# Answers to Even-Numbered Exercises 15

*from page 833*

1. The **dirname** utility treats its argument as a pathname and writes to standard output the path prefix, that is, everything up to but not including the last component. Thus

   ```
   dirname a/b/c/d
   ```

   writes a/b/c to standard output. If path is a simple filename (has no **/** characters), **dirname** writes a **.** to standard output.

   Implement dirname as a Z Shell function. Make sure that it behaves sensibly when given such arguments as **/**.

2. Implement the basename utility, which writes the last component of its pathname argument to standard output, as a Z Shell function. For example,

   ```
   zsh % basename a/b/c/d
   ```

   writes d to standard output.

   The following function is named **bn** so that you know that you are not running the **/bin/basename** utility. It behaves the same way as basename.

```
% function bn {
function> export argone=$1
function>
function> if [ $# = 0 ]
function if>        then
function then>        exit 1
function then>    elif [ "$1" = "/" ]
function elif>        then
function elif-then>        echo /
function elif-then>     else
function else>        echo $1 | sed 's:.*/::'
function else> fi
function> }
```

3. The GNU/Linux basename utility has an optional second argument. If you type

   *basename **path suffix***

   basename removes the ***suffix*** from ***path*** after removing the prefix. For example,

   ```
   zsh % basename src/shellfiles/prog.bash .bash
   prog
   zsh % basename src/shellfiles/prog.bash .c
   prog.bash
   ```

   Add this feature to the function you wrote for exercise 2.

4. Write a Z Shell function that takes a directory name as an argument and writes to standard output the maximum of the lengths of all filenames in that directory. If the function's argument is not a directory name, write an error message to standard output and exit with nonzero status.

   ```
   % function maxfn {
   function> integer max thisone
   function>
   function> if [ ! -d $1 -o $# = 0 ]
   function if>     then
   function then>     echo "Usage: maxfn dirname"
   function then>     return 1
   function then> fi
   function>
   function> max=0
   function> for fn in $(/bin/ls $1)
   function for>     do
   function for>     thisone=${#fn}
   function for>     if [ $thisone -gt $max ]
   function for if>        then
   function for then>        max=$thisone
   function for then>     fi
   function for>      one
   function> echo "Longest filename is $max characters."
   function> }
   ```

5. Modify the function you wrote for exercise 4 to descend all subdirectories of the named directory recursively and to find the maximum length of any filename in that hierarchy.

6. Write a Z Shell function that lists the number of regular files, directories, block special files, character special files, FIFOs, and symbolic links in the working directory. Do this in two different ways:

   a. Use the first letter of the output of **ls –l** to determine a file's type.

```
% function ft {
function> integer reg dir blk char fifo symlnk other
function>
function> for fn in $(ls)
function for>     do
function for>     case $(ls -l $fn | cut -b1) in
function for case>         d)
function for case>             ((dir=$dir+1))
function for case>             ;;
function for case>         b)
function for case>             ((blk=$blk+1))
function for case>             ;;
function for case>         c)
function for case>             ((char=$char+1))
function for case>             ;;
function for case>         p)
function for case>             ((fifo=$fifo+1))
function for case>             ;;
function for case>         l)
function for case>             ((symlnk=$symlnk+1))
function for case>             ;;
function for case>         a-z)
function for case>             ((other=other+1))
function for case>             ;;
function for case>         *)
function for case>             ((reg=reg+1))
function for case>             ;;
function for case>     esac
function for>     done
function>
function> echo $reg regular
function> echo $dir directory
function> echo $blk block
function> echo $char character
function> echo $fifo FIFO
function> echo $symlnk symbolic link
function> echo $other other
function> }
```

   b. Use the file type condition tests of the [[ builtin to determine a file's type. Note: As of zsh 4.0.4 (distributed with Red Hat 8) a **–d** test for a directory returns *true* for a symbolic link, thus the test for a symbolic link is moved to the beginning of the tests.

```
% function ft2 {
function> integer reg dir blk char fifo symlnk other
function>
function> for fn in $(ls)
function for>     do
function for>         if [[ -h $fn ]]
function for if>             then ((symlnk=$symlnk+1))
function for then>         elif [[ -f $fn ]]
function for elif>             then ((reg=reg+1))
function for elif-then>         elif [[ -d $fn ]]
function for elif>             then ((dir=$dir+1))
function for elif-then>         elif [[ -b $fn ]]
function for elif>             then ((blk=$blk+1))
function for elif-then>         elif [[ -c $fn ]]
function for elif>             then ((char=$char+1))
function for elif-then>         elif [[ -p $fn ]]
function for elif>             then ((fifo=$fifo+1))
function for elif-then>         elif [[ -q $fn ]]
function for elif>             then ((other=other+1))
function for elif-then> fi
function for>     done
function>
function> echo $reg regular
function> echo $dir directory
function> echo $blk block
function> echo $char character
function> echo $fifo FIFO
function> echo $symlnk symbolic link
function> echo $other other
function> }
```

7. The **makercs** program (page 812) depends on the fact that find writes the pathname of a directory before writing the pathname of any files in that directory. Suppose that this were not reliably true. Fix **makercs**.

8. Change **makercs** (page 812) so that if any call to ci fails, the program continues (as it does now) but eventually exits with nonzero status.

   Before the line `checkargs "$@" initialize` the **retflg** variable to zero:

   ```
   retflg=0
   ```

   Change the ci command to

   ```
   ci -l -q "-t-$pathname" "$pathname" "$target" >&4 2>&3 ||
           (print -u3 "Cannot create $target" && retflg=1)
   ```

   Finally, change the exit command to

   ```
   exit $retflg
   ```

9. Modify the **quiz** program (page 816) so that the choices for a question are also randomly arranged.

## Advanced Exercises

10. In the **makercs** (page 812) program, file descriptors 3 and 4 are opened; during the loop, output is directed to these descriptors. An alternative method would be simply to append the output each time it occurs, using, for example,

    ```
    print "Cannot create $target" >> $ERRS
    ```

    rather than

    ```
    print -u3 "Cannot create $target"
    ```

    What is the difference? Why does it matter?

    Redirecting the output opens the file (descriptor), writes to the file, and closes the file (descriptor). Opening a file descriptor at the start of a program, writing to it as needed, and closing it when the program finishes is more efficient than the first method, especially when you write to the file many times.

11. The check in **makercs** (page 812) to prevent you from copying hierarchies on top of each other is simplistic. For example, if you are in your home directory, the call **makercs . ~/work/RCS** will not detect that the source and target directories lie on the same path. Fix this check.

12. In principle, recursion is never necessary. It can always be replaced by an iterative construct, such as **while** or **until**. Rewrite makepath (page 809) as a nonrecursive function. Which version do you prefer? Why?

    ```
    makepath2()
    {
    wd=$(pwd)
    pathname=$1

    while [[ $pathname = */* && ${#pathname} > 0 ]]
            do
            if [[ ! -d $pathname ]]
                    then
                    mkdir "${pathname%%/*}"
            fi
            cd "${pathname%%/*}"
    ```

```
        pathname="${pathname#*/}"
        done
if [[ ! -d $pathname && ${#pathname} > 0 ]]
        then
        mkdir $pathname
fi
cd $wd
}
```

The recursive version is simpler: There is no need to keep track of the working directory and you do not have to take care of making the final directory separately.

13. Lists are commonly stored in environment variables by putting a colon (:) between each of the list elements. (The value of the **PATH** variable is a good example.) You can add an element to such a list by catenating the new element to the front of the list, as in

```
PATH=/opt/bin:$PATH
```

If the element you add is already in the list, you now have two copies of it in the list. Write a Z Shell function, **addenv** that takes two arguments: (1) the name of a shell variable and (2) a string to prepend to the list that is the value of the shell variable only if that string is not already an element of the list. For example, the call

```
addenv PATH /opt/bin
```

would add **/opt/bin** to **PATH** only if that pathname is not already in **PATH**. Be sure that your solution works, even if the shell variable starts out empty. Also make sure that you check the list elements carefully. If **/usr/opt/bin** is in **PATH** but **/opt/bin** is not, the example just given should still add **/opt/bin** to **PATH**. (*Hint:* You may find this easier to do if you first write a function **locate_field** that tells you whether a string is an element in the value of a variable.)