoted string is processed in the same manner as a ‘‘$’’ that is not in a quoted string (see below), unless it is preceded by a ‘‘\’’. 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"$’’. Quotes permit the inclusion of blanks in the heading, which is a single argument to *pr.* Note that ‘‘\’’ inside quotes disappears only when preceding ‘‘$’’ or ‘‘"’’.

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 [ −v ] [ name [ arg1 ... ] ]
```

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 (and also 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*). ‘‘$0’’ is replaced by *name.* Shell variables (‘‘$a’’ − ‘‘$z’’), described below, are replaced in the same way.

The special argument ‘‘$*’’ is a name for the *current* sequence of all arguments from ‘‘$1’’ through the last argument, each argument separated from the previous by a single blank.

The special argument ‘‘$$’’ is the ASCII representation of the unique process number of the current Shell. This string is useful for creating temporary file names within command files.

The sequence ‘‘$x’’, where ‘‘x’’ is any character except one of the 38 characters mentioned above, is taken to refer to a variable ‘‘x’’ whose value is the null string. All substitution on a command line occurs *before* the line is interpreted: no action that alters the value of any variable can have any effect on a reference to that variable that occurs on the *same* line.

The argument **−t,** used alone, causes *sh* to read the standard input for a single line, execute it as a command, and then exit. 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.

The argument **−v** (''verbose'') causes every command line to be printed after all substitution occurs, but before execution. Each argument is preceded by a single blank. When given, the **−v** must be the first argument.

Used alone, the argument ''−'' suppresses prompting, and is commonly used when piping commands into the Shell:

```
ls | sed "s/.*/echo &:;cat &/" | sh −
```

prints all files in a directory, each prefaced by its name.

Initialization. When the Shell is invoked under the name ''−'' (as it is when you login), it attempts to read the file ''.profile'' in the current directory and execute the commands found there. When it finishes with ''.profile'', the Shell prompts the user for input as usual. Typical files contain commands to set terminal tabs and modes, initialize values of Shell variables, look at mail, etc.

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.

Command file errors; interrupts. An y Shell-detected error, or an interrupt signal, during the execution of a command file causes the Shell to cease execution of that file. (Except after *onintr;* see below.)

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 come from the zero length file ''/dev/null''.

Special commands. The follo wing commands are treated specially by the Shell. These commands generally do not work when named as arguments to programs like *time, if,* or *nohup* because in these cases they are not invoked directly by the Shell.

chdir and *cd* are 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 [*integer*] is done by manipulating the arguments to the Shell. In the normal case, *shift* has the effect of decrementing the Shell argument names by one (''$1'' disappears, ''$2'' becomes ''$1'', etc.). When the optional *integer* is given, only arguments equal to or greater than that number are shifted.

''**:**'' is simply ignored.

''='' *name* [*arg1* [*arg2*]]
The single character Shell variable *(name)* is assigned a value, either from the optional argument(s), or from standard input. If a single argument is given, its value is used. If a second argument is included, its value is used only if the first argument has a null value. This permits a simple way of setting up default values for arguments:

```
= a "$1" default
```

causing default to be used if ''$1'' is null or omitted entirely.

Such variables are referred to later with a ''$'' prefix. The variables ''$a'' through ''$m'' are guaranteed to be initialized to null, and will never have special meanings. The variables ''$n'' through ''$z'' are *not* guaranteed to be initialized to null, and may, at some time in the future, acquire special meanings. Currently, these variables have predefined meanings:

$n is the argument count to the Shell command.

$p contains the Shell directory search sequence for command execution. Alternatives are separated by ‘‘:’’. The default initial value is:

= p ":/bin:/usr/bin"

which executes from the current directory (the null pathname), then from ‘‘/bin’’, then from ‘‘/usr/bin’’, as described above. For the super-user, the value is:

= p "/bin:/etc:/"

Using the same syntax, users may choose their own sequence by storing it in a file named ‘‘.path’’ in their login directory. The ‘‘.path’’ information is available to successive Shells; the ‘‘$p’’ value is not. If the ‘‘.path’’ file contains a second line, it is interpreted as the name of the Shell to be invoked to interpret Shell procedures. (See ‘‘$z’’ below).

$r is the exit status code of the preceding command. ‘‘0’’ is the normal return from most commands.

$s is your login directory.

$t is your login tty letter.

$w is your file system name (first component of ‘‘$s’’).

$z is the name of the program to be invoked when a Shell procedure is to be executed. Its default value is ‘‘/bin/sh’’, but it can be overridden by supplying a second line in the ‘‘.path’’ file. It can be used to achieve consistent use of a specific Shell during periods when several distinct Shells are present in the system. For safety in the presence of change, use ‘‘$z’’ as a command rather than ‘‘sh’’.

No substitution of variables (or arguments) occurs within single quotes (´). Within double quotes ("), a variable string is substituted unchanged, even if it contains characters (‘‘"’’, ‘‘\’’, or ‘‘$’’) that might otherwise be treated specially. In particular, the argument ‘‘$1’’ can be passed unchanged to another command by using ‘‘"$1"’’. Outside quotes, substituted characters possess the same special meanings they have as if typed directly.

To illustrate, suppose that the shell procedure ‘‘mine’’ is called with two arguments:

```
sh  mine  ´a; echo "$2"´  ´"´
```

Then sample commands in ‘‘mine’’ and their output are as follows:

echo ´$1´	$1
echo "$1"	a; echo "$2"
echo $1	a
	$2
echo $2a"	a
echo "$2a"	"a
echo $2	syntax error

The appearance of the string ‘‘$2’’ (rather than ‘‘"’’) occurs because the Shell performs only one level of substitution, i.e., no rescanning is done.

onintr [*label*]
Causes control to pass to the label named (using a *goto* command) if the Shell command file is interrupted. After such a transfer, interrupts are re-enabled. *Onintr* without an ar gument also enables interrupts. The special label ‘‘−’’ will cause any number of interrupts to be ignored.

next [*name*]
This command causes *name* to become the standard input. Current input is never effectively resumed. If the argument is omitted, your terminal keyboard is assumed.

pump [−[*subchar*]] [+] [*eofstr*]
This command reads its standard input until it finds *eofstr* (defaults to ‘‘!’’ if not specified) alone on a line. It normally substitutes the values of arguments and variables (marked with ‘‘$’’ as usual). If ‘‘−’’ is given alone,

substitution is suppressed, and "−*subchar*" causes *subchar* to be used in place of "$" as the indicator character for substitution. Escaping is handled as in quoted strings: the indicator character may be escaped by preceding it by "\". Otherwise, "\" and other characters are transmitted unchanged. If "+" is used, leading tabs in the input are thrown away, allowing indentation. This command may be used interactively and in pipelines.

opt [**−v**] [**+v**] [**−p** *prompt-str*]
The argument **−v** turns on tracing, in the same style as a **−v** argument for the Shell. The argument **+v** turns it off. The argument **−p** causes the next argument string to be used as the prompt string for an interactive shell.

Commands implementing control structure. Control structure is provided by a set of commands that happen currently to be built into the Shell, although no guarantee is given that this will remain so. They are documented separately as follows:

if(I) − if, else, endif, and test.
switch(I) − switch, breaksw, endsw.
while(I) − while, end, break, continue.
goto(I) − goto.
exit(I) − exit.

FILES

/etc/sha, for shell accounting.
/dev/null as a source of end-of-file.
.path in login directory to initialize $p and name of Shell.
.profile in login directory for general initialization.

SEE ALSO

The UNIX Time-Sharing System by D. M. Ritchie and K. Thompson, CACM, July, 1974, which gives the theory of operation of the Shell.
PWB/UNIX Shell Tutorial by J. R. Mashey.
chdir(I), equals(I), exit(I), expr(I), fd2(I), if(I), login(I), loginfo(I), onintr(I), pump(I), shift(I), switch(I), wait(I), while(I), pexec(III), sha(V), glob(VIII)

EXIT CODE

If an error occurs in a command file, the Shell returns the exit value "1" to the parent process. Otherwise, the current value of the Shell variable *$r* is returned. Execution of a command file is terminated by an error.

BUGS

There is no built-in way to redirect the diagnostic output; *fd2(I)* must be used.
A single command line is limited to 1000 total characters, 50 arguments, and approximately 20 operators.