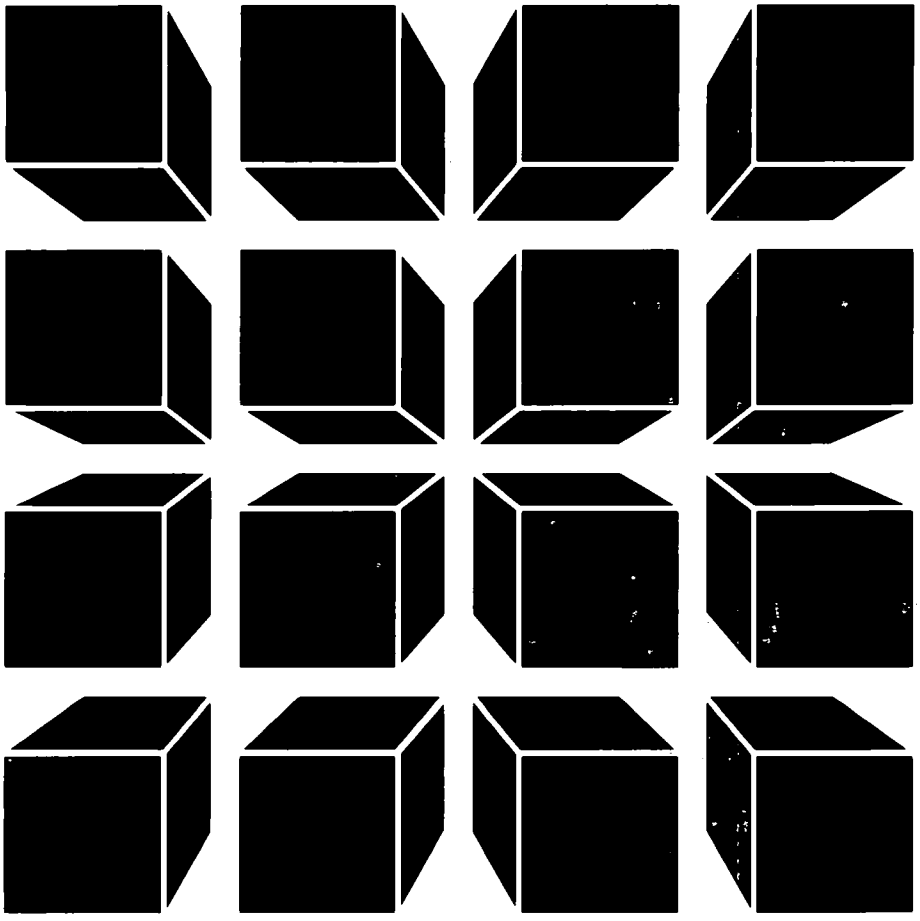
AT&T VIDEOTAPE LIBRARY

SHELL COMMAND LANGUAGE FOR PROGRAMMERS



WORKBOOK VOL II

AT&T

WORKBOOK VOLUME II

CONTENTS

VOLUME 9—Conditional Execution, Part 2

VOLUME 10—Looping Statement, Part 1

VOLUME 11—Looping Statement, Part 2

VOLUME 12—Command Line Interpretation

VOLUME 13—Built-in Statements

VOLUME 14—Redirection

VOLUME 15—Pipelines and Signal Processing

VOLUME 9

Conditional Execution, Part 2

WHAT YOU'LL BE ABLE TO DO AFTER VOLUME 9

- Use the **if-elif** control construct to conditionally execute command lines
- Use the **case** statement to conditionally execute command lines.
- Use the **tput** command with double quotes and command substitution
- Use the **sed** command to edit information from standard input or from a file.

if-elif-else STATEMENT

The shell language has the capability to test several conditions in one construct through the use of the **if-elif** statement.

FORMAT

```
if
     command
     testing_command
then
     command
     command
elif
     command
     testing_command
then
     command
     command
    .
    .
    .
else
     command
     command
fi
```

This construct is used when a number of mutually exclusive (nonoverlapping) tests need to be made. This construct is often used when processing arguments.

if-elif-else STATEMENT — EXAMPLE

```
$ pr -tn direct<CR>
 1  if
 2      test ${#} -lt "1"
 3  then
 4      echo "Enter name: \c"
 5      read name
 6      if
 7          test -z "${name}"
 8      then
 9          echo "No name entered"
10          exit 3
11      fi
12  elif
13      test -z "${1}"
14  then
15      echo "Null argument supplied"
16      exit 4
17  else
18      name="${1}"
19  fi
20  grep "${name}" ${HOME}/listings
```

if-elif-else STATEMENT — EXAMPLE

```
$ direct<CR>
Enter name: <CR>
No name entered

$ direct<CR>
Enter name: hutch<CR>
981-2678   Robert Hutchison     hutch

$ direct vaness<CR>
981-6861   Annette Vaness       vaness

$ direct ""<CR>
Null argument supplied

$
```

case STATEMENT

The **case** statement is used to match an expression's value against several other values or patterns and execute statements based on those matches.

FORMAT

```
case value
in
pattern)
     command
     command
     ;;

pattern.2)
     command
     command
     ;;

pattern.3 | pattern.4)
     command
     command
     ;;
esac
```

value is the thing that is being matched by the patterns. The word **value** is usually the output from a variable or command substitution expression. The words containing the word pattern are used to pattern match against the word represented by **value**. If a match is found, the commands following the pattern are executed until the ;; is reached. Flow of control then goes to the word **esac** and execution resumes there.

If a vertical bar (|) appears between two patterns, that pattern matching expression matches *either* of the two patterns.

case STATEMENT

- USED TO MATCH EXPRESSION AGAINST PATTERN AND EXECUTE STATEMENTS BASED ON MATCH

FORMAT

```
case string       # (expression)
in
pattern)
     command
     command
     ;;

pattern.2)
     command
     command
     ;;

pattern.3 | pattern.4)
     command
     command
     ;;
esac
```

- VERTICAL BAR (|) BETWEEN TWO PATTERNS MATCHES EITHER OF THE TWO PATTERNS

case STATEMENT — PATTERNS

The patterns used by the shell for pattern matching are the same as those used for file name generation. These patterns will:

- Match any number of any ASCII characters, including no characters.
- Match any single ASCII character.
- Match a single character from the set of characters included in the character class. If the first character of the character class is an exclamation point (!), the expression matches any character that is not part of the list.
- Permit ranges of characters and are separated with the special character dash (-). The dash must appear between two ASCII characters, the first of which must have a lower ASCII value than the second.

These pattern matching characters and expressions may appear anywhere in the pattern in any order. Any characters not part of this pattern matching group retain their literal value.

The patterns may also contain variable and command substitution.

case STATEMENT — PATTERNS

- METACHARACTERS

*	MATCHES ANY NUMBER OF ANY ASCII CHARACTERS, INCLUDING NO CHARACTERS
?	MATCHES ANY SINGLE ASCII CHARACTER
[...]	MATCHES A SINGLE CHARACTER FROM THE SET OF CHARACTERS OR RANGE OF CHARACTERS LISTED IN THE CHARACTER CLASS

- VARIABLE SUBSTITUTION
- COMMAND SUBSTITUTION

case STATEMENT — EXAMPLE

This example comes from a *.profile* file and is used to set options for various terminal types.

```
$ cat term.select<CR>
echo "Terminal: \c"
read TERM
case "${TERM}"
in
    "" | 2626 )
        TERM=2626
        PS1='${LOGNAME}: '
        tabs -8
        stty erase '^h' kill '@' echoe tabs
        ;;
    hp | 2645 )
        TERM=2645
        PS1='${LOGNAME}: '
        tabs -8
        stty erase '^h' kill '@' echoe tabs
        ;;
    2640 )
        TERM=2640b
        PS1='${LOGNAME}: '
        tabs -8
        stty erase '^h' kill '@' echoe tabs
        ;;
```

case STATEMENT — EXAMPLE

```
        vt100[kjms] | vt100 )
            if
                test ${TERM} != "vt100k"
            then
                TERM=vt100
            fi
            PS1="$ "
            PS2="> "
            tabs -8
            stty tabs erase '^h' kill '@' echoe
            ;;
        *)
            echo terminal type \"${TERM}\" not recognized
            try again.
            ;;
    esac
```

This part of the **.profile** procedure reads a value into the variable **TERM** and matches the value of **${TERM}** against several patterns and performs the command between the pattern and the following ;;.

tput

A good command to use with double quotes and command substitution is **tput**. The **tput** command queries the terminal information database maintained in the directory /usr/lib/terminfo. It can return control sequences which take advantage of certain terminal capabilities. While in the past, one could echo the appropriate sequences for a specific terminal to say put the terminal into inverse video, **tput** allows this to be portable across many different terminal types. In order to use **tput**, one must set and export the TERM variable. **tput** is only available with System V Release 2.0 or later.

The key to the **tput** command is the **terminfo** data base (described in great detail in Section 4 of the *UNIX System V Programmer Reference Manual*). **terminfo** is a data base of sorts, listing how particular functions (such as reverse video, underlining, etc.) are implemented on different types of terminals. The features are now listed in individual files (versus **termcap** which had everything in one file), plus the files are compiled for speed of execution. The data base is quite extensive and should handle most terminals on the market.

The **tput** command allows you to access individual features without having to worry what terminal you may be currently using.

> **Note:** You *must* set the environment variable **TERM** in order to get the results you expect.

The first example returns to standard input the number of columns that this terminal supports, according to the value of **${TERM}**. The second example stores the sequence of characters needed to clear the terminal screen in a variable called CLEAR.

tput

- ALLOWS DIRECT QUERY OF THE **terminfo** DATA BASE (LIKE THE OLD *termcap* FILE)
- ALLOWS TERMINAL INDEPENDENT OPERATIONS
- BE SURE THE **TERM** VARIABLE IS SET
- QUERY TERMINAL SETTINGS (e.g., NUMBER OF COLUMNS OR LINES)
- PRINT CERTAIN FUNCTIONS (e.g., INVERSE VIDEO)

FORMAT

```
tput capname

where capname is the capability name
```

tput

FORMAT

tput cols NUMBER OF COLUMNS

tput clear CLEARS THE SCREEN

EXAMPLES

```
echo "This terminal supports `tput cols` columns"

CLEAR=`tput clear`;export CLEAR
```

tput CAPABILITIES

Listed is a sample of the capabilities made available. Again, see the *UNIX System V Programmer Reference Manual*, under terminfo(4), for the complete list.

QUERY TERMINAL SETTINGS

These capabilities typically return a number; for instance. the number of lines per screenful on a terminal.

BASIC SCREEN FUNCTIONS

These functions are normally made available through function keys.

tput CAPABILITIES

QUERY TERMINAL SETTINGS

cols NUMBER OF COLUMNS

lines NUMBER OF LINES

BASIC SCREEN FUNCTIONS

bel ECHO A "BELL"

clear CLEAR THE SCREEN

el CLEAR TO END OF LINE

ed CLEAR TO END OF DISPLAY

ENHANCE SCREEN DISPLAY

These features deserve closer attention as they can be used to improve the "ergonomics" of programs. Exactly what the terminal will display is somewhat dependent upon the terminal itself.

Turning on more than one feature at a time may or may not turn off features. The feature **sgr0** (that's a zero) will try to revert the terminal back to its original (power on) state.

What actually is output is a string that can be saved in a variable using command substitution.

tput CAPABILITIES

ENHANCE SCREEN DISPLAY

smso	BEGIN STANDOUT MODE (VARIES FROM TERMINAL TO TERMINAL)
rmso	END STANDOUT MODE
smul	BEGIN UNDERLINE MODE
rmul	END UNDERLINE MODE
blink	BLINKING DISPLAY
bold	EMBOLDEN DISPLAY
dim	DIM DISPLAY
rev	REVERSE VIDEO DISPLAY
sgr0	END ENHANCE DISPLAY (MAY NOT REVERT BACK TO ORIGINAL STATE)

tput — EXAMPLES

EXAMPLE 1

Save the output string of the **bold** feature.

It is then output using the **echo** command.

EXAMPLE 2

Turn on the highlight feature before prompting for data.

Turn off after printing, then turn on underlining so the input is underlined as it is entered.

Turn the feature off.

tput — EXAMPLES

EXAMPLE 1

```
bold='tput bold'
reset='tput sgr0'
echo "${bold}Report Title${reset}"
```

EXAMPLE 2

```
tput smso
echo "Enter name you wish to retrieve: \c"
tput rmso

tput smul
read name
tput rmul
```

sed COMMAND

The **sed** editor provides a convenient way to edit information either from the standard input or from a file as a batch process. In either case, the edited result is sent to the standard output.

The **sed** stream editor uses the same commands as the **ed** editor, but applies the given command to every line in the file and prints the result to the standard output. Relative line numbers may not be used, but absolute line numbers may. When absolute line numbers are used, only the mentioned line or lines are edited.

Since input is expected from the standard input, a file name is not designated and it may be used as a filter (appearing in between pipe symbols).

The format of the **sed** command is as follows:

```
$ sed "editor command" [file file ...]<CR>
```

The **editor** command is enclosed in either single or double quotes, depending on whether there are parameter or command substitutions within the **editor** command.

Example:

```
$ who | sed "s/ .*//" | pr -5atw80
  cec0        cec1        cec32       cec14       rht
  tab         cec18       jba         root
```

sed COMMAND

FORMAT

```
$ sed "editor command" [file file ...]<CR>
```

- IF FILE IS NOT SPECIFIED, sed TAKES STANDARD INPUT (MAY BE USED AS A FILTER)
- **editor command** IS PERFORMED ON EVERY LINE OF THE FILE
- ABSOLUTE LINE NUMBERS MAY BE USED
- RELATIVE LINE NUMBERS MAY NOT BE USED
- WILL COPY EDITED FILE TO STANDARD OUTPUT — FILE CONTENTS ARE NOT CHANGED
- OUTPUT REDIRECTION (NOT TO SAME FILE)
- EXAMPLE

```
$ who | sed "s/ .*//" | pr -5atw80
cec0    cec1    cec32    cec14    rht
tab     cec18   jba      root

$
```

sed COMMAND — EXAMPLE

Programs that make modifications to files to which names are supplied as arguments should always first check to see if those arguments were properly supplied or give them default file names.

```
$ cat tree<CR>
find ${1:-.} -print | sort |
sed "s|[^/]*/|    |g"

$ tree<CR>

    backup
        outline
        unit.6
        unit.7
        unit.8
    laserit
    outline
    skeleton
    tree
    unit.4
    unit.5
    unit.8

$
```

In this example, the shell tries to interpret the |, ^, and the * characters, but putting them in double quotes protected them from the expansion. In this case, you can also use single quotes.

VOLUME 9 EXERCISE

1. Write a shell program named **rlw** to do the following:

   a. Display the name of the present working directory and list its contents in 5-column format on the terminal.

   b. If the present working directory contains any subdirectories, do steps 1 and 2 for each subdirectory. (**HINT:** A recursive call to the shell program is very handy here.)

   1. Modify the program to accept the starting directory name as a positional parameter. If the positional parameter is not set, the program should use the present working directory.

   2. Modify the program to accept more than one directory name as positional parameters and perform its function on each in turn.

   3. Modify the program to announce the name of the subdirectory and ask whether it should:

      a. Descend into the subdirectory and perform its function

      b. List the directory contents without changing directories

      c. Ignore the directory.

VOLUME 9 EXERCISE

2. Write a shell program named **mycal** that:

   a. if no arguments, it prints the calendar for the current month and year.

   b. if one argument, that is an abbreviation of a month, prints the calendar for that month in the current year. If the argument is a number, then it prints the calendar for the entire year.

   c. if two arguments, it prints the calendar for that month and year. The month should be entered as an abbreviation or a number.

   EXAMPLE

```
$ mycal<CR>
     September 1986
 S  M Tu  W Th  F  S
                1  2  3
 4  5  6  7  8  9 10
11 12 13 14 15 16 17
18 19 20 21 22 23 24
25 26 27 28 29 30

$ mycal oct<CR>
     October 1986
 S  M Tu  W Th  F  S
                   1
 2  3  4  5  6  7  8
 9 10 11 12 13 14 15
16 17 18 19 20 21 22
23 24 25 26 27 28 29
30 31
```

VOLUME 9 EXERCISE

```
$ mycal Feb 1818<CR>
     February 1818
  S  M Tu  W Th  F  S
  1  2  3  4  5  6  7
  8  9 10 11 12 13 14
 15 16 17 18 19 20 21
 22 23 24 25 26 27 28

$ mycal 4 1980<CR>
     April 1980
  S  M Tu  W Th  F  S
        1  2  3  4  5
  6  7  8  9 10 11 12
 13 14 15 16 17 18 19
 20 21 22 23 24 25 26
 27 28 29 30
```

3. Write a shell procedure named **findit** that will prompt for the starting directory (assume current directory for default), and the name of a file to be located within a subdirectory structure, and will invoke the **find** command with the proper format. Use command substitution with the **tput** command to highlight the prompting.

4. Write a shell procedure named **diskchk** that will print the file system names and number of blocks and inodes for all file systems that are running out of resources. A positional parameter is to be supplied, and if either the number of inodes OR the number of remaining blocks for a particular file system falls below that number, the file system name, number of blocks, and the number of inodes should be printed.

   a. Add a default value to the limit being checked for of 3000 for blocks and 1000 for inodes.

   HINT: The **df** command is used to find the number of remaining inodes and blocks for each file system. It is run without arguments.

   HINT: The device name might have embedded blanks that will have to be removed.

VOLUME 9 EXERCISE - ANSWERS

1. `$ cat rlw<CR>`

```
#program name:   rlw
#      author:   U. R. Wontu
#   arguments:   optional starting directories
#     purpose:   wide listing of directories recursively down
trap 'rm /tmp/${LOGNAME}* ;exit 1'  1 2 3 15
if     test ${#} = "0" #If no args, put . into $1
then   set .
fi
for i in ${*}    #for each positional parm.
do
       cd ${i}    #You must be in the dir. for prog to work
       echo;echo;pwd
       ls |
       while
             read name #one file in curr. dir at a time
       do
             if test -d ${name} #if a directory, keep a copy for
             then               #the recursive call
                    echo "[${name}]"
                    echo `pwd`/${name} >>/tmp/${LOGNAME}xx
             elif   test -x $name
             then   echo "-${name}-"
             else   echo ${name}
             fi
       done | pr -5tw80
       if test -f /tmp/${LOGNAME}xx #if file exists, is recursive
       then
           mv /tmp/${LOGNAME}xx /tmp/${LOGNAME}1xx #get rid in case
           for k in `cat   /tmp/${LOGNAME}1xx `  #used again
           do
           cat <<-EOF
           For directory            #Use default of "in ${*}"
               ${k}
           Do you want to:
              a) descend into and run ${0},
              b) list just ${k}'s contents
              c) ignore ${k}

           EOF
           echo "Please enter a, b, or c: \c"
           read ans
           case "${ans}"
           in
               a) eval ${0}  ${k};;
               b) ls ${k} | pr -5tw80;;
```

```
            c) ;;
            *) cat <<-EOF

            Sorry, you must enter a, b, or c.
            So I\'m skipping ${k}
            EOF
            ;;
         esac
   done
   fi
done
if
   test -f /tmp/${LOGNAME}1
then
   rm /tmp/${LOGNAME}1  #clean up after myself
fi

$
```

VOLUME 9 EXERCISE - ANSWERS

2. `$ cat mycal<CR>`

```
# mycal                        - modify the cal command
# number of arguments 0 - print current month and year
#                     1 - if ${1} is a month, print month
#                         and current year.
#                         if ${1} is a year, print the
#                         entire year.
#                     2 - print month of the given year
case ${#}
in
    0) set 'date'
       month=${2}
       year=${6};;
    1) case "${1}"
       in
           [0-9]*) year=${1};;
                *) month=${1}
                   set 'date'
                   year=${6};;
       esac;;
    *) month=${1}
       year=${2};;
esac
case "${month}"
in
    [Jj]an*) month=1;;
    [Ff]eb*) month=2;;
    [Mm]ar*) month=3;;
    [Aa]pr*) month=4;;
    [Mm]ay*) month=5;;
    [Jj]un*) month=6;;
    [Jj]ul*) month=7;;
    [Aa]ug*) month=8;;
    [Ss]ep*) month=9;;
    [Oo]ct*) month=10;;
    [Nn]ov*) month=11;;
    [Dd]ec*) month=12;;
esac

cal ${month} ${year}
```

VOLUME 9 EXERCISE - ANSWERS

```
$ mycal<CR>
      September 1983
   S  M Tu  W Th  F  S
                  1  2  3
   4  5  6  7  8  9 10
  11 12 13 14 15 16 17
  18 19 20 21 22 23 24
  25 26 27 28 29 30

  $ mycal oct<CR>
     October 1983
   S  M Tu  W Th  F  S
                     1
   2  3  4  5  6  7  8
   9 10 11 12 13 14 15
  16 17 18 19 20 21 22
  23 24 25 26 27 28 29
  30 31

 $ mycal Feb 1818<CR>
     February 1818
   S  M Tu  W Th  F  S
   1  2  3  4  5  6  7
   8  9 10 11 12 13 14
  15 16 17 18 19 20 21
  22 23 24 25 26 27 28

 $ mycal 4 1980<CR>
     April 1980
   S  M Tu  W Th  F  S
         1  2  3  4  5
   6  7  8  9 10 11 12
  13 14 15 16 17 18 19
  20 21 22 23 24 25 26
  27 28 29 30
```

VOLUME 9 EXERCISE - ANSWERS

3.
```
$ cat findit
#program name: findit
#  programmer: student
#   arguments: none

echo "`tput smso`Please enter name of file
       to be found: `tput sgr0`\c"
read name
echo "`tput smul`Please enter directory to begin search
       (default: current): `tput rmul`\c"
read dir
echo "`tput blink`The path of the file which
       matches that name is:`tput sgr0`"
find ${dir:=.} -name ${name:?"A name must be supplied, try again"}

$ findit
Please enter name of file to be found: myfile<CR>
Please enter directory to begin search (default: current): <CR>
The path of the file which matches that name is: ./bin/myfile
```

4.
```
$ cat diskchk<CR>
# program name: diskchk
#       Author: Ivan Itch
#    arguments: none
#      purpose: check on file systems running out of
#               blocks or inodes.  3000 is the default

df | sed "s/ *)/)/" | sed "s/  */      /g" |
while
read line
do
  if
     test `echo ${line} | cut -d" " -f3` -lt ${1-3000}
  then
     echo `echo ${line} | cut -d" " -f1,3,5` running out of blocks
  elif
     test `echo ${line} | cut -d" " -f5` -lt ${1-1000}
  then
     echo `echo ${line} | cut d" " -f1,3,5` running out of inodes
  fi
done

$
```

VOLUME 9 SUMMARY

Command	Meaning	Example
tput	Query terminfo database maintained in directory /usr/lib/terminfo.	$ tput clear <RET>
sed	Stream editor. Edits information either from standard input or from a file	$ sed "editor command" [file file ...] <RET>

Statement	Syntax

if-then-elif	if
	  commands
	  testing_command
	then
	  commands
	elif
	  commands
	  testing_command
	then
	  commands
	else
	  commands
	fi

VOLUME 10

Looping Statements Part 1

WHAT YOU'LL BE ABLE TO DO AFTER VOLUME 10

- Use the **while** statement to continually execute statements and commands
- Use the **shift** statement to alter the positional parameters
- Use the **getopt**(1) command to properly process command line options
- Use the **continue** and **break** statements

while STATEMENT

The **while** statement is used to continually (conditionally) execute statements or commands.

FORMAT

```
while
   command
      .
      .
      .
   testing_command
do
   command
      .
      .
      .
   command
done
```

There may be several commands or statements between the keywords **while** and **do**. The exit status of the last of those commands or statements determines whether the commands between **do** and **done** will be done.

It is important to know what the exit status is of the last command before **do**. Some commands do not supply proper exit status and should not be used as the testing condition of a **while** loop.

while STATEMENT — LOGIC

- PERFORM COMMANDS BETWEEN **while** AND **do**
- IF EXIT STATUS FROM LAST COMMAND BEFORE "**do**" IS "TRUE," THEN PERFORM COMMANDS BETWEEN **do** AND **done** AND THEN BACK TO **while**
- IF EXIT STATUS IS "FALSE," CONTINUE WITH NEXT STATEMENT AFTER **done**
- EXIT STATUS IS THAT OF LAST COMMAND EXECUTED IN **do-done** GROUP, IF ANY; OTHERWISE, IT IS 0 (TRUE)

while STATEMENT — EXAMPLE

```
$ cat phone<CR>
while
    echo "Enter name: \c"
    read name
    test "${name}" != ""
do
    direct "${name}"
done

$ phone<CR>
Enter name: hutch<CR>
981-2678    Robert Hutchison    hutch
Enter name: more<CR>
981-2677    John More           more
Enter name: <CR>

$
```

shift STATEMENT

FORMAT

```
shift [n]
```

- WILL SHIFT THE POSITIONAL PARAMETERS TO THE LEFT **n** POSITIONS
- **$**{**0**} WILL REMAIN UNCHANGED
- **$**{@} AND **$**{*} WILL BE CHANGED
- **$**{#} WILL BE DECREMENTED BY **n**
- USED TO ACCESS MORE THAN 9 POSITIONAL PARAMETERS

shift STATEMENT — EXAMPLE

The following example shows how options are typically processed. It uses a **while** loop, testing the value of **${1}** each time. The value of **${1}** changes, however, each time through the loop because there is a **shift** within the loop (at the end).

The **case** statement classifies each argument and prints the respective message. This program assumes that if the **-a** option is specified, then a file name must follow it directly.

```
$ cat options<CR>
while
    test "${1}" != ""
do
    case "${1}"
    in
    -a* )
        file_name=`echo ${1} | sed 's/-a//'`
        echo "\nOption  a  selected"
        echo "File name is ${file_name}"
        a_option="yes"
        ;;
    -b )
        echo "\nOption  b  selected"
        b_option="yes"
        ;;
    esac
    shift
done
echo "\nRest of program using file name = ${file_name}"
```

This procedure continually checks positional parameters until exhausted to see if they match either of the patterns **-asomething** or **-b**. If either of the patterns were matched, their respective lines of code are executed. After testing each argument the positional parameters are shifted to the left by one position, making the new value of **${1}** the old value of **${2}**, and so on.

shift STATEMENT — EXAMPLE

When the options procedure is run, the following prints:

EXAMPLE 1

```
$ options -aindex<CR>

Option  a  selected
File name is index

Rest of program using file name = index
```

EXAMPLE 2

```
$ options -b<CR>
Option  b  selected

Rest of program using file name =
```

shift STATEMENT — EXAMPLE

EXAMPLE 3

```
$ options -acatalogue -b<CR>

Option  a  selected
File name is catalogue

Option  b  selected

Rest of program using file name = catalogue

$
```

In the first example, the **-a** option is chosen and the file name *index* is specified.

The second example shows the **-b** option selected. In this example, file name is not set because the **-a** option is not selected.

The third example shows the combination of both the **-a** and the **-b** options. It is important to notice the order they are processed.

getopt COMMAND

FORMAT

```
set -- `getopt valid_options ${*}`
```

Always use the **getopt** command in programs that accept options. It allows the operator to enter the options (in any order) with or without matching arguments and produces an error message if an invalid argument is specified. The **getopt** command also permits the combining of options and separates them into separate options, each with their own preceding dash (-).

The **getopt** command uses the **valid-options** string to determine the list of valid options and which options require matching arguments. This is a string of option letters that may contain colons (:) after one or more of the letters. If the colon is present, then the option must be supplied a matching argument (with or without leading white space). If no argument is supplied to the option, then getopt will print an error message of the form:

```
getopt: option requires an argument -- option_letter
```

valid-options STRING EXAMPLE

```
abc:d:efg:
```

This string defines the list of valid options to be the options **a, b, c, d, e, f,** and **g**. Furthermore, the options **c, d,** and **g**, if supplied, require a matching argument. That argument may appear directly next to the option letter or may be separated from it with spaces.

The **getopt** command, if supplied with an option in which argument requires a matching argument, separates that argument from the argument letter, making them appear as two separate parameters.

After all of the options and before any of the arguments, the **getopt** command supplies another positional parameter with the value **--**. When a routine that processes options receives this positional parameter, it may assume that all of the other positional parameters are arguments and not options.

If an invalid option is entered, then the **getopt** command issues an error message and exits with a nonzero status. The form of the error is:

```
getopt: illegal option -- option_letter
```

It is usually the practice to test the exit status of the **getopt** command and exit if the status is nonzero.

getopt COMMAND

FORMAT

```
set -- `getopt valid_options ${*}`
```

- ALLOWS COMBINING OPTIONS — WILL SEPARATE
- ADDS A "--" AS A POSITIONAL PARAMETER AFTER LAST OPTION
- IF OPTION LETTER FOLLOWED BY COLON, THEN IT MUST BE ACCOMPANIED BY AN ARGUMENT
- ARGUMENTS MAY APPEAR NEXT TO OPTION LETTER OR SEPARATED BY SPACES
- CHECKS FOR VALID OPTIONS (IF NOT VALID, WILL PRINT AN ERROR MESSAGE)
- EXIT STATUS NONZERO IF INVALID OPTION SPECIFIED
- set -- DISABLES RECOGNITION OF SHELL OPTION REQUESTS (-x,-v,etc)

getopt COMMAND — EXAMPLE

The following example shows the **try.getopt** command being invoked with several different option combinations.

```
$ cat try.getopt<CR>
set -- 'getopt o:r:nt ${*}'
echo ${*}
```

EXAMPLE 1

```
$ try.getopt -ooutput.file -n -t<CR>
-o output.file -n -t --
```

EXAMPLE 2

```
$ try.getopt -tnooutput.file input.file<CR>
-t -n -o output.file -- input.file
```

EXAMPLE 3

```
$ try.getopt input.file<CR>
-- input.file

$
```

The first example uses this command with options, but no arguments, making the special argument **--** appear as the last argument. Also, note that the argument **output.file** was separated from its introducing option letter.

The second example shows the combining of options, following a single dash (**-**). The options **-t, -n,** and **-o** are combined following a single dash on the command line, but after the **getopt** command is executed, they all appear as separate arguments. Also, the argument **input.file** appears after the special argument **--**.

getopt COMMAND — EXAMPLE

The last example invokes the **try.getopt** command with no options, only an argument. In this case, the special argument **--** is placed at the beginning of the list.

getopt COMMAND — EXAMPLE

This program shows a typical usage of the **getopt** command.

```
$ cat try.getopt<CR>
set -- 'getopt o:r:nt ${*}'
if
    test ${?} != "0"
then
    exit 1
fi
while
    test "${1}" != "--"
do
    case "${1}"
    in
    -o )
        output_file="${2}"
        shift 2
        ;;
    -r )
        report_file="${2}"
        shift 2
        ;;
```

The first line of the program uses the **getopt** command to rearrange the options, accepting **o, r, n,** and **t** as valid options. If any other option is entered, the **getopt** command generates an error message and exits with a nonzero status.

The **if** statement causes the program to exit immediately if an invalid option is entered.

When the program determines that the command line is syntactically correct, it starts to process those options until it reaches the special argument **--**. If the **-o** option is entered, the variable **output_file** is set equal to the argument that followed the option letter. It may be assumed that the matching argument exists and is in the proper place because the **getopt** command ensured this. After processing this argument, the arguments are shifted to the left by two positions, one for the option letter and one for the matching argument.

getopt COMMAND — EXAMPLE

```
    -n )
        number_option="yes"
        shift
        ;;
    -t )
        heading="no"
        shift
        ;;
    * )
        echo Invalid
        shift
        ;;
    esac
done
shift  # to discard "--"
echo "Processing files ${*}"
echo "Output file = ${output_file}"
echo "Report file = ${report_file}"
echo "Numbering option = ${number_option:-no}"
echo "Heading option = ${heading:-yes}"
```

The **-n** and **-t** options require no matching arguments [a colon (:) does not follow the option letter in the **optstring** string of the **getopt** command line]. So when encountered, the positional parameters are only shifted once (to remove the option).

When all option processing is completed, the positional parameters are shifted once more to remove the special positional parameter **--**, introduced by the **getopt** command. Then, the program informs the user of the options selected and terminates. A typical use of the **getopt** command would do something with the positional parameters other than print them.

getopt COMMAND — EXAMPLE

```
$ try.getopt -tnooutput.file -r report.file unit.13<CR>
Processing files  unit.13
Output file = output.file
Report file = report.file
Numbering option = yes
Heading option = no

$ try.getopt -tn *<CR>
Processing files logfile phone try.getopt unit.07 unit.13
Output file =
Report file =
Numbering option = yes
Heading option = no

$ try.getopt unit.13<CR>
Processing files unit.13
Output file =
Report file =
Numbering option = no
Heading option = yes

$
```

The first invocation uses the **-t, -n, -r**, and **-o** options. The **-r** and **-o** options are to have matching arguments that are supplied in each case.

The second sample only uses the **-t** and **-n** options and supplies all of the files in the current directory as arguments.

Question: What performs the expansion of the expression * ? Is it the **getopt** command or the shell? If it is the shell, which shell does the expansion — the parent or the child?

Answer: The parent shell, because the command line is being interpreted by the parent shell.

continue STATEMENT

FORMAT

```
continue [n]
```

The **continue** statement is used to immediately return back to the top of a loop without continuing with the body of the loop. The **continue** statement may be used in any looping construct.

If [**n**] is supplied, the **continue** statement causes control to resume at the top of the last **n** looping constructs.

continue STATEMENT

FORMAT

```
continue [n]
```

- WILL RETURN BACK TO THE BEGINNING OF LAST [**n**] LOOPING STATEMENT(S) ENTERED
- TAKES EFFECT IMMEDIATELY

continue STATEMENT — EXAMPLE

In this example, it is desirable to end the directory lookup only when a control-d is entered in response to the name inquiry. When a **read** statement is supplied with a control-d as input, it returns a nonzero exit status, thereby ending the **while** loop.

If a null name is entered, it is treated as a nonfatal error and simply reprompts for the name.

```
$ cat try.continue<CR>
while
    echo "\nEnter name (control-d to end): \c"
    read name
do
    if
        test "${name}" = ""
    then
        continue
    else
        direct "${name}"
    fi

    if
        test ${?} != 0
    then
        echo Name not found
    fi
done
echo "\nEnd of directory inquiry"
```

When the **continue** is encountered (if a null name is entered), transfer of control resumes at the **while**.

continue STATEMENT — EXAMPLE

```
$ try.continue<CR>

Enter name (control-d to end): Annette
981-6861    Annette Vaness        vaness

Enter name (control-d to end):

Enter name (control-d to end): Milly Ampere
Name not found

Enter name (control-d to end): <control-d>
End of directory inquiry

$
```

This example, when provided with a name, calls the **direct** command to look up the phone number of that person. If a null name is entered, the user is reprompted for input. In the last example, the program ends when a null name is entered. This version only ends when a control-d is entered as the response to the prompt.

break STATEMENT

FORMAT

```
break [n]
```

The **break** statement immediately ends the last loop entered, or if a number (n) is supplied as an argument, breaks out of the last **n** looping constructs. Usually **break** is only used to break out of one looping construct.

This statement is not similar to that of the C language, which can be used to break out of looping constructs, but also is used to end a case section of a **switch** statement. In the shell programming language, the **break** statement is only used to terminate a loop or loops.

If **break** is used to break out of more than one looping construct, it should be well documented in the source telling from where it is breaking and to where.

break STATEMENT

FORMAT

```
break [n]
```

- WILL TERMINATE **n** LEVELS OF LOOPING CONSTRUCTS
- IS USED ONLY IN LOOPING STATEMENTS

break STATEMENT — EXAMPLE

```
$ cat try.break<CR>
while
    echo '\nEnter name ("done" to end): \c'
    read name
do
    if
        test "${name}" = "done"
    then
        break
    fi
    if
        test "${name}" = ""
    then
        continue
    else
        direct "${name}"
    fi
    if
        test ${?} != 0
    then
        echo Name not found
    fi
done
echo "\nEnd of directory inquiry"
```

This example is similar to the previous example, except that it terminates when the response **done** is entered. The **break** statement ends the last looping construct that was entered, thus ending the **while** statement. When this statement is finished, the program prints `"End of directory inquiry"` and exits.

Question: What will happen if the response to the prompt is a control-d? Will the program end? If it will, how can it be changed to prevent this?

Answer: Put the read statement between the do and done to prevent it from ending as a result of typing a control-d.

break STATEMENT — EXAMPLE

The following is a sample of the output generated when using the **try.break** program.

```
$ try.break<CR>

Enter name ("done" to end): pante
981-6160    Dom Pante         pante

Enter name ("done" to end):

Enter name ("done" to end): King Lear
Name not found

Enter name ("done" to end): done

End of directory inquiry

$
```

If a recognized name is entered, the **direct** command prints out the name and phone number. If it is not found, as with the case of King Lear, the **direct** command returns a nonzero exit status and the message `Name not found` prints. If a null name is entered, the program reprompts the user for input (as in the example above).

In this example, if the word **done** is entered, the program breaks out of the **while** loop, prints the message `End of directory inquiry`, and exits the program.

VOLUME 10 EXERCISE

1. Modify the progam given below to handle multiple arguments and the following options; **'-u'**, **'-g'** and **'-h'**. A '-u' option will print the user's uid number also. A '-g' option will print the user's gid number also. A '-h' option will print the user's fully qualified pathname to their login directory.
   Name the new shell procedure **ha_imr**.

   Note: The **getopt** command should be used before processing the option.

```
if
  test ${#} -lt 1
then
  echo Usage: ${0} login
  exit 2
fi
entry="`grep \^${1}: /etc/passwd`"
if
   test -z "${entry}"
then
   echo ${1} is not a valid login on `uname`.
   exit 1
else
   echo "${1}\t `echo ${entry} | cut -d'-' -f2 | cut -d'(' -f1`"
fi
```

EXAMPLE:

$ ha_imr -h cec21 cec18<CR>

```
cec21       Student 21      /class/chcec/cec21
cec18       Student 18      /class/chcec/cec18
```

$ ha_imr xxx<CR>

xxx is not a valid login on ltuxa.

$

VOLUME 10 EXERCISE - ANSWER

1. **$ cat ha_imr<CR>**

```
# users.name - print information about a user
# arguments  - valid login names
# options    -  -u print the user's uid number
#               -g print the user's gid number
#               -h print the user's login directory name
set -- 'getopt ugh ${*}'
if
       test ${?} != 0
then
       echo Usage: ${0} [-ugh] login [login...]
       exit 1
fi
heading="LOGIN  NAME "
while
        test "${1}" != "--"
do
        case "$1"
        in
             -u) uflag=1
                 heading="${heading} UID";;
             -g) gflag=1
                 heading="${heading} GID";;
             -h) hflag=1
                 heading="${heading} LOGIN DIR.";;
        esac
        shift
done
shift
if
       test ${#} -lt 1
then
       echo Usage: ${0} [-ugh] login [login...]
fi
if
        test ${#} -gt 1
then
        echo "${heading}"
        echo
fi
while
test "$1"
do
        entry="'grep \^${1}: /etc/passwd'"
        if
            test -z "${entry}"
```

```
        then
            echo ${1} is not a valid login on `uname`
        else
        echo "${1}\t `echo ${entry} | cut -d'-' -f2 | cut -d'(' -f1` \c"
        fi
        if
            test "${uflag}" -eq 1
        then
            echo \\t `echo ${entry} | cut -d':' -f3` \\c
        fi
        if
            test "${gflag}" -eq 1
        then
            echo \\t `echo ${entry} | cut -d':' -f4` \\c
        fi
        if
            test "${hflag}" -eq 1
        then
            echo \\t `echo ${entry} | cut -d':' -f6` \\c
        fi
        echo
shift
done
```

```
$ ha_imr cec10 cec12<CR>
LOGIN        NAME
cec10       Student10
cec12       Student12

$ ha_imr -u cec1 cec21<CR>
LOGIN        NAME           UID
cec1        Student1        1001
cec21       Student21       1021

$ ha_imr -h cec4 cec11<CR>
LOGIN        NAME           LOGIN DIR.
cec4        Student4        /class/chcec/cec4
cec11       Student11       /class/chcec/cec11

$ ha_imr -uh cec4 cec11<CR>
LOGIN        NAME           UID     LOGIN DIR.
cec4        Student4        1004    /class/chcec/cec4
cec11       Student11       1011    /class/chcec/cec11
```

VOLUME 10 SUMMARY

Command	Meaning	Format
getopt	Allows combining options Will separate options Will issue an error message if invalid option entered Used in shell procedures which accept options	set -- 'getopt valid_options ${*}'

Statement	Meaning	Format
shift	Used to change values of positional parameters, shifting them to the left by n number of positions	shift [n]
continue	Will return to beginning of last [n] looping constructs entered	continue [n]
break	Will terminate n levels of looping constructs	break [n]

Statement	Syntax

while	while
	  commands
	  testing_command
	do
	  commands
	done

VOLUME 11

Looping Statements, Part 2

WHAT YOU'LL BE ABLE TO DO AFTER VOLUME 11

- Use the **for** statement to continually execute commands or statements
- Use the **expr**(1) command to perform arithmetic comparisons and computations
- Use the **until** statement to continually execute commands or statements
- Use the **line**(1) command to read input from the standard input.

for STATEMENT

The **for** statement is used to do a series of commands a fixed number of times.

FORMAT

```
for variable_name
in value ... value
do
    command
        .
        .
        .
    command ${variable_name}
        .
        .
        .
    command
done
```

The commands between **do** and **done** are done as many times as there are values following the keyword **in**. Each time through the loop, the variable as indicated by **variable_name** takes on a new value from the list of values. The first time through the loop, **variable_name** has the value of the first value, the second time, the second value, and so on.

for STATEMENT — EXAMPLE

This example causes the files specified to be run through the **mm** program separately and for the output to be stored in a file with a similar name (with **mm.** prepended to the name).

```
$ pr -tn mm.mult<CR>
1   for file_name
2   in ${*} # this is the default case
3   do
4       mm ${file_name} > mm.${file_name}
5   done

$ mm.mult unit.13 unit.14<CR>

$
```

This invocation of the **mm.mult** command causes the files *unit.13* and *unit.14* to be **mm**'ed and the output to be stored in files named *mm.unit.13* and *mm.unit.14*, respectively.

Line 2 indicates the values that the variable file name has for each iteration of the **for** loop. The first time through, the variable file name has the value *unit.13*; the second time, it has the value *unit.14*.

> **Note:** If the line that begins with the keyword **in** is omitted, then the **for** loop assumes that the specified variable will take on the values of all of the positional parameters.

for STATEMENT — EXAMPLE

The following example expects the name of a command to be supplied as an argument. It then searches the **PATH** variable directory path names to see where the command is located. When the command is found, the entire path name is printed.

This program is useful to reference a command when the location of the command is not known. It is also useful when several versions of the same program exist and the user wants to see which version will be accessed first (check arrangement of the **PATH** variable).

```
$ cat where<CR>
IFS="${IFS}:"
for name
in ${PATH}
do
    if
        test -x "${name}/${1}" -a ! -d "${name}/${1}"
    then
        echo "${name}/${1}"
    elif
        test -f "${name}/${1}"
    then
        echo "${name}/${1}  ---not executable---"
    fi
done

$ where ed<CR>
/bin/ed

$ where kdefs<CR>
/isg/caiac/2.4/bin/kdefs
/std/bin/kdefs    --- not executable ---
```

The value of **IFS** has been modified to include a colon (:) (the **PATH** delimiter). At this point, when the value of **PATH** is referenced, it appears as several arguments, not as a long string of characters, and it becomes useful in constructs such as **for**.

expr COMMAND

The **expr** command is used to evaluate an expression. This expression may either be a string comparison, an arithmetic expression, or a pattern matching expression. In this course, the first two capabilities are discussed.

FORMAT

```
$ expr arguments<CR>
```

The supplied arguments make up the expression to be evaluated. It is important to remember that the words that make up the expression must be separated by blanks in order for them to appear as arguments. Additionally, if any characters are used in these expressions that are recognized by the shell, they must be escaped.

String Comparison:

- expression = expression
- expression \> expression
- expression \< expression
- expression \<= expression
- expression \>= expression
- expression != expression

In the case of numeric comparisons, the standard output of the **expr** command is the truth value of the expression. This value is nonzero if the expression is true or zero, (0) if the expression is false. These expressions may also be combined with the logical operators | and **&**, but those characters need to be escaped because of their special meaning to the shell. It is unfortunate that the return values are inconsistent with those of the shell.

If the value is to be evaluated or assigned to a variable, the command line should appear in a command substitution expression.

```
$ echo ${TERM}
5420

$ expr ${TERM} = 5420
1

$ expr ${TERM} = 5620
0

$
```

expr COMMAND

FORMAT

```
$ expr arguments<CR>
```

- ARGUMENTS MAKE UP EXPRESSION
- COMPARISON
  - — expression = expression
  - — expression \> expression
  - — expression \< expression
  - — expression \<= expression
  - — expression \>= expression
  - — expression != expression
- WILL PRINT 1 IF TRUE, 0 IF FALSE

expr COMMAND

- ARITHMETIC CALCULATIONS
  - — expression + expression
  - — expression - expression
  - — expression * expression
  - — expression / expression
  - — expression % expression
- INTEGER ARITHMETIC ONLY
- STRINGS CONTAINING INTEGERS ONLY

expr COMMAND — EXAMPLE

```
$ cat front<CR>
i=1
while
    test "${i}" -le "${1}"
do
    sed -n ${i}p "${2}"
    i=`expr ${i} + 1`
done

$ front 4 front<CR>
i=1
while
    test "${i}" -le "${1}"
do

$
```

expr COMMAND — EXAMPLE

This example determines the time of day and prints an appropriate word of greeting. It uses command substitution expressions nested two deep (the inner command substitution backwards quotes need to be escaped).

The hour is taken first, then it is numerically tested against a fixed hour and then the truth value of the **expr** is tested, and if zero, the message prints.

```
$ cat greeting<CR>
if
    test `expr \`date +%H\` \< 12` = "1"
then
    echo "Good Morning"
elif
    test `expr \`date +%H\` \< 18` = "1"
then
    echo "Good Afternoon"
else
    echo "Good Evening"
fi

$ greeting<CR>
Good Morning

$
```

until STATEMENT

The **until** statement is the complement of the **while** statement. It performs the commands in between the keywords **until** and **do**. It then evaluates the exit status of the last of those commands. In the case of the **until** statement, the commands between the **do** and the **done** are only done if the exit status of the last of those commands is false.

FORMAT

```
until
    command
        .
        .
    testing_command
do
    command
        .
        .
    command
done
```

The statements between **until** and **do** are always done. If the exit status of the command before **do** is false, the commands between **do** and **done** are performed. After those commands are performed, transfer of control resumes at the keyword **until**.

until STATEMENT

FORMAT

```
until
        command
            .
            .
        testing_command
do
        command
            .
            .
        command
done
```

- COMPLEMENT OF **while** STATEMENT
- PERFORM COMMANDS BETWEEN **until** AND **do**
- IF EXIT STATUS OF LAST COMMAND IS FALSE (NONZERO), DO COMMANDS BETWEEN **do** AND **done** AND GO BACK TO **until**

until STATEMENT — EXAMPLE

The following example is similar to that example in the **getopt** lesson. It processes command line options and then prints which options were chosen. The only difference is that the test is testing to see if the argument is equal to -- instead of not equal to.

```
set -- `getopt o:r:nt ${*}`
if
    test ${?} != "0"
then
    exit 1
fi
until
    test "${1}" = "--"
do
    case "${1}"
    in
    -o )
        output_file="${2}"
        shift 2
        ;;
    -r )
        report_file="${2}"
        shift 2
        ;;
```

until STATEMENT — EXAMPLE

```
        -n )
            number_option="yes"
            shift
            ;;
        -t )
            heading="no"
            shift
            ;;
        * )
            echo Invalid
            shift
            ;;
        esac
    done
    shift
    echo "Processing files ${*}"
    echo "Output file = ${output_file}"
    echo "Report file = ${report_file}"
    echo "Numbering option = ${number_option:-no}"
    echo "Heading option = ${heading:-yes}"
```

This program is logically equivalent to the **getopt** example, but is more readable. The control construct better represents the English equivalent sentence. The **getopt** example uses two negatives in the same command line, and everyone knows "you should never use no double negatives."

line COMMAND

The **line** command reads one line of input from the standard input (if no file name is specified) and prints it to the standard output. If more than one line is supplied, the **line** command only prints the first of the supplied lines.

FORMAT

```
$ line<CR>
```

The **line** command is often used to take information from the terminal. This is not the best way of getting information from the terminal — the better approach is to use the **read** statement. The **read** statement is better because it is a built-in part of the shell (it is more efficient). The problem with the **read** statement is that it may not have either its input or output redirected (although its process may have), whereas redirection is possible with the **line** command.

The **line** command is often used as a filter to only pass through the first line of the output and disregard the rest.

line COMMAND

FORMAT

```
$ line<CR>
```

- READS ONE LINE AND WRITES IT TO THE STANDARD OUTPUT
- MAY BE USED TO GET LINES ONE AT A TIME FROM STANDARD INPUT
- USEFUL AS A FILTER TO GET FIRST LINE
- DOES NOT STRIP **IFS** CHARACTERS FROM INPUT (UNLIKE **read**)

line COMMAND — EXAMPLE

This example prints the line number of the first line in a specified file that matches the specified pattern. When **grep** is used with the **-n** option, it prints out the matching lines with line numbers attached to the beginning. The **line** command only passes to the **sed** command the first line that matched. The **sed** command removes all of the line except the line number; and because the command line is in a command substitution expression, the value of the line is stored in the variable named **line_number**. The program then prints that line number.

```
$ cat trep<CR>
if
    test "${#}" -lt 2
then
    echo "Usage: ${0} pattern file"
    exit 1
fi
pattern="${1}"
file_name="${2}"
line_number=`grep -n "${pattern}" "${file_name}" |
                    line |
                    sed  's/:.*//'`
echo "Found on line ${line_number}"

$ trep 'ne 7' test.file<CR>
Found on line 7

$
```

VOLUME 11 EXERCISE

1. Write a shell procedure that will:

   i. Accept a file name as a positional parameter.

   ii. Print the file name if the specified file name is that of an executable file.

   A. Name this procedure **ckpath**.

   B. Modify the program to accept a **-p** option followed by a directory path name. If the **-p** option is selected, the file that should be tested will be in the specified directory. If the file is in the specified directory and is executable, the entire path name should be printed.

      Note: The **getopt** command should be used before processing the option. This will allow the directory path name to immediately follow the **-p** or be separated from it by spaces.

EXAMPLES

```
$ ckpath ckpath<CR>
ckpath

$ ckpath -p ${HOME} ckpath<CR>
/ustg/hutch/ckpath

$ ckpath -p ${HOME} non.exec<CR>
```

VOLUME 11 EXERCISE

C. Modify the procedure to check for several files as arguments. The procedure should do steps i and ii for each file specified. If the **-p** option was selected, **ckpath** should assume that all the specified file names are in the directory whose names accompany the **-p** option.

EXAMPLES

```
$ ckpath prog1 prog2 text2 prog3<CR>
prog1
prog2
prog3

$ ckpath -p /ustg/hutch text1 prog5 prog6 text4<CR>
/ustg/hutch/prog5
/ustg/hutch/prog6
```

D. Modify **ckpath** to search the **PATH** variable for the specified file names if the **-p** option was not selected.

EXAMPLES

```
$ ckpath ed where prog5 text4<CR>
/bin/ed
/std/bin/where
/ustg/hutch/bin/prog5

$ ckpath -p /bin ed where prog5 text4<CR>
/bin/ed

$
```

VOLUME 11 EXERCISE

E. Modify the procedure to accept an **-s** flag, which will cause the test when searching the **PATH** to only check if the file exists and has a size greater than zero.

HINT: The **-s** option, to test, checks to see if a file exists and has a nonzero size.

EXAMPLES

```
$ ckpath -s -p${HOME} non.exec prog1<CR>
/ustg/hutch/non.exec
/ustg/hutch/prog1
```

VOLUME 11 EXERCISE - ANSWERS

1. `$ cat ckpath<CR>`

```
# program name: ckpath
#       author: Hill, Murray
#    arguments: command name(s)
#      options: -p  - followed by directory path_name
#                     will only search specified dir.
#               -s  - will only test for file existence
#      purpose: Will print the path_name of an executable
#               file.  Will search PATH if -p option is
#               not selected.  If -p is selected will
#               search specified directory.  Will print
#               complete path_name of executable file or
#               nothing otherwise.

set -- `getopt sp: ${@}`
if test ${?} -ne 0
then
    echo Usage: ${0} [ -p path ] [ -s ] file_name...
    exit 2
fi
while
    test "${1}" != "--"
do
    case "${1}"
    in
    -s )
        flag="-s"
        shift
        ;;
    -p )
        path="${2}"
        shift 2
        ;;
    esac
done
shift
path=`echo ${path:-${PATH}} |
    sed -e 's/^:/.:/' -e 's/:$/:./' -e 's/::/:.:/'`
: ${flag:="-x"}
IFS="${IFS}:"
for argument
in ${*}
do
    for path_seg
    in ${path}
    do
```

```
        if
            test ${flag} "${path_seg}/${argument}"
        then
            echo ${path_seg}/${argument}
        fi
    done
done
```

VOLUME 11 SUMMARY

Command	Meaning	Example
expr	Used to evaluate either a string comparison or an arithmetic expression	$ expr ${TERM} = 5425 <RET>
line	Reads one line of input from standard input and prints it to standard output. Redirection is possible when using the line command.	$ line <RET>

Statement	Syntax
for	for var_name in value ... value do commands done

until	until
	  commands
	  testing_command
	do
	  commands
	done

VOLUME 12

Command Line Interpretation

WHAT YOU'LL BE ABLE TO DO AFTER VOLUME 12

- Describe the nine functions of the shell interpreter
- Describe redirection in terms of the shell
- Use nested command substitution expressions.

THE SHELL COMMAND LINE

ORDER OF EVALUATION

- READ COMMAND LINE OR CONSTRUCT
- PRINT IF VERBOSE TRACE
- VARIABLE SUBSTITUTION
- COMMAND SUBSTITUTION
- REDIRECT I/O
- IFS PROCESSING
- METACHARACTER EXPANSION FOR FILE NAME GENERATION
- EXECUTION TRACE
- ENVIRONMENT PROCESSING
- EXECUTE COMMAND

READ COMMAND LINE

The first step the shell performs is to read a command line from a file and initially parse the line with respect to spaces and tabs. This line may come from a file. That file may be either the terminal special file or a UNIX System file.

The shell reads until it encounters a semicolon (;), background invocation symbol (&), **logical and** (&&), **logical or** (||), or a newline character (NL). If it detects that a shell construct is being read in, the shell reads until it finds the end of that construct.

After the shell has read in the command line or construct, it parses the line into a series of words. The shell determines that a word is either anything at the beginning of the line, the end of the line, or surrounded by spaces or tabs. This parsing is always done, regardless of the value of the **IFS** variable. A reparsing of the line with respect to the IFS-contained characters is done at a later time.

VERBOSE TRACE PRINT

The next step the shell performs is to see if the verbose trace is set. If it is, the shell prints the command line or construct to the standard error output.

Several things are inherited by child processes, such as the environment, open files, etc. The verbose trace setting is not. This means that if the verbose trace is set for a particular shell, it echoes the command lines from that file and not the lines of shell programs that the program executes.

VARIABLE SUBSTITUTION

- POSITIONAL PARAMETER, VARIABLE, SPECIAL SUBSTITUTION EXPRESSIONS
- EXPRESSION REPLACED BY VALUE OF EXPRESSION
- IF QUOTED, REMAINS AS ONE ARGUMENT
- ASSUMES THE VALUE OF UNSET VARIABLE IS NULL

COMMAND SUBSTITUTION

A command substitution expression is a command line enclosed in backward quotes. The value of that expression is that of the output of the command line. The command line may be anything that could be typed from the terminal, including sequentially executed commands, pipelines, command groups, etc.

The command line that appears as part of the command substitution expression may use any of the facilities of the shell such as parameter substitution, file name generation, etc. The entire command line is completely evaluated by the shell before executing the command line.

If the command generates extra blanks, tabs, or newlines, the shell removes them when it reparses the command line with respect to the characters contained in the IFS variable. If these extra blanks, tabs, or newlines should be preserved, enclose the entire expression in double quotes.

Command substitution expressions may be nested by escaping the inner backward quotes. Nesting command substitution expressions two deep requires a single backslash before the inner backward quotes. Nesting three deep requires **five** backslashes.

COMMAND SUBSTITUTION

- COMMAND ENCLOSED IN GRAVE ACCENTS (BACKWARD QUOTES)
- EXPRESSION REPLACED BY OUTPUT OF COMMAND
- MAY BE NESTED — INNERMOST ACCENTS GRAVES MUST BE ESCAPED WITH BACKSLASHES TO DISTINGUISH FROM OUTER ACCENTS
- COMMAND LINE BETWEEN GRAVE ACCENTS WILL BE COMPLETELY EVALUATED BEFORE EXECUTED
- MAY USE ANY FACILITY OF THE SHELL (VARIABLE SUBSTITUTION EXPRESSIONS, FILE NAME GENERATION CHARACTERS, ETC.) WITHIN COMMAND SUBSTITUTION

COMMAND SUBSTITUTION — EXAMPLE

The following example shows the nesting of command substitution expressions two deep. Note that the innermost backward quotes are escaped through the use of a backslash character. This program first finds out today's date, and then uses that as the pattern to supply to the **grep** command in searching the file name stored in the variable **file_name**. It then assigns that value to the variable **appoint** and prints the result.

```
$ cat is.today<CR>
for file_name
in ${*}
do
    appoint="`grep \`date +%m/%d/%y\` ${file_name}`"
    echo "${appoint}"
done

$ cat calendar<CR>
04/14/87    Finish unit.15
04/15/87    Finish unit.16
04/15/87    Finish unit.17

$ is.today calendar<CR>
04/14/87    Finish unit.15

$
```

I/O REDIRECTION

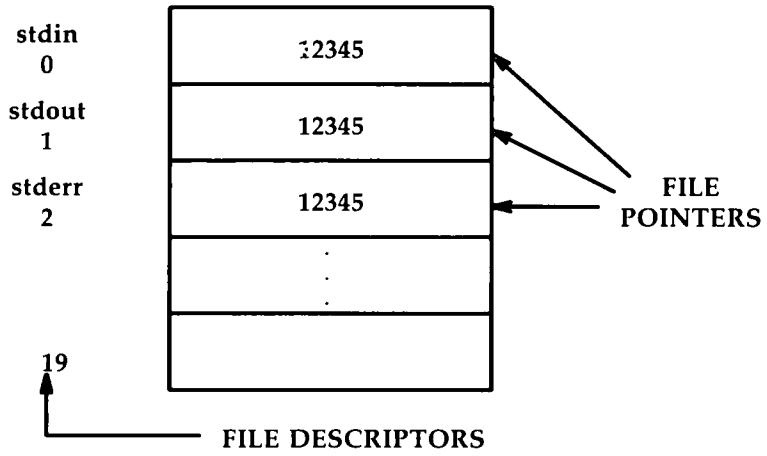
- REDIRECTION PROCESSED, AND EXPRESSION IS REMOVED FROM COMMAND LINE
- FILE DESCRIPTOR TABLE (OPERATING SYSTEM STRUCTURE)
- READING IS DONE FROM FILE DESCRIPTOR
- WRITING IS DONE TO FILE DESCRIPTOR
- ENTRIES ARE CALLED "FILE POINTERS"
  - — ENTRIES POINT TO INFORMATION ABOUT HOW A FILE IS OPENED (CALLED "FILE POINTERS")
  - — ENTRY 0 POINTS TO INFORMATION ABOUT THE OPEN OF THE TERMINAL FOR INPUT — SYNONYM IS "STANDARD INPUT"
  - — ENTRIES 1 AND 2 POINT TO INFORMATION ABOUT THE OPEN OF THE TERMINAL SPECIAL FILE FOR OUTPUT — SYNONYMS "STANDARD OUTPUT" AND "STANDARD ERROR"

I/O REDIRECTION — FILE DESCRIPTOR TABLE

File descriptor table entries 0, 1, and 2 are reserved for the standard input, output, and error output. When a user logs in, these three file descriptor entries are populated when the terminal special file is opened. They initially point to the same file table entry. The terminal special file is opened three times, once for input and the other two times for output. Normally when a command reads information, it reads from file descriptor number 0; and when writing output, it writes to file descriptor 1. When commands write error messages, these messages are usually written to file descriptor number 2.

When input or output is redirected, the shell performs several steps. They involve the closing of that file descriptor (0 if standard input is being redirected, 1 if standard output is being redirected), the opening of the file, and the placing of the new file pointer in the file descriptor slot recently vacated by the closing of the original file.

I/O REDIRECTION
FILE DESCRIPTOR TABLE



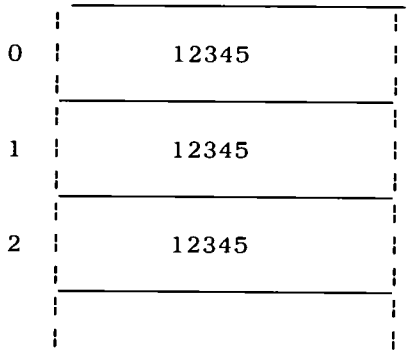
I/O REDIRECTION

- INPUT REDIRECTION
  - — "STANDARD INPUT" IS CLOSED (FOR THE CHILD PROCESS)
  - — NEW FILE IS OPENED (FILE SPECIFIED IN REDIRECTION EXPRESSION)
  - — FILE POINTER OF NEWLY OPENED FILE IS PLACED OVER THAT OF CLOSED STANDARD INPUT
- SAME METHOD FOR REDIRECTING "STANDARD OUTPUT" OR "STANDARD ERROR"

I/O REDIRECTION — EXAMPLE

The following is an illustration of the first few entries in a file descriptor table. Normally, a file descriptor table has 20 entries, but the shell may only reference the first 10 entries of that table. How to access files other than standard input, output, and error output is discussed in Volume 15.

This is before redirection was processed. The number 12345 points to an area of memory that contains information about how the standard input file was opened.



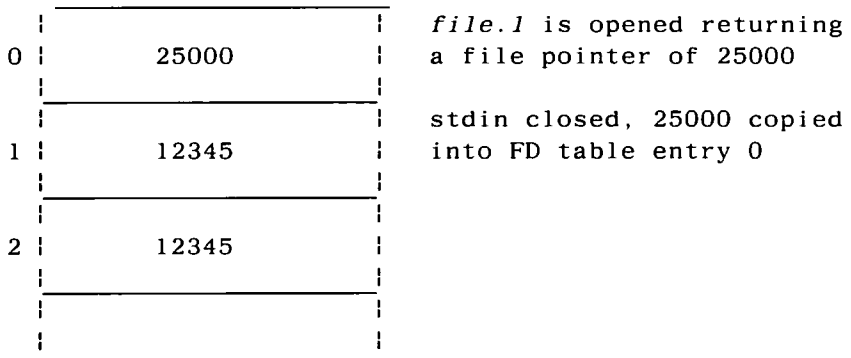
BEFORE REDIRECTION

I/O REDIRECTION — EXAMPLE

In this example, standard input is redirected from a file named *file.1*. The shell first closes file descriptor number 0 and opens the file *file.1* for input. It then copies the file pointer generated by that open into the file descriptor entry 0.

```
$ command < file.1<CR>
```

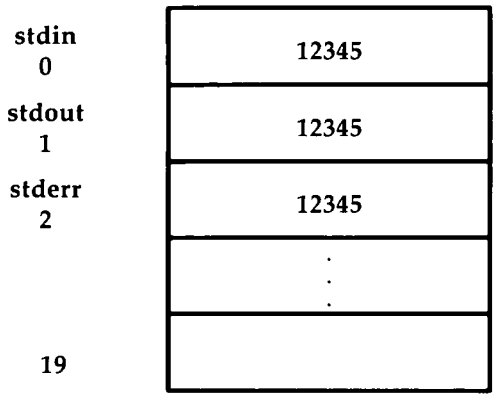
At this point, when a command reads from file descriptor 0, it is reading from the newly opened file and not the terminal. Take similar actions if output or error output is being redirected.



AFTER REDIRECTION

At this point, when a command reads from file descriptor 0, it is reading from the newly opened file and not the terminal. The shell takes similar actions if output or error output is being redirected.

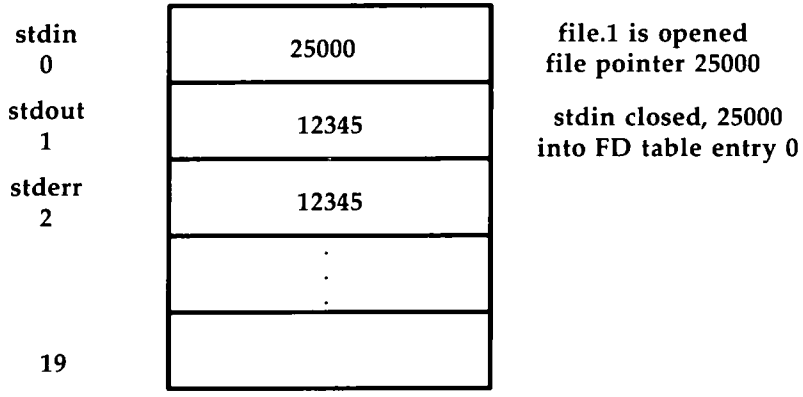
I/O REDIRECTION — EXAMPLE



BEFORE REDIRECTION

I/O REDIRECTION — EXAMPLE

```
$ command < file.1<CR>
```



AFTER REDIRECTION

IFS PROCESSING

- REPARSES LINE USING CHARACTERS CONTAINED IN THE IFS VARIABLE TO DETERMINE WORDS
- QUOTED NULL ARGUMENTS ARE RETAINED (e.g., " " OR '')
- UNQUOTED NULL ARGUMENTS ARE REMOVED
- REPLACES NEWLINE CHARACTERS (AS MIGHT BE GENERATED BY COMMAND SUBSTITUTION) WITH SPACES
- VARIABLE ASSIGNMENT STATEMENTS ARE PROTECTED FROM IFS PROCESSING AND FILE NAME GENERATION

METACHARACTER EXPANSION

The next step taken by the shell is to look for words that contain any file name generation characters (metacharacters). If any of the words on the command line contain any of these characters, the shell tries to pattern match that word against the file names in the current directory. If any file names match, they replace the metacharacter expression on the command line. If there is no match, the metacharacter expression remains on the command line.

The following example shows the variable **xpand** being set to the literal characters **Files are ***. The string being assigned to the variable **xpand** is enclosed in double quotes because the string is made up of more than one word.

```
$ xpand="Files are *"<CR>

$ echo "${xpand}"<CR>
Files are *

$ echo "${xpand}"<CR>
Files are file.1 file.2 file.3 file.4 file.5 file.6

$
```

When the value of **xpand** is echoed and the value is enclosed in double quotes, variable expansion is done, but the file name generation is not performed. When the same command line is executed without the double quotes, then the asterisk is expanded.

Like IFS parsing, variable assignment statements are protected from file name generation. If the assignment

star=*

were made, the value of the variable **star** is an asterisk, not all of the names in the current directory. It never hurts to enclose variable assignment strings in double quotes; variable substitutions and command substitutions will be performed anyway. It is necessary, when taking the value of a variable such as **star**, to enclose the expression in double quotes if the asterisk is not to be expanded.

METACHARACTER EXPANSION

Metacharacters, or file name generation characters, are as follows:

***** matches any *number of* ASCII characters

? matches any *single* ASCII character

[xy] matches one character if it is one of the characters from the set listed in the character class.

If the first character of the character class is "!", the expression matches any character that is **not** listed in the character class.

Ranges of characters are represented with a dash (-) between two other characters. However, if a dash appears at the beginning of the character class, it has no special meaning.

It is important to note that file name generation is done **after** parameter, command substitution, and I/O redirection. Therefore, it should never be used carelessly, especially in I/O redirection expressions. For example, the command line

who >*

produces a file with the name *, which is not nearly the expected result. And when removing that file.....watch out!

METACHARACTER EXPANSION

- ANY WORDS CONTAINING A METACHARACTER (FILE NAME GENERATION CHARACTER)
- MATCHES FILE NAMES FROM CURRENT OR SPECIFIED DIRECTORY
- IF NO MATCH, PATTERN REMAINS AND IS GIVEN TO COMMAND TO BE EXECUTED
- VARIABLE ASSIGNMENT STATEMENTS ARE PROTECTED FROM METACHARACTER EXPANSION

EXAMPLE

```
$ xpand=Files\ are\ *<CR>

$ echo "${xpand}"<CR>
Files are *

$ echo ${xpand}<CR>
Files are file.1 file.2 file.3 file.4 file.5
file.6

$
```

METACHARACTER EXPANSION

- EXPANSIONS

  ***** MATCHES ANY NUMBER OF ASCII CHARACTERS

  ? MATCHES ANY SINGLE ASCII CHARACTER

  [xy] MATCHES ONE CHARACTER IF IT IS ONE OF THE CHARACTERS FROM THE SET LISTED IN THE CHARACTER CLASS

- IF THE FIRST CHARACTER IS "!", THE EXPANSION WILL MATCH ANY CHARACTER NOT PART OF THE CHARACTER CLASS
- A DASH (-) AT THE BEGINNING OR END OF THE CHARACTER CLASS HAS NO SPECIAL MEANING

EXECUTION TRACE

- COMMAND LINES PRINTED ON STANDARD ERROR PRIOR TO EXECUTION
- COMMAND LINES PRECEDED WITH A PLUS (+) ON OUTPUT
- VARIABLE ASSIGNMENTS SHOWN WITHOUT THE PRECEDING PLUS (+)
- COMMAND SUBSTITUTION EXPRESSIONS APPEAR BEFORE THE PRINTING OF THE COMMAND LINE THAT CONTAINED THE EXPRESSION
- ORDER OF PARTS OF A PIPELINE NOT DEFINED — WILL SHOW SEPARATELY

ENVIRONMENT PROCESSING

The next step in command line evaluation or interpretation is to assign variables and search the **PATH** variable for the location of the command file.

Variable assignments are done from right to left when more than one assignment appears on a command line. It is important to realize the difference between these multiple assignments and keyword parameters.

The **PATH** variable is then searched for the location of the command file. The complete pathname of that file replaces the command name on the command line. If the command name contains a slash anywhere, the **PATH** variable is not searched and it is assumed that the command name is a fully qualified path to the file, whether absolute or relative.

ENVIRONMENT PROCESSING

- VARIABLE ASSIGNMENTS
  - — MULTIPLE ASSIGNMENTS SEPARATED WITH SPACES
  - — ASSIGNMENTS MADE FROM RIGHT TO LEFT
- PATH SEARCHED FROM LEFT TO RIGHT
- NULL DIRECTORY PATH NAME IMPLIES CURRENT DIRECTORY
- SEE LOGIC FOR PATH SEARCH

EXECUTE COMMAND

- CHILD PROCESS TRIES TO LOAD THE COMMAND FILE INTO MEMORY
- ASSUMES COMPILED PROGRAM, USES **exec** PRIMITIVE
- IF MODE IS EXECUTABLE AND FILE IS NOT LOADABLE, THE CHILD SHELL PROCESS ASSUMES IT IS A SHELL PROCEDURE AND INTERPRETS IT
- BUILT-IN STATEMENTS ARE NOT EXECUTED BY A SEPARATE PROCESS
- IF EXECUTED AS A FOREGROUND PROCESS, THE INVOKING SHELL WAITS FOR THE CHILD SHELL TO COMPLETE
- IF EXECUTED AS A BACKGROUND PROCESS, A PROCESS ID NUMBER IS PRINTED AND THE INVOKING SHELL IMMEDIATELY RESUMES EXECUTION

VOLUME 12 EXERCISE

1. A command line substitution is a command line enclosed in _____ quotes.

2. The grep command is used in the following shell procedure **is.today**. What if the output from the **grep** command exceeded one line? Would all of the output still print on separate lines as they appear in the file *calendar*?

```
$ cat is.today<CR>
for file_name
in ${*}
do
    appoint="`grep \`date +%m/%d/%y\` ${file_name}`"
    echo "${appoint}"
done

$ cat calendar<CR>
04/14/87    Finish unit.15
04/15/87    Finish unit.16
04/15/87    Finish unit.17

$ is.today calendar<CR>
04/14/87    Finish unit.15
```

3. List three reserved file descriptor table entries.

VOLUME 12 EXERCISE - ANSWERS

1. A command line substitution expression is a command line enclosed in **back** quotes.

2. Yes, the output will print on separate lines as they appear in the file calendar, because in line 5 of the program the variable appoint has been double quoted to preserve the <CR>

```
$ is.today calendar <CR>

04/15/87    Finish unit.16
04/15/87    Finish unit.17
```

3. File descriptor **0** for standard input. File descriptor **1** for standard output. File descriptor **2** for standard error.

VOLUME 12 SUMMARY

Special Character	Meaning
*	Matches zero or more ASCII characters.
?	Matches a single ASCII character.
[xy]	Matches one character from the set listed in the character class.
[!xy]	Matches one character not part of the character class.

VOLUME 13

Built-In Statements

WHAT YOU'LL BE ABLE TO DO AFTER VOLUME 13

- Use the built-in statements in writing shell programs
- Use the dot (.) statement in packaging systems
- Use the **eval** command in processing shell commands
- Limit file sizes using the **ulimit** statement.

null (:) STATEMENT

The **null** statement will do nothing and return an exit status of zero. The important thing that happens is that the complete command line is evaluated. This means that if the command line contains any expressions that involve side effects, those side effects will take place. For example, if a command line has a special substitution expression, a variable assignment might be performed. An example is

```
: ${TERM:=745}
```

Another type of expression that has side effects is a command substitution expression. This involves the execution of a command that might change the environment, files, etc.

null (:) STATEMENT

- SATISFIES SHELL'S NEED TO FIND A COMMAND ON THE COMMAND LINE
- ALLOWS REST OF COMMAND LINE TO BE EVALUATED
- IS USED WITH SPECIAL SUBSTITUTION EXPRESSIONS

. STATEMENT

The dot (.) statement is used to execute shell programs. The difference between this method and either explicitly or implicitly invoking shell programs is that a child process is not created. The shell interpreter from which the command line was invoked is the interpreter that will interpret and process the lines from that file.

The format for the dot (.) statement is as follows

```
$ . file_name<CR>
```

file_name is the name of a shell command file found in a directory listed in the **PATH** variable or a fully qualified path name. The shell takes this command name and searches the **PATH** variable for the location of that file and then causes the invoking shell process to interpret the lines in that file.

. STATEMENT

FORMAT

```
$ . file_name<CR>
```

- ALTERNATE METHOD TO EXECUTE SHELL PROCEDURES
- CAUSES LINES TO BE INTERPRETED BY THE CURRENT SHELL
- WILL NOT CREATE CHILD PROCESS TO INTERPRET COMMAND LINES OF FILE
- USEFUL FOR SETTING VARIABLES
- PATH IS SEARCHED TO FIND *file_name*
- I/O REDIRECTION IS PERMITTED
- CAN BE USED LIKE "SUBROUTINES"
- BEST VIEWED AS A TEXT-INSERTION FACILITY
- FILE MUST BE READABLE

. STATEMENT — DEFINITIONS FILES

- USED IN PACKAGING SOFTWARE
- CONTAINS INSTALLATION DEPENDENT VARIABLE SETTINGS
- RUN BY ALL PROCEDURES THAT ARE PART OF THE SYSTEM TO SET UP OPERATING ENVIRONMENT
- RUN AS A DOT (.) PROCEDURE
- CAN BE NESTED

. STATEMENT — EXAMPLE

```
$ cat definitions<CR>
CAIAC=/std/caiac/2.4
KBIN=/std/caiac/2.4/bin:/std/caiac/bin
KMAIN=/std/caiac/2.4/topic/menu
KTOPIC=/std/caiac/2.4/courseware
export CAIAC KBIN KMAIN KTOPIC

$ cat use.defs<CR>
#     file definitions used to set
#     up environment

. definitions

echo The CAIAC root is "${CAIAC}"

$ use.defs<CR>
The CAIAC root is /std/caiac/2.4

$
```

break, continue, AND cd STATEMENTS

- **break**
  - — USED TO BREAK OUT OF LOOPING CONSTRUCTS
  - — MAY BE SUPPLIED WITH AN ARGUMENT — NUMBER OF LOOPS FROM WHICH TO BREAK
- **continue**
  - — RETURNS TO BEGINNING OF LOOPING CONSTRUCT
  - — WILL NOT PROCESS REMAINDER OF LOOP
- **cd**
  - — CHANGES DIRECTORY TO SPECIFIED PATH NAME
  - — IF NO ARGUMENT, CHANGES TO *HOME* DIRECTORY
  - — IF **CDPATH** IS SET, SEARCHES FOR DIRECTORY PATH NAME

eval STATEMENT

The **eval** statement is used whenever a command line needs reevaluating. This may occur when a variable is part of that command line and may contain other variable substitution expressions or characters that have special meanings to the shell such as a semicolon (;) (for sequential execution) or a vertical bar (|) (for a pipeline). Parameter (or variable) substitution is only done once. The value of a parameter substitution is not scanned for other variable substitution expressions. Similarly, if a command substitution expression yields a command line or part of a command line, the command line must be reevaluated. This is done using the **eval** statement.

When the **eval** statement is used, the shell evaluates the command line as normal; then before executing the command, evaluates it once more.

eval STATEMENT — EXAMPLE

```
$ cat simulate<CR>
while
    echo ":${PS1}\c"
    read command_line
do
    ${command_line}
done

$ simulate<CR>

:$ echo ${HOME}<CR>
${HOME}

:$ cd ${HOME}<CR>
simulate: ${HOME}: bad directory

$ simulate    # program was ended by bad cd<CR>

:$ ls -p | pr -t5w80<CR>
| not found
pr not found
-t5w80 not found

:$
```

eval STATEMENT — EXAMPLE

```
$ cat simulate<CR>
while
    echo ":${PS1}\c"
    read command_line
do
    eval ${command_line}
done

$ simulate<CR>

:$ ls -p | pr -3tw48<CR>
calendar          test.file         unit.06
ckpath            time.words        unit.07
definitions       timeofday         unit.08
defs              title             unit.09

:$ echo ${HOME}<CR>
/ustg/hutch

:$
```

exec STATEMENT

The **exec** statement is similar to the system primitive (system call) **exec**. It replaces the current process image with the process image of the command to be executed. Therefore, if an **exec** statement were executed from the login shell level, it would replace the login shell. When the program being executed finishes, there is no shell to go back to, and a login prompt is issued.

```
$ cat laserprint<CR>
exec opr -bp165 -dpy2q -m -o0 -ttx ${*}

$ laserprint -jhutch unit.16<CR>

$
```

In this example, the **laserprint** command executes another command **opr** with several fixed options. It is not necessary for the shell process that invoked the **opr** command to remain during the execution of the **opr** command, so it is replaced (transformed) by that program **opr**. This reduces the total number of processes by one. Normally, three processes are needed:

- The invoking shell (interactive shell)
- The shell that was created to interpret the lines in *laserprint*
- The process to run the **opr** command.

When **exec** is used, there are only two processes used:

- The interactive shell
- The shell that interprets the command lines in the file *laserprint* and later executes the **opr** command.

exit AND export STATEMENTS

- **exit**
  - — TERMINATES SHELL PROCEDURE IMMEDIATELY
  - — ARGUMENT BETWEEN 0 AND 255
  - — ARGUMENT IS RETURNED TO CALLING SHELL AS EXIT STATUS
  - — NO ARGUMENT, STATUS OF LAST EXECUTED COMMAND
- **export**
  - — PUTS VARIABLES IN ENVIRONMENT
  - — MAKES VARIABLES AVAILABLE TO CHILD PROCESSES
  - — IF NO ARGUMENT, PRINTS VARIABLES EXPORTED BY THIS SHELL PROCESS

FUNCTIONS

Functions provide an efficient vehicle for executing relatively simple programs. Complex programs should be left as shell programs.

DEFINING A FUNCTION

The commands listed between the braces must follow specific syntax rules. There must be a space after the first brace and before the last brace. There must be a semicolon and space following each command. The semicolons and spaces are **not** necessary if the braces and the commands are typed on separate lines (which is preferred). Notice that there is nothing between the parentheses following the name of the function. The parentheses signify to the shell that this is a function definition (notation similar to that used in C). The function is then stored directly in memory (versus a UNIX System file).

FUNCTIONS

return STATEMENT

The use of the **return** statement is analogous to the use of the **exit** statement in a shell program. **return** is only recognized in a function definition. Use of **return** is optional. If left out, the function returns the exit code of the last command executed.

EXECUTING A FUNCTION

When executing the function, simply specify the function name followed by any arguments. Do not type the parentheses. The shell first checks if the command name entered is a function. If not, it uses **${PATH}** to find the command name.

REMOVING A FUNCTION

The function can be removed by using **unset** or by the termination of the shell. Functions can be defined in programs also. They are removed when the program ends. You may wish to code frequently used ones in your *.profile* file.

Functions are not known to child processes. An error message would be produced if a function name was an argument to export. The only way a child process can know about a function is if that function is defined in that process.

> **NOTE:** Whenever a function is invoked, the positional parameters for the **current** shell process are reset. The arguments to a function become the new positional parameters. If no arguments are supplied, the positional parameters are removed.

FUNCTIONS

- SIMILAR TO A SHELL PROGRAM EXCEPT IT IS STORED IN MEMORY
- EXECUTES FASTER THAN A PROGRAM BECAUSE IT IS IN MEMORY
- EXECUTES IN THE CURRENT SHELL PROCESS (NO CHILD)

FORMAT

```
name() { cmd1; cmd2; ...; return n; }
```

Or:

```
name()                          #  Preferred
{
    cmd1
    cmd2
    return n
}
```

- OBSERVE USE OF SPACING AND SEMICOLONS WITHIN THE COMMAND LIST

FUNCTIONS

- ARGUMENTS ARE ANALOGOUS TO POSITIONAL PARAMETERS AND SPECIFIED IN THE SAME FASHION (**$**{**1**}, **$**{**2**}, etc.)
- THE **return n** STATEMENT IS ANALOGOUS TO **exit**
- DEFAULT FOR **n** — EXIT STATUS OF THE LAST COMMAND EXECUTED
- NOT KNOWN TO CHILD PROCESSES — MAY NOT EXPORT
- **set** PRINTS A LIST OF ALL DEFINED FUNCTIONS — USE **unset** TO REMOVE

FUNCTIONS — EXAMPLE 1

```
$ whoon()<CR>  # Define a function
> {<CR>
> who | grep "${1}"<CR>
> }<CR>

$ set<CR>      # Display the function
HOME=/ustg/fred
MAIL=/usr/mail/fred
PATH=/bin:/usr/bin:/usr/lbin:/ustg/fred/bin:
TERM=5420
whoon() {
who | grep "${1}"
}

$ whoon fred<CR>    # Execute the function
fred     tty32      Mar 19 10:25

$ unset whoon<CR>   # Remove the function

$ set<CR>      # Show it is removed
HOME=/ustg/fred
MAIL=/usr/mail/fred
PATH=/bin:/usr/bin:/usr/lbin:/ustg/fred/bin:
TERM=c108
```

FUNCTIONS — EXAMPLE 2

```
prompt()
{
    while
        echo "Answer yes or no :\c"
        read yes_no
    do
        case "${yes_no}"
        in
            y* | Y*)
                yes_no="y"
                break
                ;;
            n* | N*)
                yes_no="n"
                break
                ;;
            *)
                echo "\n"
                continue
                ;;
        esac
    done
}
```

FUNCTIONS — EXAMPLE 2

```
echo "Would you like headings?"
prompt
if
    test "${yes_no}" = "y"
then
    echo headings selected
fi

echo lots of stuff done here.........

echo "Would you like a summary only?"
prompt
if
    test "${yes_no}" = "y"
then
    echo summary only
fi
```

HASHING

Hashing command names reduces the time necessary to find a command using the **PATH** variable. The hashing is performed automatically when you log in and cannot be turned off.

Basically, hashing allows a more efficient way to find command names rather than by linearly searching each directory listed in **PATH** until the name is found. This search is still done, but only **once** rather than each time the command is executed.

- COMMAND NAMES ARE HASHED BY THE SHELL
- REDUCES THE NUMBER OF DIRECTORY SEARCHES
- SHELL **-h** OPTION — LOCATE AND REMEMBER (**hash**) FUNCTION COMMANDS AS THEY ARE DEFINED
- CHILD PROCESS INHERITS PARENT'S **hash** TABLE, CHANGES ITS OWN COPY

hash STATEMENT

The **hash** command performs the following various functions:

a. The command with no arguments prints the **hash** table listing the path name of the command, the number of times it was executed, and the cost, which is actually the position the directory is within the **PATH** variable. Remember that the **PATH** variable is searched from left to right. The more directories that have to be searched, the more it costs in terms of time spent searching.

b. The **-r** option causes the table to be cleaned out, thus all remembered locations are forgotten. Any other arguments are ignored. This is a useful command for *.profile* — to clear names of any commands executed to set up the environment. This should also be used if **PATH** is rearranged.

c. By specifying names of commands as arguments, the names are set up in the **hash** table without actually executing the command.

The shell **-h** option causes functions to be hashed as they are defined. Normally functions are hashed upon execution. The option can be invoked by using the **set** command or the **sh** command. In terms of the environment, a child process inherits the parent's **hash** table and changes it locally. Upon return to the parent, the **hash** table is restored to the state it was before executing the child.

hash STATEMENT

- PRINTS THE FOLLOWING **hash** TABLE INFORMATION:
  - — HITS — NUMBER OF TIMES A COMMAND HAS BEEN INVOKED BY THE SHELL PROCESS
  - — COST — MEASURE OF WORK REQUIRED TO LOCATE (NOT EXECUTE) A COMMAND.

EXAMPLE

```
PATH=/bin:/usr/bin:/usr/lbin:${HOME}/bin:

Execute from /bin      - cost is 1
Execute from /usr/lbin - cost is 3
```

- — **hash -r** CLEARS THE **hash** TABLE
- — **hash name ...** CAUSES THE LOCATION OF **name** IN THE SEARCH PATH (**PATH**) TO BE REMEMBERED BY THE SHELL WITHOUT EXECUTING THE COMMAND

HASHING — EXAMPLE

1. `$ echo ${PATH}<CR>`
   `/bin:/usr/bin:/usr/lbin:/ustg/fred/bin:`

2. `$ hash date vi<CR>`

3. `$ hash<CR>`

hits	cost	command
0	1	/bin/date
0	2	/usr/bin/vi

4. `$ date<CR>`
   `Tue Mar 20 10:17:50 EST 1987`

5. `$ hash<CR>`

hits	cost	command
1	1	/bin/date
0	2	/usr/bin/vi

6. `$ date<CR>`
   `Tue Mar 20 10:18:18 EST 1987`

7. `$ hash<CR>`

hits	cost	command
2	1	/bin/date
0	2	/usr/bin/vi

8. `$ cat example<CR>`

```
date
ls | pr -5atw80
hash
```

HASHING — EXAMPLE

```
 9.  $ example<CR>
     Tue Mar 20 10:19:12 EST 1987
     bin        c.c        example        man
     hits     cost     command
     1        1        /bin/cat
     3        1        /bin/date
     0        2        /usr/bin/vi
     1        1        /bin/ls
     1        1        /bin/pr
     1*       5        example

10.  $ hash<CR>
     hits     cost     command
     1        1        /bin/cat
     2        1        /bin/date
     0        2        /usr/bin/vi
     1*       5        example

11.  $ hash -r<CR>

12.  $ hash<CR>
     hits     cost     command
```

newgrp STATEMENT

newgrp statement is used to change into a new group. This allows the user to access files using a new group identification, therefore, new permissions. The format for changing groups is as follows:

```
$ newgrp group_name<CR>
```

group_name is the name of the new group. The shell first checks the file */etc/group* to see if the user invoking the **newgrp** statement has permission to enter that group. If he or she does, then a new shell is **exec**ed using the new group identification.

On some systems, a password may be required to enter certain groups. The **id** command may be used to find out the user and group identification designations. The format for the **id**(1) command is as follows:

```
$ id<CR>
uid=26242(hutch) gid=46014(ustg)
```

This command gives both the numeric and alphabetic representations of both the user and group identifications.

> **Note:** The **newgrp** statement will always **exec** the file */bin/sh.* If another version of the shell is used, the default shell is invoked.

newgrp STATEMENT

- CHANGES GROUP IDENTIFICATION — ASSUMES NEW GROUP PERMISSIONS
- CHECKS FILE */etc/group* TO SEE IF USER IS ALLOWED IN GROUP
- **exec**s A NEW SHELL WITH THE NEW GROUP IDENTIFICATION
  - — NONEXPORTED VARIABLES GONE
  - — ALWAYS */bin/sh*
- PASSWORD MAY BE REQUIRED TO CHANGE GROUPS
- THE **id** COMMAND REPORTS **uid** AND **gid**

read AND readonly STATEMENTS

- **read**
  - — READS INPUT FROM STANDARD INPUT INTO VARIABLE(S)
  - — IF MORE THAN ONE ARGUMENT SUPPLIED TO READ, EACH VARIABLE GETS ONE WORD, AND WHAT IS LEFT OVER IS PUT IN THE LAST VARIABLE
  - — IF RECEIVES <control-d>, EXITS WITH A STATUS OF 1
- **readonly**
  - — WRITE-PROTECT VARIABLE
  - — IF SUPPLIED NO ARGUMENTS, PRINTS LIST OF **readonly** VARIABLES

type STATEMENT

There are four kinds of commands recognized by **type**:

- Shell built in
- functions
- Hashed
- Search of **${PATH}**.

In the case of the latter, the full path name of the command is listed. If the full path name of a hashed command is desired, you need to print the **hash** table.

type STATEMENT

- BUILT-IN STATEMENT
- INDICATES HOW NAMES WOULD BE INTERPRETED IF USED AS A COMMAND NAME

EXAMPLES

```
$ grep fredeng whofile<CR>
fredeng    tty43        Feb  9 15:09

$ type grep<CR>
grep is hashed (/bin/grep)

$ type ls<CR>
ls is /bin/ls

$ type cd<CR>
cd is shell builtin

$ whoson()<CR>
> {
>     who ¦ grep "${1}"
> }

$ type whoson<CR>
whoson is a function
whoson() {
   who ¦ grep "${1}"
}
$ type garbage<CR>
garbage not found
```

set STATEMENT

- USED TO RESET POSITIONAL PARAMETERS
- USED TO SET OPTIONS (TRACE, NONEXECUTABLE SHELL, ETC.)
- IF SUPPLIED WITH NO ARGUMENTS, PRINTS ALL **set** VARIABLES

```
$ set<CR>

$ set -v<CR>

$ set -- -v<CR>
```

unset AND wait STATEMENTS

- **unset**
  - — USAGE:

```
unset var1 var2 ...
```

  - — BUILT-IN STATEMENT
  - — REMOVES VARIABLES AND/OR FUNCTIONS FROM THE ENVIRONMENT
  - — VARIABLES CAN EITHER BE LOCAL OR EXPORTED
  - — UNSETTING AN EXPORTED VARIABLE IN A CHILD PROCESS DOES NOT UNSET IT IN THE PARENT
  - — NOT TO BE CONFUSED WITH SETTING A VARIABLE TO NULL
  - — CANNOT UNSET: **PATH**, **PS1**, **PS2**, **MAILCHECK**, OR **IFS**
- **wait**
  - — WAITS FOR ALL CHILDREN IF NO ARGUMENT SPECIFIED
  - — WAITS FOR SPECIFIED CHILD PROCESS
  - — ACCEPTS PROCESS IDENTIFICATION NUMBER AS ARGUMENT

times, trap, ulimit, AND umask STATEMENTS

- **times**
  - — PRINTS ACCUMULATED ELAPSED AND CPU TIME
- **trap**
  - — RUNS SPECIFIED COMMAND(S) WHEN A SIGNAL IS RECEIVED (SUCH AS GENERATED BY A **break**, **kill**, ETC.)
  - — WILL BE DISCUSSED IN DETAIL IN UNIT 18
- **ulimit**
  - — IF SUPPLIED WITH NO ARGUMENTS, PRINTS MAXIMUM FILE SIZE IN BLOCKS of 512 BYTES
  - — IF SUPPLIED WITH AN ARGUMENT, SETS THE FILE SIZE LIMIT TO THAT SIZE (IN BLOCKS)
  - — WILL BE IN EFFECT FOR CURRENT PROCESS AND ALL CHILD PROCESSES
- **umask**
  - — CHANGES DEFAULT FILE PERMISSIONS ON NEWLY CREATED FILES OR DIRECTORIES
  - — WITH NO ARGUMENTS, PRINTS CURRENT **umask** SETTING

shift AND test STATEMENTS

- **shift**
  - — SHIFTS POSITIONAL PARAMETERS TO THE LEFT BY AS MANY POSITIONS AS SPECIFIED BY THE ARGUMENT SUPPLIED
  - — IF NO ARGUMENT SUPPLIED, ASSUMES 1
  - — **${*}**, **${@}**, **${#}** ARE AUTOMATICALLY ADJUSTED
- **test**
  - — YIELDS A TRUE-FALSE VALUE
  - — ALLOWS FOR
    1. FILE STATUS TESTING
    2. STRING COMPARISON
    3. NUMERIC COMPARISON

VOLUME 13 EXERCISE

1. Write a function called **dir_lookup** that prints out a menu of all the directory names under the current directory. The function should then prompt for a number that corresponds to a directory. Read the user's response and then change to that directory. After changing to that directory, the function should do a '**ls -axF**'.

   EXAMPLE:

   ```
   $ pwd
   /class/chcec/cec21

   $ dir_lookup
   1       bin
   2       lib
   3       solutions

   Enter Selection: 2

   C       main    text

   $ pwd
   /class/chcec/cec21/lib

   $
   ```

2. Modify the function **dir_lookup** to maintain an environment variable called **DIRECTORY**. The function should add the name of each directory the user changes to the value of the variable and also add all the names in **DIRECTORY** to the menu listing.

VOLUME 13 EXERCISE

3. Write a shell procedure named **simulate** that will display the user's prompt (**PS1**) [preceded by a colon (:)] on the terminal, read a command line from the terminal, and then cause that command line to be executed.

   A. Set up a new environment variable named **MENU** that will be formatted like the **PATH** variable. It should contain the current directory and the directory *${HOME}/lib*.

      This variable assignment should be made at the login shell level and exported so the **simulate** program will know its value.

   B. Now, have the program determine whether or not the supplied name is a menu. Menus are found in **$HOME/lib.** If it is a menu, the file should be displayed on the terminal. If it is not a menu, it is a command line and should be executed.

      In determining whether the supplied name is a menu, the program should look for a nonexecutable file in each of the directories in the **MENU** variable. If the file is found in one of the directories listed in the **MENU** variable and is not executable, it is a menu and should be printed instead of executed.

      If the **IFS** variable was reset by the shell procedure, make sure it is reset to its original condition before executing any other shell commands.

   C. Make sure you test this program using simple commands, commands with variable and command substitution expressions, sequential command lines, and pipelines. The command should work with any of these.

   D. Have this program continue prompting until the command **done** is received.

VOLUME 13 EXERCISE

E. Modify the program so that when a null command line is entered, a default menu will print. The value of the default should either be taken from the new environment variable **DEFAULT** or given a default value of **main**. The value of **DEFAULT** should be set at the login shell level and exported.

F. Modify the program to allow for any printing command to be used instead of the **cat** command when displaying menus. This should be the value in the variable **SHOW** that would be defaulted to if not otherwise set in the command **cat**.

VOLUME 13 EXERCISE - ANSWERS

1. **dir_lookup()**

```
{
   i=0
   > /tmp/dir.lookup${$}
   for dir
   in * # all files in current directory
   do
      if
         test -d "${dir}"
      then
         i=`expr ${i} + 1`
         echo "\n${i}  ${dir}" >> /tmp/dir.lookup${$}
      fi
   done
   if
      test "${i}" -eq 0
   then
      echo There are no directories under the current directory.
      exit 1
   fi
   cat /tmp/dir.lookup${$}
   echo "\nEnter number from 1 to ${i}: \c"
   read selection
   echo
   dirname=`grep "${selection}" /tmp/dir.lookup${$} | cut -f2`
   cd ${dirname}
   echo "ls -axF ${dirname}"
   ls -axF
}
```

VOLUME 13 EXERCISE - ANSWERS

2. **dir_lookup()**

```
{
   i=0
   > /tmp/dir.lookup${$}
   for dir
   in *
   do
      if
         test -d "${dir}"
      then
         i=`expr ${i} + 1`
         echo "\n${i} \t `pwd`/${dir}" >> /tmp/dir.lookup${$}
      fi
   done
   OLDIFS="${IFS}"
   IFS="${IFS}:"
   for dir
   in ${DIRECTORY}
   do
      i=`expr ${i} + 1`
      echo "\n${i}     ${dir}" >> /tmp/dir.lookup${$}
   done
   IFS="${OLDIFS}"
   if
      test "${i}" -eq 0
   then
      echo There are no directories under the current directory.
      exit 1
```

```
    fi
    cat /tmp/dir.lookup${$}
    echo "\nEnter number from 1 to ${i}: \c"
    read selection
    echo
    if
       test "${selection}" -lt 1 -o "${selection}" -gt "${i}"
    then
       echo Invalid selection
       exit 2
    fi
    dirname=`grep "${selection}" /tmp/dir.lookup${$} | cut -f2`
    DIRECTORY="${DIRECTORY}:${dirname}"
    cd ${dirname}
    echo "ls -axF ${dirname}"
    ls -axF
}
```

VOLUME 13 EXERCISE - ANSWERS

3. **$ cat simulate<CR>**

```
OLDIFS="${IFS}"
while
   echo ":${PS1}\c"
   read selection
do
   if
      test "${selection}" = ""
   then
      selection=${DEFAULT:-main}
   fi
   IFS="${IFS}:"
   for directory
   in ${MENU}
   do
      IFS="${OLDIFS}"
      if
         test -f ${directory}/"${selection}" -a ! -x ${directory}/"${selection}"
      then
         ${SHOW:-cat} ${directory}/"${selection}"
         continue 2
      fi
   done
   eval ${selection}
done
```

VOLUME 13 SUMMARY

Statement	Meaning	Example
	Used to execute shell procedures. Will not create child process to interpret command lines of file	$ . .profile <RET>
cd	Change directory to specified path name	$ cd /instr/kal <RET>
exec	Replaces current process image with process image of the command to be executed. Similar to exec system call	$ exec $FILESYS/tsys <RET>
export	Puts variable in environment	$ export $PATH <RET>
hash	Prints hash table listings	$ hash <RET>
newgrp	Change into a new group	$ newgrp uucp <RET>
read into variables	Reads input from standard input	$ read var1 <RET>
readonly	Write-Protect variable	$ readonly var1 <RET>
type	Indicates how names would be interpreted if used as a command name	$ type echo who <RET>
unset	Removes variables and/or functions from the environment	$ unset var1 <RET>
times	Prints accumulated elapsed and CPU time	$ times <RET>
umask	Changes default file permissions on newly created file or directories	$ umask 023 <RET>

Statement	Meaning
break	Used in looping constructs to exit out of a specified loop
continue	Will return to the next iteration of a looping construct
eval	Used to reevaluate a command line
exit	Terminates shell procedure immediately
trap	Runs specified commands when a signal is received
shift	Shifts positional parameters to the left by as many positions as specified by the argument supplied

VOLUME 14

Redirection

WHAT YOU'LL BE ABLE TO DO AFTER VOLUME 14

- Use redirection to alter the source or the destination of a command
- Write error messages to the standard error output
- Access other files from within a shell program
- Describe the actions and the purpose of the file descriptor table
- Open files using the **exec** statement
- Redirect multiple sources to a single destination.

REDIRECTION — REVIEW

- Standard input redirection

```
$ command < file<CR>
```

This causes the command **command** to take its input from the file named *file* rather than from the terminal. The file name may be the result of a parameter or command substitution expression, but must not contain any file name generation characters.

- Standard output redirection

```
$ command > file<CR>
$ command >> file<CR>
```

There are two methods for redirecting the standard output. The first example replaces the file *file* already existing. The second example only appends (adds to) the end of the file. In either case, if the file does not exist, it is created in the current directory.

When redirecting output, as with all redirection, file permissions are checked first. If a file is being created in a directory, you must have write (and execute) permissions in that directory. If input is being redirected, then you must have read permission on that file.

- Standard error redirection

```
$ command 2> file<CR>
$ command 2>> file<CR>
```

This prints all messages that were written to the standard error in the command **command** to the file named *file*. The difference between the two examples is if the file existed, then the first example would replace that file; the second would add to the end of the file.

WRITING TO STANDARD ERROR

The output of any command is redirected to the standard error using the special symbols **>&2**. This redirects the output of the preceding command and sends it to the standard error (file descriptor 2).

FORMAT

The format for using the **echo** command to print a simple message to the standard error output is as follows:

```
$ cat myscript<CR>
echo "Usage: ${0} pattern file_name .. file_name" >&2
              .
              .
              .
```

This prints the message

```
Usage: mycommand pattern file_name .. file_name
```

to the standard error output (assuming **mycommand** is the name of the command file). Normally, the standard error prints on the terminal; but if it were redirected, it (alone) prints to the redirection object. This method of writing messages also has a benefit over writing all messages to the same place because the error messages do not clutter up the redirected output of the command. They appear on the screen even if the output of the command is redirected.

WRITING TO STANDARD ERROR

The output of a command, other than the **echo** command, may be sent to the standard error output, such as the **cat** command.

```
$ cat >&2 <<EOF<CR>
The ${0} command  - options
     -h          full help message
     -p          specify directory
     -s          silent (no error messages)
EOF
```

There are a few things going on in this example. The output of the **cat** command is sent to the standard error and the input of the **cat** command is taken from the text that appears between the two instances of the word EOF. This example prints the message

```
The mycommand command  - options
     -h          full help message
     -p          specify directory
     -s          silent (no error messages)
```

assuming that the name of the command is **mycommand** to the standard error. This information prints to your terminal unless the standard error is redirected to a file.

In this example, the redirection should appear on the same line as the command name. If it appeared on the last line, the shell would not recognize the EOF delimiter.

WRITING TO STANDARD ERROR

FORMAT

- SIMPLE MESSAGE

```
$ cat myscript<CR>
echo "Usage: ${0} pattern file_name .. file_name" >&2
                .
                .
                .
```

- **&** IS USED TO DENOTE FILE DESCRIPTOR NUMBER (OF STANDARD ERROR)
- PRINTS MESSAGE TO THAT FILE DESCRIPTOR (FILE ASSOCIATED WITH THE FILE DESCRIPTOR)
- COMMAND OUTPUT REDIRECTION

```
$ cat >&2 <<EOF<CR>
The ${0} command  - options
    -h    full help message
    -p    specify directory
    -s    no error messages
EOF
```

- PRINTS ON TERMINAL UNLESS THE ERROR OUTPUT OF THE SHELL PROCEDURE IS REDIRECTED

ACCESSING OTHER FILES — EXAMPLE

```
$ cat file.des<CR>
echo "Start of Menu"
cat ${HOME}/menus/main >&4
echo "End of Menu"

$ file.des 4> temp<CR>
Start of Menu
End of Menu

$ cat temp<CR>
               UNIX SYSTEM V

         cat - print a file
        grep - search for pattern
programming - programming languages

$
```

REDIRECTION

FORMAT

```
[d]operator to
```

WHERE

- [**d**] IS A FILE DESCRIPTOR NUMBER
- DEFAULTS FOR [**d**]
  - — > DEFAULT IS 1 (STANDARD OUTPUT)
  - — >> DEFAULT IS 1 (STANDARD OUTPUT)
  - — < DEFAULT IS 0 (STANDARD INPUT)
- **operator** IS EITHER >, <, OR >>
- **to** IS EITHER A FILE PATH NAME OR AN EXPRESSION OF THE FORM

  ```
  &n
  ```

  WHERE **n** IS A FILE DESCRIPTOR

READING FROM FILES — EXAMPLE

```
$ cat maintain<CR>
while
     file=`line < &3`
     test "${file}" != ""
do
     cat <<-EOF
      MENU SELECTIONS

      1    EDIT FILE
      2    DELETE FILE
      3    TRUNCATE FILE

      EOF
      echo "\nEnter selection for file ${file}: \c"
      read option
      case "${option}"
      in
          1) vi ${file} ;;
          2) rm ${file} ;;
          3) > ${file}  ;;
      esac
done

$ cat update.files<CR>
/usr/lib/error.log
/lib/trouble.log
/etc/motd
```

READING FROM FILES — EXAMPLE

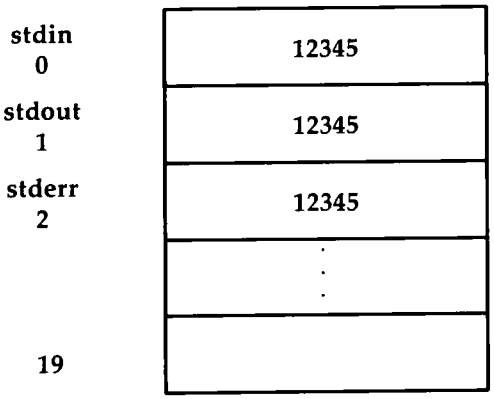
```
$ maintain 3<update.files<CR>
      MENU SELECTIONS

      1      EDIT FILE
      2      DELETE FILE
      3      TRUNCATE FILE

Enter selection for file /usr/lib/error.log: 3

      MENU SELECTIONS

      1      EDIT FILE
      2      DELETE FILE
      3      TRUNCATE FILE

Enter selection for file /lib/trouble.log: 3

      MENU SELECTIONS

      1      EDIT FILE
      2      DELETE FILE
      3      TRUNCATE FILE

Enter selection for file /etc/motd: 1

(vi /etc/motd would now be executed)
$
```

REDIRECTION — EXAMPLE

COMMAND:

```
$ echo "This is an error message" >&2<CR>
```

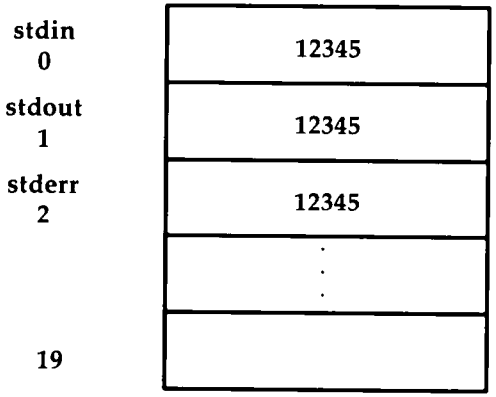
FILE DESCRIPTOR TABLE (BEFORE REDIRECTION)



(Continued)

REDIRECTION — EXAMPLE

FILE DESCRIPTOR TABLE (AFTER REDIRECTION)



WRITING TO FILES

- OUTPUT OF WRITING COMMAND SHOULD BE REDIRECTED TO A FILE DESCRIPTOR
- THE FILE DESCRIPTOR SHOULD BE OPEN BY REDIRECTION ON THE COMMAND LINE OR BY **exec** IN THE PROGRAM

OPENING FILES

The **exec** statement is used to open files from within a shell program. The following is an example of the method used to open file descriptor 2 for permanent output to the file named *file_name*:

```
exec 2> file_name
```

This means from this point on, when any messages are written to the standard error, they are added to the file *file_name*.

The command line

```
exec 3> ${1}<CR>
```

in a shell procedure "opens" the file as supplied by the first positional parameter for output. Whenever a command's output is redirected from within that shell program to file descriptor 3, that output is added to the file specified in the first positional parameter.

For example, if the command line in a shell procedure

```
echo message >&3<CR>
```

is issued following the **exec** statement, the message **message** is written to the file specified by the first positional parameter.

OPENING FILES

- THE **exec** STATEMENT
  - — MAY BE USED TO OPEN FILES FROM WITHIN A SHELL PROCEDURE

EXAMPLE

```
exec 2> file_name
```

- THIS WOULD OPEN THE FILE *file_name* TO REPLACE THE TERMINAL AS THE DESTINATION FOR THE STANDARD ERROR

- THE COMMAND LINE

```
exec 3> ${1}<CR>
```

  IN A SHELL PROCEDURE "OPENS" THE FILE AS SUPPLIED BY THE FIRST POSITIONAL PARAMETER FOR OUTPUT

- ACCESSIBLE THROUGH FILE DESCRIPTOR NUMBER 3

```
echo message >&3<CR>
```

exec STATEMENT — EXAMPLE 1

The following program opens either the value of the first positional parameter or, if no value is given, the file *log* for output using file descriptor 3. From that point on, when information is redirected to file descriptor 3, the information is written to the file specified.

After writing to file descriptor 3, the file *completion* is opened for output using the **exec** statement and the message "End of log information" is written to file descriptor 4, thereby going to the file *completion*.

```
$ cat change<CR>
exec 3> ${1:-log}
echo "Name: ${LOGNAME}" >&3
echo "TERM: ${TERM}" >&3

exec 4> completion
echo "Log Successful" >&4
date >&4

echo "End of log information" >&3

$ change<CR>

$ cat log<CR>
Name: hutch
TERM: 2621
End of log information

$ cat completion<CR>
Log Successful
Wed Apr 20 14:45:47 EST 1987

$
```

When the program is run, no output is generated to the terminal, but rather to the files *log* and *completion*.

exec STATEMENT — EXAMPLE 2

This example takes three arguments, the first two being files containing information to be merged into a single file. The third argument names the file to hold the merged information. This program opens the files using the **exec** statement.

Notice how one can put multiple commands in the testing part of a **while** statement and even have the body of the loop be a **null** statement.

```
$ nl merge<CR>
  1  if
  2     test  -z "$1"  -o  -z "$2"  -o  -z "$3"
  3  then
  3     echo "Usage: $0 infile1 infile2 outfile" >&2
  5     exit 255
  6  fi
  7  # change I/O connections for current shell process
  8  exec  3<"$1"  4<"$2"  5>"$3"
  9  while
 10     input=`line <&3`       # read line from &3
 11     eof_3=$?               # save error code; 0 means not EOF
 12     # don't write any output unless something was read
 13     if       test "$eof_3" -eq 0
 14     then     echo "$input" >&5
 15     fi
 16     input=`line <&4`       # read line from &4
 17     eof_4=$?               # save error code; 0 means not EOF
 18     # don't write any output unless something was read
 19     if       test "$eof_4" -eq 0
 20     then     echo "$input" >&5
 21     fi
 22     # continue while >=1 file still supplying input
 23     test "$eof_3" -eq 0 -o "$eof_4" -eq 0
 24  do
 25     :                 # null command
 26  done
```

exec STATEMENT — EXAMPLE 2

When one uses the **merge** command with an improper amount of arguments, an error message is displayed. When invoked with the proper amount of arguments (3), it takes a line in turn from each of the first two files and stores all lines in the third file.

```
$ cat lower<CR>
    1   a
    2   b

$ cat upper<CR>
    1   A
    2   B
    3   C
    4   D

$ merge<CR>
Usage: merge infile1 infile2 outfile

$ merge  lower  upper  mixed<CR>

$ cat mixed<CR>
a
A
b
B
C
D
$
```

VOLUME 14 EXERCISE

1. Write a shell program named **copy** that takes two arguments. The first argument is a regular file that exists, and the second argument is a filename. **copy** is going to read ${1} using file descriptor 3 and write to ${2} using file descriptor 4. Read one line at a time from ${1} and write it to ${2}. Be sure to write all error messages to standard error.

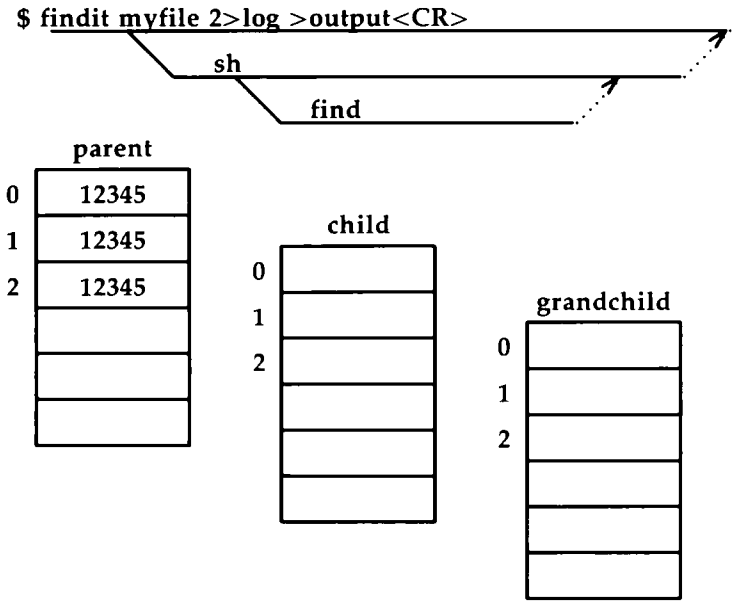
2. Assuming that the program **findit** contains the following command lines

```
$ cat findit<CR>
find . -name ${1} -print >&2
```

fill in the file descriptor table for the child and grandchild processes.

When the file *log* was opened for output, the information about the opening of the file was recorded at position 22000. When the file *output* was opened, the information about the opening was recorded at location 45000 (over).

VOLUME 14 EXERCISE



File pointer from opening log is 22000.

File pointer from opening output is 45000.

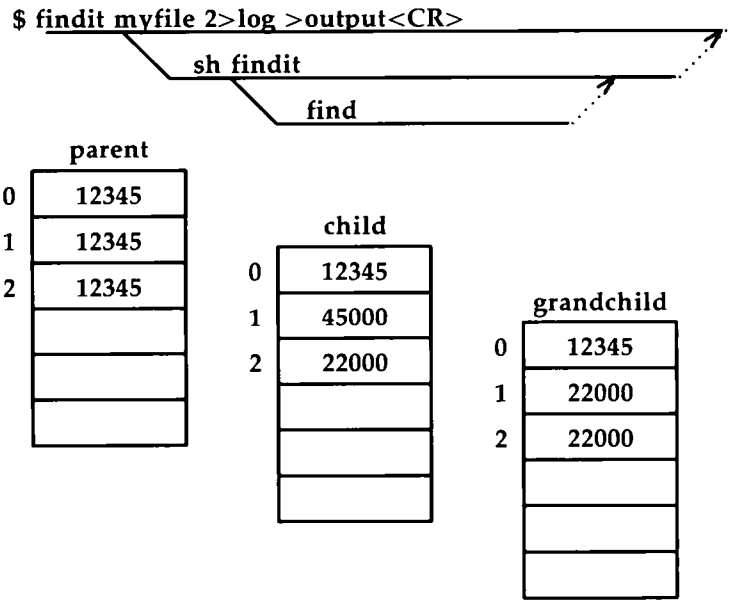
VOLUME 14 EXERCISE - ANSWERS

1. **$ cat copy<CR>**

```
# copy - copy argument one to argument two
# two arguments: 1 - original file
#                2 - copy
# exit status    0 - normal termination
#                1 - not enough arguments
if
    test ${#} -lt 2
then
    echo Usage: ${0} original destination >&2
    exit 1
fi
exec 3< ${1}
exec 4> ${2}
while
    input="`line <&3`"
    test "${input}" != ""
do
    echo "${input}" >&4
done
```

VOLUME 14 EXERCISE - ANSWERS

2. $ findit myfile 2>log >output<CR>



File pointer from opening log is 22000.

File pointer from opening output is 45000.

VOLUME 14 SUMMARY

Statement	Meaning	Example
exec	May be used to open files from within a shell procedure	exec 2> file

Redirection	Command
Standard Input	$ command < file <RET>
Standard Output	$ command > file <RET> $ command >> file <RET>
Standard Error	$ command 2> file <RET> $ command 2>> file <RET>

VOLUME 15

Pipelines and Signal Processing

WHAT YOU'LL BE ABLE TO DO AFTER VOLUME 15

- Use the pipe operator and understand how the pipeline is implemented in the UNIX system
- Define the exit status of a pipeline
- Use looping constructs as filters
- Use conditional statements as filters
- Use common groups as filters
- Recognize the trappable signals and what causes them
- Use the **trap** statement to catch signals and perform commands when the signal is received
- Use the **trap** statement to reset trap settings
- Use the **trap** statement to inspect trap settings
- Understand the difference between ignoring a signal and doing nothing when a signal is received
- Use nested **trap** sequences to protect trap routines from further incoming signals
- Use quotes properly when setting traps.

PIPELINES — DEFINITION

- A SEQUENCE OF COMMANDS SEPARATED BY THE PIPE OPERATOR
- PIPE FILE
- REDIRECTION

pipe OPERATOR

The pipe operator is a special character that is detected and processed by the shell. The traditional character | is used in the formation of a pipeline, but the character ^ may also be used. The caret (^) is allowed in creating pipelines, but not recommended. It was the method of creating pipelines in early versions and is only included as part of the language for compatibility reasons.

When the shell detects a pipe operator, it creates a UNIX system pipe file. This is a FIFO (first-in first-out) data structure that is read from and written to at the same time.

The command that appears to the left of the pipe operator has its output redirected to the write end of the pipe file. The command to the right of the pipe operator has its input redirected from the read end of the pipe. It is important to only use commands that write to the standard output to the left of a pipe operator and commands that take their input from the standard input to the right of a pipe operator. The same routines are used by the shell in processing this redirection. Redirection is done as if output were being redirected to a regular file.

pipe OPERATOR

- RECOGNIZED BY THE SHELL AS A SPECIAL CHARACTER
- EITHER THE CHARACTER "|" OR THE CHARACTER "^"
- CREATES *UNIX* SYSTEM PIPE FILE (FIFO)
- REDIRECTS OUTPUT OF THE COMMAND ON THE LEFT TO WRITE END OF PIPE
- REDIRECTS INPUT OF THE COMMAND ON THE RIGHT TO READ END OF PIPE

UNIX SYSTEM pipe FILE

The UNIX system pipe file is a FIFO data structure that uses 10 blocks over and over. The size of the block is determined by the hardware being used and is either 512 or 1024 bytes in length. Many times, depending on the size of the system buffers, the file remains in memory and never has to be written to a disk. If there is not sufficient memory, then the file is written to a disk file, but that is an infrequent event.

There are two pointers (file pointers) generated when a pipe file is opened. One points to the read end of the pipe and another points to the write end of the pipe. The file pointer that points to the read end of the pipe replaces the standard input file descriptor table entry for standard input of the command to the right of the pipe operator. A similar action takes place to the standard output file descriptor table entry of the command to the left of the pipe, but the write end of the pipe is used.

USING CONTROL CONSTRUCTS AS FILTERS

Control constructs may be used as sources, filters, or sinks if they meet the following requirements:

- Source must generate output to the standard output.
- Filter must generate output to the standard output and accept input from the standard input.
- Sink must accept input from the standard input.

In looping constructs, a common way to accept input is through the **read** statement. It takes a line of input and stores it in the supplied variable. If used in a **while** statement, it takes a line of input from the pipe and stores it. If more than one variable name is supplied to the **read** statement, it puts the first word to come from the pipe into the first variable, the second into the second supplied variable, and so on.

When used in a **while** statement, the **read** statement also exits with a nonzero status when the preceding command is finished and the pipe supplies an EOF.

Conditional constructs, since they are only executed once, must process all the information sent through the pipe. Conditional constructs are used when an external condition determines the path the data should flow.

Whenever a control construct is piped to or from, or if the input or output of a control construct is redirected, then a separate process is created to run that control construct. This is necessary because the file descriptor tables of the two processes are altered by the redirection.

USING CONTROL CONSTRUCTS AS FILTERS

- MUST ACCEPT INPUT FROM STANDARD INPUT
  - — LOOPING CONSTRUCTS MAY ACCEPT ONE INPUT LINE AT A TIME, USING THE **read** STATEMENT
  - — CONDITIONAL CONSTRUCTS MUST PROCESS ENTIRE OUTPUT IN ONE PASS
- MUST GENERATE OUTPUT TO STANDARD OUTPUT
- CREATES A SEPARATE PROCESS TO RUN THE CONTROL CONSTRUCT

PIPING TO A while LOOP — EXAMPLE

This example prints out the specified file and attaches line numbers to the beginning. It takes the output of the **cat** command and pipes it to a **while** loop. The **while** loop reads the output of the **cat** command one line at a time and prints that line along with its line number. The **expr** command is used to increment the line counter.

```
$ cat line.number<CR>
line_number=1
cat ${1} |
while
    read name
do
    echo "${line_number}:    ${name}"
    line_number=`expr ${line_number} + 1`
done

$ cat test.file<CR>
This is line 1
This is line 2
This is line 3

$ line.number test.file<CR>
1:  This is line 1
2:  This is line 2
3:  This is line 3

$
```

Note: This program removes any preceding blanks or tabs in the line that was read. This is due to the nature of the **read** statement. The **read** statement only stores into the specified variable characters, starting at the first non-**IFS** character.

PIPING TO CONDITIONAL STATEMENTS EXAMPLE

This example takes the output of the **cut** command and takes one of several paths, depending on the value of the **TERM** variable. If **TERM** is set to 2621, hp, 2645, 2626, or vt100, the information from the pipeline is sent to the **pr** command with the options **-5tw80**. Depending on the width of the terminal being printed to, different options are given to the **pr** command.

```
$ cat cond<CR>
who | cut -c1-8 |
case "${TERM}"
in
    2621 | hp | 2645 | 2626 | vt100)
        pr -5tw80
        ;;
    adm3a | 745 | ti)
        pr -4tw72
        ;;
    450 | 350)
        pr -8tw132
        ;;
    blit | jerq)
        pr -6tw86
        ;;
esac
```

PIPING TO CONDITIONAL STATEMENTS EXAMPLE

In the first example, the variable **TERM** is set as a keyword parameter to 745 and the output appears in four columns only using 72 character positions (output is simulated). When the value of **TERM** is set to 2621, the output is printed in five columns using 80 character positions.

```
$ TERM=745 cond<CR>
ds52        yanick      sbf         delores
tbr         hutch       clg         cynthia
brt         rjo

$

$ TERM=2621 cond<CR>
ds52        brt         hutch       sbf         delores
tbr         yanick      rjo         clg         cynthia

$
```

Question: What would the output look like if the variable **TERM** had no value (or not one of the expected terminal types)?

Answer: The program is executed up to the case statement. No output is displayed on the screen.

PIPING TO/FROM CONSTRUCTS — EXAMPLE

This example shows a control construct used as a filter. It is taking its input by using the **read** statement and writing output to the standard output through the **echo** command.

```
$ cat loop<CR>
cd ${1:-.}
ls |
while
    read name
do
    if
        test -d "${name}"
    then
        echo "[${name}]"
    elif
        test -x "${name}"
    then
        echo "*${name}*"
    else
        echo "${name}"
    fi
done | pr -3tw48
```

The program first changes directories to the specified directory, or if no name is specified, it remains where it is. Then it lists out that directory and tests the files contained therein. If the file is a directory, the name appears between [brackets]. If the file is not a directory, but executable, the program prints the name surrounded with asterisks; otherwise, the name appears normally. The output of the program is displayed in three columns, using 48 character positions.

PIPING TO/FROM CONSTRUCTS EXAMPLE

```
$ loop<CR>
FIFO            title.18      unit.10
FIFO0           title.3       unit.11
[backup]        title.4       unit.12
*ckpath*        *trep*        unit.13
*edit.files*    unit.01       unit.14
*file.des*      unit.02       unit.15
```

PIPING WITH GROUPS

Groups, like other control constructs, may be piped to or from. The input of a group is treated as one. So when output is redirected to a command group, the commands within the group share the input. If the group appears to the left of a pipe operator, the output of the group is treated as one command.

EXAMPLE 1

```
$ ( cat file.1 ; grep pattern file.2 ) | sort<CR>
```

Example 1 sequentially merges the output of the **cat** and **grep** commands. The output of the **cat** command first goes into the pipe; and when the **cat** command has finished, the output of the **grep** command follows it.

Example 2 shows where a command group is being used as a filter:

EXAMPLE 2

```
$ ps<CR>
   PID  TTY  TIME COMMAND
  6149   27  0:04 sh
 11359   27  0:06 ps

$ ps | ( line >/dev/null ; cat ) | sort<CR>
  6149   27  0:04 sh
 11392   27  0:06 ps
 11393   27  0:00 sort
 11394   27  0:00 sh
 11395   27  0:00 line
```

The **line** command takes the first line of the input of the **ps** command and throws it away into the file */dev/null*. Since the **line** command only reads one line of input and then ends, the **cat** command receives the rest and simply passes it on to the **sort** command. This process effectively removes the heading line from the output and sorts the rest of the output of the **ps** command.

PIPING WITH GROUPS

EXAMPLE 1 (TREATED AS ONE COMMAND)

```
$ ( cat file.1 ; grep pattern file.2 ) ¦ sort<CR>
```

- COMMAND GROUPS ALLOW OUTPUT TO BE TREATED AS THAT FROM ONE COMMAND
- SEQUENTIALLY MERGES THE OUTPUT OF ALL COMMANDS IN GROUP AND FEEDS IT TO THE **sort** COMMAND

EXAMPLE 2 (GROUP COMMAND USED AS A FILTER)

```
$ ps<CR>
   PID  TTY      TIME COMMAND
  6149  tty27   0:04 sh
 11359  tty27   0:06 ps

$ ps ¦ ( line >/dev/null ; cat ) ¦ sort<CR>
  6149  tty27   0:04 sh
 11392  tty27   0:06 ps
 11393  tty27   0:00 sort
 11394  tty27   0:00 sh
 11395  tty27   0:00 line
```

SIGNALS

Signals are events external to a program that might affect its execution. These events may be initiated from the user's terminal, from the operating system, or from another running command. These signals terminate the running program if they are not trapped. This act of trapping signals is performed through the use of the **trap** statement. By using the **trap** statement, the programmer chooses to either ignore the signal, perform a command or commands when the signal is received, or reset a previously set trap.

This is a list of trappable signals in the shell. Only signals 0, 1, 2, 3, 15, 16, and 17 should be trapped in a shell program. If signal 0 is trapped, then the trap routine runs when the program completes; however, it may complete. This is often used to clean up temporary files that might have been created by the shell program.

00 Normal program termination.

01 Hangup (line disconnect).

02 Interrupt (break).

03 Quit (FS character). This signal is similar to the break, except that it also results in a dump of core in the current directory (assuming that there is write permission in that directory).

04 Illegal instruction.

05 Trace trap (traced program tries to 'exec').

06 IOT Instruction.

07 EMT instruction.

08 Floating point exception.

10 Bus error.

11 Segmentation violation (accessing an address out of the program address space).

12 Bad argument to a system call.

13 Write on a **pipe** with no one to read on it.

14 Alarm clock.

15 Kill signal. This is the default signal produced by the **kill** command.

16 User-defined signal number 1. This and the signal may be used by programs to send signals to each other.

17 User-defined signal number 2.

Noncatchable Signals

The following is a list of signals that should never be trapped. Signal 09 may not be trapped; the other two 18 and 19 **should** not be trapped.

09 Noncatchable kill.

18 Death of a child should not be trapped by a shell program. It causes problems with pipelines.

19 Power failure. Leave this one for the operating system.

SIGNALS

- EXTERNAL CONDITIONS THAT MAY AFFECT THE EXECUTION OF A RUNNING PROGRAM
- MAY COME FROM
  - — THE USER'S TERMINAL (BREAK)
  - — THE OPERATING SYSTEM
  - — OTHER SOFTWARE (VIA THE **kill** COMMAND)
- NORMALLY TERMINATES PROCESS UNLESS "TRAPPED"
- SIGNALS MAY BE
  - — CAUGHT
  - — IGNORED
  - — RESET

SIGNALS

00 NORMAL PROGRAM TERMINATION

01 HANGUP

02 INTERRUPT (BREAK)

03 QUIT (FS CHARACTER)

04 ILLEGAL INSTRUCTION

05 TRACE TRAP (TRACED PROGRAM TRIES TO 'exec')

06 IOT INSTRUCTION

07 EMT INSTRUCTION

08 FLOATING POINT EXCEPTION

10 BUS ERROR

11 SEGMENTATION VIOLATION (ACCESSING AN ADDRESS OUT OF THE ADDRESS SPACE)

12 BAD ARGUMENT TO A SYSTEM CALL

13 WRITE ON A **PIPE** WITH NO ONE TO READ FROM IT

14 ALARM CLOCK

15 **kill** SIGNAL

16 USER-DEFINED SIGNAL NUMBER 1

17 USER-DEFINED SIGNAL NUMBER 2

SIGNALS

NONCATCHABLE SIGNALS

09 NONCATCHABLE KILL

18 DEATH OF A CHILD SHOULD NOT BE TRAPPED BY A SHELL PROGRAM — WILL CAUSE PROBLEMS WITH PIPELINES

19 POWER FAILURE

trap STATEMENT

FORMAT

The **trap** statement is used to trap signals and perform functions when the signal is received.

FORMAT

```
trap "command" signal signal
```

"**command**" is the command that is executed whenever one of the specified signals is received. As soon as the command is executed, then the program resumes at the point where it was interrupted. If the program ends at this point, use the **exit** statement as part of the command sequence. Often, the command specified is not only a single command, but a sequence of commands separated by semicolons or enclosed in quotes.

EXAMPLE

```
trap 'echo "Interrupted routine" ;
            rm tmp${$} ; exit 4'     1 2 3
```

This sequence causes, if signal 1, 2, or 3 is received, the message "Interrupted routine" to print, the file *tmp${$}* to be removed, and the program to exit with a status of four (4).

It is possible to have signals ignored. The following two ways can cause this to happen. These command lines normally appear at the beginning of the shell procedure:

```
trap '' 2 3

     or

trap ':' 2 3
```

There is a small difference between the two. The first truly ignores the specified signals, and the second does nothing when the signal is received. This may not seem important, but it has a significant difference as explained on the next page.

If the **trap** statement is issued with no arguments, it prints the trap settings for all trapped signals.

trap STATEMENT

FORMAT

```
trap ["commands"]  [signal...]
```

EXAMPLE

```
trap 'echo "Interrupted routine" ;
            rm tmp${$} ; exit 4'     1 2 3
```

- IF SPECIFIED SIGNALS ARE RECEIVED
  - — SPECIFIED COMMAND EXECUTES
  - — CONTINUES AT LINE AFTER ONE EXECUTING WHEN SIGNAL WAS RECEIVED UNLESS THE COMMAND CONTAINED AN **exit** STATEMENT
- IF EXPLICITLY NULL OR NO COMMAND SPECIFIED, THEN

```
trap '' 2 3          # IGNORE

   OR

trap ':' 2 3         # DO NOTHING
```

- TRAP WITHOUT ANY ARGUMENTS LISTS CURRENT TRAP SETTINGS

IGNORING VS. THE NULL STATEMENT

The difference between ignoring a signal and doing nothing when the signal is received is only significant in processes that are children to the process in which the trap is set.

If the signal is ignored in the parent, then it will be ignored in the child as well. If a signal is trapped in a parent, the trap settings will be reset in child processes that execute compiled programs, such as the **echo** command, the editor, etc. If the signal is to be trapped and nothing is to be done, then a trap has been set and will be reset in a child process. If it does not ignore the signal, it will do nothing when it is received.

It is possible to reset trap settings on signals by using the **trap** statement without a command.

IGNORING VS. THE NULL STATEMENT

- IN CHILD PROCESSES, IGNORED SIGNALS ARE IGNORED
- IN CHILD PROCESSES, ALL TRAP SETTINGS ARE RESET
- THE NULL STATEMENT IS A TRAP SETTING — THE **do-nothing** STATEMENT IS RESET IN CHILD PROCESSES
- EXAMPLE

```
trap ""  2     #IGNORE INTR AND TELL CHILDREN
                TO DO SAME

trap ":" 2     #IGNORE INTR BUT DO NOT MODIFY
                CHILDREN'S RESPONSE TO SIGNAL
```

RESETTING TRAPS

If no command is specified, but signals are specified, then any trap settings set for those signals are removed. This means from that point on if that signal is received, the program will terminate.

The format for resetting traps is as follows:

```
trap 1 2 3
```

This command line removes any trap settings for the signals 1, 2, and 3. If any of those signals are received, then the program terminates.

RESETTING TRAPS

- IF NO COMMAND SPECIFIED, WILL RESET SETTINGS FOR SPECIFIED SIGNALS
- THE COMMAND LINE

  trap 1 2 3

  REMOVES PREVIOUS TRAP SETTINGS FOR THESE SIGNALS

EXAMPLE USING TRAP

```
$ cat mybc<CR>
trap "" 2
trap '
        rm /tmp/bc${$}
        exit 1
    ' 3 15
echo "scale=3 \n scale" > /tmp/bc${$}
echo "scale = \c"
bc /tmp/bc${$}
rm /tmp/bc${$}

$
```

This program prints two lines to the file /tmp/bc${$}.

```
scale=3
scale
```

The program then prints "scale = " to the terminal. The program then executes the command **bc**. **bc** is a program that will do simple arithmetic. The default is to do integer arithmetic. If **bc** has an argument, **bc** will read that file and execute each line. In /tmp/bc${$}, the first line sets the scale factor to 3 and the second line will print out the scale factor; so our program prints "scale = " to standard output and **bc** prints "3" to standard output. **bc** is terminated by <ctrl>D.

This program ignores the signal generated by the break key. If the signal generated by a <ctrl>! is entered, the file /tmp/bc${$} is removed and the program exits with an exit status of 1.

EXAMPLE USING TRAP

```
$ cat mybc<CR>
trap "" 2
trap '
        rm /tmp/bc${$}
        exit 1
    ' 3 15
echo "scale=3 \n scale" > /tmp/bc${$}
echo "scale = \c"
bc /tmp/bc${$}
rm /tmp/bc${$}

$
```

trap STATEMENT — QUOTING CONSIDERATIONS

When the shell initially reads in a **trap** statement, it evaluates the entire statement and saves it in case the signal is received. If the signal is received, then the command is reevaluated. A problem occurs when, for example, the trap setting contains a parameter substitution expression for which the variable was set during the execution of the program. If that is the case, then the first time the **trap** statement is read by the shell, the substitution takes place, thereby replacing the expression with a null value.

This is prevented by enclosing the entire trap setting in single quotes. This prevents the shell from doing any substitutions when the command line is initially read. They are held off until the **trap** is invoked.

trap STATEMENT — QUOTING CONSIDERATIONS

- COMMANDS WITHIN **trap** ARE SCANNED TWICE
  - — ONCE WHEN READ IN
  - — ANOTHER TIME WHEN EXECUTED
- USING DOUBLE QUOTES
  - — ALL VARIABLE AND COMMAND SUBSTITUTION EXPRESSIONS ARE EVALUATED WHEN THE SHELL FIRST READS THE SIGNAL PROCESSING ROUTINE
- USING SINGLE QUOTES
  - — PREVENTS THE SHELL FROM INITIALLY DOING ANY SUBSTITUTIONS WHEN THE COMMAND LINE IS READ IN

VOLUME 15 EXERCISE

1. Write a shell procedure named **catch** that will:

   i. Catch the delete character to set the erase character to be <ctrl>c and set the kill character to be <ctrl>x and print their new settings.

   ii. Catch the quit character to set the erase character to be <ctrl>h and set the kill character to be '@' and print their new settings.

   iii. Read standard input and echo it out until done is entered.

   iv. Reset the erase and kill characters back to normal.

   When executing your shell procedure, test the delete and quit characters and use your new erase and kill characters.

2. Edit your **.profile** to do the following:

   i. Create a variable **START** and set it to the time you logged in. Hint; use command substitution.

   ii. Catch the hangup signal (signal 0). When the signal comes in, print a message of the form:

   Log in session on 09/24/87 from 08:24:32 AM to 02:18:41 PM.

   and then exit.

VOLUME 15 EXERCISE - ANSWERS

1. `$ cat catch<CR>`

```
# catch - catch the delete and quit characters
#         and reset erase and kill characters
trap "stty erase '^c' echoe
      stty kill '^x'
      echo erase character is <ctrl>c
      and kill character is <ctrl>x." 2
trap "stty erase '^h' echoe
      stty kill '@'
      echo erase character is <ctrl>h and kill character is '@'." 3
while
    echo "Enter command line: \c"
    read input
    test "${input}" = "done"
do
    echo ${input}
done
stty erase '^h' echoe
stty kill '@'
```

2. `$ cat .profile<CR>`

```
trap 'echo Login session on `date +%D` from ${START} to `date +%r`' 0
START="`date +%r`"
stty erase '^h' echoe
umask 027
CDPATH=.:${HOME}
PATH=${PATH}:${HOME}/bin
PS1="${LOGNAME}: "
echo "Enter terminal type: \c"
read TERM
export PATH PS1 CDPATH TERM
date
echo There are `who | wc -l` users on the system `uname`
```

VOLUME 15 SUMMARY

Signal	Meaning
00	Normal Program Termination
01	Hangup
02	Interrupt (DELETE)
03	Quit
04	Illegal instruction
05	Trace trap
06	IOT instruction
07	EMT instruction
08	Floating point exception
10	Bus error
11	Segmentation violation (accessing and address out of address space)
12	Bad argument to a system call
13	Write to a pipe with no one to read from it
14	Alarm clock
15	Kill signal