Aliases

An alias is a name and corresponding value set using the alias(1) builtin command. Whenever a reserved word may occur (see above), and after checking for reserved words, the shell checks the word to see if it matches an alias. If it does, it replaces it in the input stream with its value. For example, if there is an alias called "lf" with the value "ls -F", then the input

```
lf foobar <return>
```

would become

```
ls -F foobar <return>
```

Aliases provide a convenient way for naive users to create shorthands for commands without having to learn how to create functions with arguments. They can also be used to create lexically obscure code. This use is discouraged.

Argument List Processing

All of the single letter options have a corresponding name that can be used as an argument to the '-o' option. The set -o name is provided next to the single letter option in the description below. Specifying a dash "-" turns the option on, while using a plus "+" disables the option. The following options can be set from the command line or with the set(1) builtin.

-a allexport
Export all variables assigned to. (UNIMPLEMENTED for 4.4alpha)

-C noclobber
Don't overwrite existing files with ">". (UNIMPLEMENTED for 4.4alpha)

-e errexit
If not interactive, exit immediately if any untested command fails. The exit status of a command is considered to be explicitly tested if the command is used to control an if, elif, while, or until; or if the command is the left hand operand of an "&&" or "||" operator.

-f noglob
Disable pathname expansion.

-n noexec
If not interactive, read commands but do not execute them. This is useful for checking the syntax of shell scripts.

-u nounset
Write a message to standard error when attempting to expand a variable that is not set, and if the shell is not interactive, exit immediately. (UNIMPLEMENTED for 4.4alpha)

-v verbose
The shell writes its input to standard error as it is read. Useful for debugging.

-x xtrace
Write each command to standard error (preceded by a "+") before it is executed. Useful for debugging.

-I ignoreeof
Ignore EOF's from input when interactive.

-i interactive
Force the shell to behave interactively.

-m monitor

Turn on job control (set automatically when interactive).

-s stdin
Read commands from standard input (set automatically if no file arguments are present). This option has no effect when set after the shell has already started running (i.e. with set(1)).

-V vi
Enable the builtin vi(1) command line editor (disables -E if it has been set).

-E emacs
Enable the builtin emacs(1) command line editor (disables -V if it has been set).

-b notify
Enable asynchronous notification of background job completion. (UNIMPLEMENTED for 4.4alpha)

The shell reads input in terms of lines from a file and breaks it up into words at whitespace (blanks and tabs), and at certain sequences of characters that are special to the shell called "operators". There are two types of operators: control operators and redirection operators (their meaning is discussed later). Following is a list of operators:

Control operators:

```
& && ( ) ; ;; | || <newline>
```

Redirection operator:

```
< > >| << >> <& >& <<- = <>
```

Arithmetic Expansion

Arithmetic expansion provides a mechanism for evaluating an arithmetic expression and substituting its value. The format for arithmetic expansion is as follows:

```
$((expression))
```

The expression is treated as if it were in double-quotes, except that a double-quote inside the expression is not treated specially. The shell expands all tokens in the expression for parameter expansion, command substitution, and quote removal.

Next, the shell treats this as an arithmetic expression and substitutes the value of the expression.

Background Commands -- &

This is unimplemented in the NT shell

If a command is terminated by the control operator ampersand (&), the shell executes the command asynchronously -- that is, the shell does not wait for the command to finish before executing the next command.

The format for running a command in background is:

command1 & [command2 & ...] If the shell is not interactive, the standard input of an asynchronous command is set to /dev/null.

Backslash

A backslash preserves the literal meaning of the following character, with the exception of <newline>. A backslash preceding a <newline> is treated as a line continuation.

Builtins

This section lists the builtin commands which are builtin because they need to perform some operation that can't be performed by a separate process. In addition to these, there are several other commands that may be builtin for efficiency (e.g. printf(1), echo(1), test(1), etc).

alias [name[=string] ...]
: If name=string is specified, the shell defines the alias "name" with value "string". If just "name" is specified, the value of the alias "name" is printed. With no arguments, the alias builtin prints the names and values of all defined aliases (see unalias).

bg [job] ...
: Continue the specified jobs (or the current job if no jobs are given) in the background.

  This feature is not implemented in the NT version.

command command arg...
: Execute the specified builtin command. (This is useful when you have a shell function with the same name as a builtin command.)

cd [directory]
: Switch to the specified directory (default $HOME). If an entry for CDPATH appears in the environment of the cd command or the shell variable CDPATH is set and the directory name does not begin with a slash, then the directories listed in CDPATH will be searched for the specified directory. The format of CDPATH is the same as that of PATH. In an interactive shell, the cd command will print out the name of the directory that it actually switched to if this is different from the name that the user gave. These may be different either because the CDPATH mechanism was used or because a symbolic link was crossed.

. file
: The commands in the specified file are read and executed by the shell.

eval string...
: Concatenate all the arguments with spaces. Then reparse and execute the command.

exec [command arg...]
: Unless command is omitted, the shell process is replaced with the specified program (which must be a real program, not a shell builtin or function). Any redirections on the exec command are marked as permanent, so that they are not undone when the exec command finishes.

exit [exitstatus]

Terminate the shell process. If exitstatus is given it is used as the exit status of the shell; otherwise the exit status of the preceding command is used.

export name...

The specified names are exported so that they will appear in the environment of subsequent commands. The only way to un-export a variable is to unset it. The shell allows the value of a variable to be set at the same time it is exported by writing

```
export name=value
```

With no arguments the export command lists the names of all exported variables.

fc [-e editor] [first [last]]
fc -l [-nr] [first [last]]
fc -s [old=new] [first]

The fc builtin lists, or edits and re-executes, commands previously entered to an interactive shell.

This feature is not implemented in the NT version.

-e editor

Use the editor named by editor to edit the commands. The editor string is a command name, subject to search via the PATH variable. The value in the FCEDIT variable is used as a default when -e is not specified. If FCEDIT is null or unset, the value of the EDITOR variable is used. If EDITOR is null or unset, ed(1) is used as the editor.

-l (ell)

List the commands rather than invoking an editor on them. The commands are written in the sequence indicated by the first and last operands, as affected by -r, with each command preceded by the command number.

-n

Suppress command numbers when listing with -l.

-r

Reverse the order of the commands listed (with -l) or edited (with neither -l nor -s).

-s

Re-execute the command without invoking an editor.

first
last

Select the commands to list or edit. The number of previous commands that can be accessed are determined by the value of the HISTSIZE variable. The value of first or last or both are one of the following:

[+]number

A positive number representing a command number; command numbers can be displayed with the -l option.

-number

A negative decimal number representing the command that was executed number of commands previously. For example, -1 is the immediately previous command.

string

A string indicating the most recently entered command that begins with that string. If the old=new operand is not also specified with -s, the string form of the first operand cannot contain an embedded equal sign.

fc: The following environment variables affect the execution of

FCEDIT

Name of the editor to use.

HISTSIZE

The number of previous commands that are accessable.

fg [job]

Move the specified job or the current job to the foreground.

This feature is not implemented in the NT version.

getopts optstring var

The POSIX getopts command.

hash -rv command...

The shell maintains a hash table which remembers the locations of commands. With no arguments whatsoever, the hash command prints out the contents of this table. Entries which have not been looked at since the last cd command are marked with an asterisk; it is possible for these entries to be invalid.

With arguments, the hash command removes the specified commands from the hash table (unless they are functions) and then locates them. With the -v option, hash prints the locations of the commands as it finds them. The -r option causes the hash command to delete all the entries in the hash table except for functions.

jobid [job]

Print the process id's of the processes in the job. If the job argument is omitted, use the current job.

This feature is not implemented in the NT version.

jobs

This command lists out all the background processes which are children of the current shell process.

This feature is not implemented in the NT version.

pwd

Print the current directory. The builtin command may differ from the program of the same name because the builtin command remembers what the current directory is rather than recomputing it each time. This makes it faster. However, if the current directory is renamed, the builtin version of pwd will continue to print the old name for the directory.

read [-p prompt] [-e] variable...

The prompt is printed if the -p option is specified and the standard input is a terminal. Then a line is read from the standard input. The trailing newline is deleted from the line and the line is split as described in the section on word splitting above, and the pieces are assigned to the variables in order. If there are more pieces than variables, the remaining pieces (along with the characters in IFS that separated them) are assigned to the last variable. If there are more variables than pieces, the remaining variables are assigned the null string. The -e option causes any backslashes in the input to be treated specially. If a backslash is followed by a newline, the backslash and the newline will be deleted. If a backslash is followed by any other character, the backslash will be deleted and the following character will be treated as though it were not in IFS, even if it is.

readonly name...

The specified names are marked as read only, so that they cannot be subsequently modified or unset. The shell allows the value of a variable to be set at the same time it is marked read only by writing

readonly name=value

With no arguments the readonly command lists the names of all read only

variables.

set [{ -options | +options | -- }] arg...

The set command performs three different functions. With no arguments, it lists the values of all shell variables. If options are given, it sets the specified option flags, or clears them as described in the section called "Argument List Processing". The third use of the set command is to set the values of the shell's positional parameters to the specified args. To change the positional parameters without changing any options, use "--" as the first argument to set. If no args are present, the set command will clear all the positional parameters (equivalent to executing "shift $#".

setvar variable value

Assigns value to variable. (In general it is better to write variable=value rather than using setvar. Setvar is intended to be used in functions that assign values to variables whose names are passed as parameters.) shift [n]

Shift the positional parameters n times. A shift sets the value of $1 to the value of $2, the value of $2 to the value of $3, and so on, decreasing the value of $# by one. If there are zero positional parameters, shifting doesn't do anything.

test [expr]

Exits with a status of 0 (trueness) or 1 (falseness) depending on the evaluation of EXPR. Expressions may be unary or binary. Unary expressions are often used to examine the status of a file. There are string operators as well, and numeric comparison operators.

File operators:

-b FILE

True if file is block special.

-c FILE

True if file is character special.

-d FILE

True if file is a directory.

-e FILE

True if file exists.

-f FILE

True if file exists and is a regular file.

-g FILE

True if file is set-group-id.

-h FILE
True if file is a symbolic link. Use "-L".

-L FILE
True if file is a symbolic link.

-k FILE
True if file has its "sticky" bit set.

-p FILE
True if file is a named pipe.

-r FILE
True if file is readable by you.

-s FILE
True if file is not empty.

-S FILE
True if file is a socket.

-t FD
True if FD is opened on a terminal.

-u FILE
True if the file is set-user-id.

-w FILE
True if the file is writable by you.

-x FILE
True if the file is executable by you.

-O FILE
True if the file is effectively owned by you.

-G FILE
True if the file is effectively owned by your group.

FILE1 -nt FILE2
True if file1 is newer than (according to modification date) file2.

FILE1 -ot FILE2
True if file1 is older than file2.

FILE1 -ef FILE2
True if file1 is a hard link to file2.

String operators:

-z STRING
True if string is empty.

-n STRING or STRING
True if string is not empty. STRING1 = STRING2 True if the strings are equal. STRING1 != STRING2 True if the strings are not equal.

Other operators:

! EXPR
True if expr is false.

EXPR1 -a EXPR2
True if both expr1 AND expr2 are true.

EXPR1 -o EXPR2
True if either expr1 OR expr2 is true.

arg1 OP arg2
Arithmetic tests. OP is one of -eq, -ne, -lt, -le, -gt, or ge.

Arithmetic binary operators return true if ARG1 is equal, not-equal, less-than, less-than-or-equal, greater-than, or greater-than-or-equal than ARG2.

trap [action] signal...
Cause the shell to parse and execute action when any of the specified signals are received. The signals are specified by signal number. Action may be null or omitted; the former causes the specified signal to be ignored and the latter causes the default action to be taken. When the shell forks off a subshell, it resets trapped (but not ignored) signals to the default action. The trap command has no effect on signals that were ignored on entry to the shell.

umask [mask]
Set the value of umask (see umask(2)) to the specified octal value. If the argument is omitted, the umask value is printed.

unalias [-a] [name]
If "name" is specified, the shell removes that alias. If "-a" is specified, all aliases are removed.

unset name...

The specified variables and functions are unset and unexported. If a given name corresponds to both a variable and a function, both the variable and the function are unset.

wait [job]

Wait for the specified job to complete and return the exit status of the last process in the job. If the argument is omitted, wait for all jobs to complete and then return an exit status of zero.

Command Exit Status

Each command has an exit status that can influence the behavior of other shell commands. The paradigm is that a command exits with zero for normal or success, and non-zero for failure, error, or a false indication. The man page for each command should indicate the various exit codes and what they mean. Additionally, the builtin commands return exit codes, as does an executed function.

Command Line Editing

When sh is being used interactively from a terminal, the current command and the command history (see fc in Builtins) can be edited using vi-mode command-line editing. This mode uses commands, described below, similar to a subset of those described in the vi man page. The command set -o vi enables vi-mode editing and place sh into vi insert mode. With vi-mode enabled, sh can be switched between insert mode and command mode. The editor is not described in full here, but will be in a later document. It's similar to vi: typing <ESC> will throw you into command VI command mode. Hitting <return> while in command mode will pass the line to the shell.

Command Substitution

Command substitution allows the output of a command to be substituted in place of the command name itself. Command substitution occurs when the command is enclosed as follows:

```
$(command)
```

or ("backquoted" version):

```
`command`
```

The shell expands the command substitution by executing command in a subshell environment and replacing the command substitution with the standard output of the command, removing sequences of one or more <newline>s at the end of the substitution. (Embedded <newline>s before the end of the output are not removed; however, during field splitting, they may be translated into <space>s, depending on the value of IFS and quoting that is in effect.)

Commands

The shell interprets the words it reads according to a language, the specification of which is outside the scope of this man page (refer to the BNF in the POSIX 1003.2 document). Essentially though, a line is read and if the first word of the line (or after a control operator) is not a reserved word, then the shell has recognized a simple command. Otherwise, a complex command or some other special construct may have been recognized.

Complex Commands

Complex commands are combinations of simple commands with control operators or reserved words, together creating a larger complex command. More generally, a command is one of the following:

simple command
pipeline
list or compound-list
compound command
function definition

Unless otherwise stated, the exit status of a command is that of the last simple command executed by the command.

Contents

Copyright Notices

This shell is based on code mostly from the BSD 4.4 Lite distribution. Files from the original distribution contain the following copyright notice:

The implementation of the builtin command "test" is derived from source code from the "GNU Bourne Again Shell" bash. That file includes the following notice:

Bash is distributed in the hope that it will be useful, but WITHOUT ANY
WARRANTY; without even the implied warranty of MERCHANTABILITY or
FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License
for more details.

You should have received a copy of the GNU General Public License along
with Bash; see the file COPYING. If not, write to the Free Software
Foundation, 675 Mass Ave, Cambridge, MA 02139, USA. */

Double Quotes

Enclosing characters within double quotes preserves the literal meaning of all characters except dollarsign ($), backquote (`), and backslash (\). The backslash inside double quotes is historically weird, and serves to quote only the following characters: $ ` " \ <newline>. Otherwise it remains literal.

Functions

The syntax of a function definition is

```
name ( ) command
```

A function definition is an executable statement; when executed it installs a function named name and returns an exit status of zero. The command is normally a list enclosed between "{" and "}".

Variables may be declared to be local to a function by using a local command. This should appear as the first statement of a function, and the syntax is

```
local [ variable | - ] ...
```

Local is implemented as a builtin command.

When a variable is made local, it inherits the initial value and exported and readonly flags from the variable with the same name in the surrounding scope, if there is one. Otherwise, the variable is initially unset. The shell uses dynamic scoping, so that if you make the variable x local to function f, which then calls function g, references to the variable x made inside g will refer to the variable x declared inside f, not to the global variable named x.

The only special parameter than can be made local is "-". Making "-" local any shell options that are changed via the set command inside the function to be restored to their original values when the function returns.

The syntax of the return command is

```
return [ exitstatus ]
```

It terminates the currently executing function. Return is implemented as a builtin command.

Introduction

Sh is the standard command interpreter for the system. The current version of sh is in the process of being changed to conform with the POSIX 1003.2 and 1003.2a specifications for the shell. This version has many features which make it appear similar in some respects to the Korn shell, but it is not a Korn shell clone (run GNU's bash if you want that). Only features designated by POSIX, plus a few Berkeley extensions, are being incorporated into this shell. We expect POSIX conformance by the time 4.4 BSD is released. This man page is not intended to be a tutorial or a complete specification of the shell.

Invocation

If no args are present and if the standard input of the shell is connected to a terminal (or if the -i flag is set), the shell is considered an interactive shell. An interactive shell generally prompts before each command and handles programming and command errors differently (as described below). When first starting, the shell inspects argument 0, and if it begins with a dash '-', the shell is also considered a login shell. This is normally done automatically by the system when the user first logs in. A login shell first reads commands from the files /etc/profile.sh and profile.sh if they exist. If the environment variable SHENV is set on entry to a shell, or is set in the profile.sh of a login shell, the shell next reads commands from the file named in SHENV. Therefore, a user should place commands that are to be executed only at login time in the .profile file, and commands that are executed for every shell inside the ENV file. To set the SHENV variable to some file, place the following line in your profile.sh of your home directory

```
SHENV=$HOME/.shinit; export SHENV
```

substituting for ".shinit" any filename you wish. Since the ENV file is read for every invocation of the shell, including shell scripts and non-interactive shells, the following paradigm is useful for restricting commands in the ENV file to interactive invocations. Place commands within the "case" and "esac" below (these commands are described later):

```
case $- in *i*)
     # commands for interactive use only
     ...
esac
```

If command line arguments besides the options have been specified, then the shell treats the first argument as the name of a file from which to read commands (a shell script), and the remaining arguments are set as the positional parameters of the shell ($1, $2, etc). Otherwise, the shell reads commands from its standard input.

Lists -- Generally Speaking

A list is a sequence of zero or more commands separated by newlines, semicolons, or ampersands, and optionally terminated by one of these three characters. The commands in a list are executed in the order they are written. If command is followed by an ampersand, the shell starts the command and immediately proceed onto the next command; otherwise it waits for the command to terminate before proceeding to the next one.

"&&" and "||" are AND-OR list operators. "&&" executes the first command, and then executes the second command iff the exit status of the first command is zero. "||" is similar, but executes the second command iff the exit status of the first command is nonzero. "&&" and "||" both have the same priority.

The syntax of the if command is

```
if list
then list
[ elif list
then    list ] ...
[ else list ]
fi
```

The syntax of the while command is

```
while list
do   list
done
```

The two lists are executed repeatedly while the exit status of the first list is zero. The until command is similar, but has the word until in place of while repeat until the exit status of the first list is zero.

The syntax of the for command is

```
  for variable in word...
do   list
done
```

The words are expanded, and then the list is executed repeatedly with the variable set to each word in turn. do and done may be replaced with "{" and "}".

The syntax of the break and continue command is

```
break [ num ]
continue [ num ]
```

Break terminates the num innermost for or while loops. Continue continues with the next iteration of the innermost loop. These are implemented as builtin commands.

The syntax of the case command is

```
case word in
pattern) list ;;
...
esac
```

The pattern can actually be one or more patterns (see Shell Patterns), separated by "|" characters.

Commands may be grouped by writing either

```
(list)
```

or

```
{ list; }
```

The first of these executes the commands in a subshell.

Overview

The shell is a command that reads lines from either a file or the terminal, interprets them, and generally executes other commands. It is the program that is running when a user logs into the system (although a user can select a different shell with the chsh(1) command). The shell implements a language that has flow control constructs, a macro facility that provides a variety of features in addition to data storage, along with built in history and line editing capabilities. It incorporates many features to aid interactive use and has the advantage that the interpretative language is common to both interactive and non-interactive use (shell scripts). That is, commands can be typed directly to the running shell or can be put into a file and the file can be executed directly by the shell.

Parameter Expansion

The format for parameter expansion is as follows:

```
${expression}
```

where expression consists of all characters until the matching }. Any } escaped by a backslash or within a quoted string, and characters in embedded arithmetic expansions, command substitutions, and variable expansions, are not examined in determining the matching }.

The simplest form for parameter expansion is:

```
${parameter}
```

The value, if any, of parameter is substituted.

The parameter name or symbol can be enclosed in braces, which are optional except for positional parameters with more than one digit or when parameter is followed by a character that could be interpreted as part of the name. If a parameter expansion occurs inside double-quotes:

· Pathname expansion is not performed on the results of the expansion.

Field splitting is not performed on the results of the expansion, with the exception of @. In addition, a parameter expansion can be modified by using one of the following formats.

${parameter:-word}
Use Default Values. If parameter is unset or null, the expansion of word is substituted; otherwise, the value of parameter is substituted.

${parameter:=word}
Assign Default Values. If parameter is unset or null, the expansion of word is assigned to parameter. In all cases, the final value of parameter is substituted. Only variables, not positional parameters or special parameters, can be assigned in this way.

${parameter:?[word]}
Indicate Error if Null or Unset. If parameter is unset or null, the expansion of word (or a message indicating it is unset if word is omitted) is written to standard error and the shell exits with a nonzero exit status. Otherwise, the value of parameter is substituted. An interactive shell need not exit.

${parameter:+word}
Use Alternate Value. If parameter is unset or null, null is substituted; otherwise, the expansion of word is substituted.

In the parameter expansions shown previously, use of the colon in the format results in a test for a parameter that is unset or null; omission of the colon results in a test for a parameter that is only unset.

${#parameter}
String Length. The length in characters of the value of parameter.

The following four varieties of parameter expansion provide for substring processing. In each case, pattern matching notation (see Shell Patterns), rather than regular expression notation, is used to evaluate the patterns. If parameter is * or @, the result of the expansion is unspecified. Enclosing the full parameter expansion string in double quotes does not cause the following four varieties of pattern characters to be quoted, whereas quoting characters within the braces has this effect. (UNIMPLEMENTED IN 4.4alpha)

${parameter%word}
Remove Smallest Suffix Pattern. The word is expanded to produce a pattern. The parameter expansion then results in parameter, with the smallest portion of the suffix matched by the pattern deleted.

${parameter%%word}
Remove Largest Suffix Pattern. The word is expanded to produce a pattern. The parameter expansion then results in parameter, with the largest portion of the suffix matched by the pattern deleted.

${parameter#word}
Remove Smallest Prefix Pattern. The word is expanded to produce a pattern. The parameter expansion then results in parameter, with the smallest portion of the prefix matched by the pattern deleted.

${parameter##word}
Remove Largest Prefix Pattern. The word is expanded to produce a pattern. The parameter expansion then results in parameter, with the largest portion of the prefix matched by the pattern deleted.

Path Search

When locating a command, the shell first looks to see if it has a shell function by that name. Then it looks for a builtin command by that name. Finally, it searches each entry in PATH in turn for the command.

The value of the PATH variable should be a series of entries separated by colons. Each entry consists of a directory name. The current directory may be indicated by an empty directory name.

Command names containing a slash are simply executed without performing any of the above searches.

Pathname Expansion (File Name Generation)

Unless the -f flag is set, file name generation is performed after word splitting is complete. Each word is viewed as a series of patterns, separated by slashes. The process of expansion replaces the word with the names of all existing files whose names can be formed by replacing each pattern with a string that matches the specified pattern. There are two restrictions on this: first, a pattern cannot match a string containing a slash, and second, a pattern cannot match a string starting with a period unless the first character of the pattern is a period. The Shell Patterns section describes the patterns used for both Pathname Expansion and the case(1) command.

Pipeline

A pipeline is a sequence of one or more commands separated by the control operator |. The standard output of all but the last command is connected to the standard input of the next command.

The format for a pipeline is:

```
[!] command1 [ | command2 ...]
```

The standard output of command1 is connected to the standard input of command2. The standard input, standard output, or both of a command is considered to be assigned by the pipeline before any redirection specified by redirection operators that are part of the command.

If the pipeline is not in the background (discussed later), the shell waits for all commands to complete.

If the reserved word ! does not precede the pipeline, the exit status is the exit status of the last command specified in the pipeline. Otherwise, the exit status is the logical NOT of the exit status of the last command. That is, if the last command returns zero, the exit status is 1; if the last command returns greater than zero, the exit status is zero.

Because pipeline assignment of standard input or standard output or both takes place before redirection, it can be modified by redirection. For example:

```
$ command1 2>&1 | command2
```

sends both the standard output and standard error of command1 to the standard input of command2.

A ; or <newline> terminator causes the preceding AND-OR-list (described next) to be executed sequentially; a & causes asynchronous execution of the preceding AND-OR-list.

Positional Parameters

A positional parameter is a parameter denoted by a number (n > 0). The shell sets these initially to the values of its command line arguments that follow the name of the shell script. The set(1) builtin can also be used to set or reset them.

Quoting

Quoting is used to remove the special meaning of certain characters or words to the shell, such as operators, whitespace, or keywords. There are three types of quoting: matched single quotes, matched double quotes, and backslash.

Redirections

Redirections are used to change where a command reads its input or sends its output. In general, redirections open, close, or duplicate an existing reference to a file. The overall format used for redirection is:

```
[n] redir-op file
```

where redir-op is one of the redirection operators mentioned previously. Following is a list of the possible redirections. The [n] is an optional number, as in '3' (not '[3]'), that refers to a file descriptor.

[n]> file
Redirect standard output (or n) to file.

[n]>| file
Same, but override the -C option.

[n]>> file
Append standard output (or n) to file.

[n]< file
Redirect standard input (or n) from file.

[n1]<&n2
Duplicate standard input (or n1) from file descriptor n2.

[n]<&-
Close standard input (or n).

[n1]>&n2
Duplicate standard output (or n) from n2.

[n]>&-
Close standard output (or n).

[n]<> file
Open file for reading and writing on standard input (or n).

The following redirection is often called a "here document".

```
[n]<< delimiter
    here-doc-text...
delimiter
```

All the text on successive lines up to the delimiter is saved away and made available to the command on standard input, or file descriptor n if it is specified. If the delimiter

as specified on the initial line is quoted, then the here-doc-text is treated literally, otherwise the text is subjected to parameter expansion, command substitution, and arithmetic expansion. If the operator is "<<-" instead of "<<", then leading tabs in the here-doc-text are stripped.

Reserved Words

Reserved words are words that have special meaning to the shell and are recognized at the beginning of a line and after a control operator. The following are reserved words:

```
!       elif    fi      while   case
else    for     then    {       }
do      done    until   if      esac
```

Their meaning is discussed elsewhere.

Search and Execution

There are three types of commands: shell functions, builtin commands, and normal programs--and the command is searched for (by name) in that order. They each are executed in a different way.

When a shell function is executed, all of the shell positional parameters (except $0, which remains unchanged) are set to the arguments of the shell function. The variables which are explicitly placed in the environment of the command (by placing assignments to them before the function name) are made local to the function and are set to the values given. Then the command given in the function definition is executed. The positional parameters are restored to their original values when the command completes.

Shell built-ins are executed internally to the shell, without spawning a new process.

Otherwise, if the command name doesn't match a function or builtin, the command is searched for as a normal program in the filesystem (as described in the next section). When a normal program is executed, the shell runs the program, passing the arguments and the environment to the program. If the program is a shell procedure, the shell will interpret the program in a subshell. The shell will reinitialize itself in this case, so that the effect will be as if a new shell had been invoked to handle the shell procedure, except that the location of commands located in the parent shell will be remembered by the child.

Shell Patterns

A pattern consists of normal characters, which match themselves, and meta-characters. The meta-characters are "!", "*", "?", and "[". These characters lose there special meanings if they are quoted. When command or variable substitution is performed and the dollar sign or back quotes are not double quoted, the value of the variable or the output of the command is scanned for these characters and they are turned into meta-characters.

An asterisk ("*") matches any string of characters. A question mark matches any single character. A left bracket ("[") introduces a character class. The end of the character class is indicated by a "]"; if the "]" is missing then the "[" matches a "[" rather than introducing a character class. A character class matches any of the characters between the square brackets. A range of characters may be specified using a minus sign. The character class may be complemented by making an exclamation point the first character of the character class.

To include a "]" in a character class, make it the first character listed (after the "!", if any). To include a minus sign, make it the first or last character listed

Simple Commands

If a simple command has been recognized, the shell performs the following actions:

Leading words of the form "name=value" are stripped off and assigned to the environment of the simple command. Redirection operators and their arguments (as described below) are stripped off and saved for processing.

The remaining words are expanded, and the first remaining word is considered the command name and the command is located. The remaining words are considered the arguments of the command. If no command name resulted, then the "name=value" variable assignments recognized above affect the current shell.

Redirections are performed as described in the next section.

Single Quotes

Enclosing characters in single quotes preserves the literal meaning of all the characters.

Special Parameters

A special parameter is a parameter denoted by one of the following special characters. The value of the parameter is listed next to its character.

*****

Expands to the positional parameters, starting from one. When the expansion occurs within a double-quoted string it expands to a single field with the value of each parameter separated by the first character of the IFS variable, or by a <space> if IFS is unset.

@

Expands to the positional parameters, starting from one. When the expansion occurs within double-quotes, each positional parameter expands as a separate argument. If there are no positional parameters, the expansion of @ generates zero arguments, even when @ is double-quoted. What this basically means, for example, is if $1 is "abc" and $2 is "def ghi", then "$@" expands to the two arguments: "abc" "def ghi"

#

Expands to the number of positional parameters.

?

Expands to the exit status of the most recent pipeline.

- (Hyphen)

Expands to the current option flags (the single-letter option names concatenated into a string) as specified on invocation, by the set builtin command, or implicitly by the shell.

$

Expands to the process ID of the invoked shell. A subshell retains the same value of $ as its parent.

!

Expands to the process ID of the most recent background command executed from the current shell. For a pipeline, the process ID is that of the last command in the pipeline.

0 (Zero.)

Expands to the name of the shell or shell script.

Tilde Expansion (substituting a user's home

A word beginning with an unquoted tilde character (~) is subjected to tilde expansion. All the characters up to a slash (/) or the end of the word are treated as a username and are replaced with the user's home directory. If the username is missing (as in ~/foobar), the tilde is replaced with the value of the HOME variable (the current user's home directory).

Variables and Parameters

The shell maintains a set of parameters. A parameter denoted by a name is called a variable. When starting up, the shell turns all the environment variables into shell variables. New variables can be set using the form

```
name=value
```

Variables set by the user must have a name consisting solely of alphabetics, numerics, and underscores - the first of which must not be numeric. A parameter can also be denoted by a number or a special character as explained below.

White Space Splitting (Field Splitting)

After parameter expansion, command substitution, and arithmetic expansion the shell scans the results of expansions and substitutions that did not occur in double-quotes for field splitting and multiple fields can result.

The shell treats each character of the IFS as a delimiter and use the delimiters to split the results of parameter expansion and command substitution into fields.

Word Expansions

This clause describes the various expansions that are performed on words. Not all expansions are performed on every word, as explained later.

Tilde expansions, parameter expansions, command substitutions, arithmetic expansions, and quote removals that occur within a single word expand to a single field. It is only field splitting or pathname expansion that can create multiple fields from a single word. The single exception to this rule is the expansion of the special parameter @ within double-quotes, as was described above.

The order of word expansion is:

> Tilde Expansion, Parameter Expansion, Command Substitution, Arithmetic Expansion (these all occur at the same time).
>
> Field Splitting is performed on fields generated by the previous step unless the IFS variable is null.
>
> Pathname Expansion (unless set -f is in effect).
>
> Quote Removal. The $ character is used to introduce parameter expansion, command substitution, or arithmetic evaluation.