Assignment 4

This assignment will give you some more practice with bash programming. When you write a bash script, you are really writing a program in the bash programming language. You may not have thought about it in this way before, but that is precisely what you have been doing. Why do I say this now? Because shortly you will begin writing programs in another programming language, Perl, and now is a good time to realize that this is not a giant step but a sequence of small steps in mastering Perl.

We have been calling your programs shell scripts. A script is a program, make no bones about it. Scripts are programs written in a scripting language, which is a special kind of programming language. All scripting languages are programming languages, but not vice versa. The distinction will be explained in a later lecture. I will call bash both a programming language and a scripting language.

A bash instruction is also called a statement. For example, the if-instruction

```bash
if test $# -ne 2
then
    echo "usage: $0 arg1 arg2"
    exit
fi
```

is usually referred to as the if-statement. From this point forward, I will call instructions statements. The bash programming language has several statements that are known as looping statements. A looping statement is one that makes it possible to repeat a sequence of statements one or more times. For example, bash has a looping statement called a while-statement, whose form (syntax) is

```bash
while <expression>
do
    <list-of-statements>
done
```

in which <expression> is a statement such as the test command, or any other statement that is evaluated as being true or false, and <list-of-statements> is any sequence of statements (including looping statements.) The following snippet of a script shows one example of a while-statement:

```bash
echo "Try to guess my favorite color:"
read guess rest_of_line
mycolor='cat secretfile'
while [ $guess != $mycolor ]
do
    echo "Sorry, that is not my favorite color. Try again. "
    read guess rest_of_line
done
```

Here, the expression is [ $guess != $mycolor ] and the <list-of-statements> is the list of two lines

```bash
echo "Sorry, that is not my favorite color. Try again. "
read guess rest_of_line
```
The above script will test whether guess is the same string as mycolor, and if it is not, it will execute the list and then re-compare guess and mycolor. It will keep doing this until the user enters a string that is identical to the string stored in mycolor. When she does, the expression becomes false and the “while loop” is exited. A while-statement is usually called a while loop because if we visualize the sequence of executed statements as being connected by an imaginary thread, then this thread loops around and around the lines of the script.

Bash also has a for-loop and an until-loop. I will not discuss the until-loop here. The for-loop is very different from the while-loop. It has two forms. One form (again the proper term is syntax) is

```bash
for <variable> in <argument-list>
do
   <list-of-statements>
done
```

and the other is

```bash
for <variable>
do
   <list-of-statements>
done
```

The <variable> can be any valid variable name (words starting with letters and containing letters, digits, and the underscore character.) The <argument-list> can be any sequence of words, including words that look like numbers. Examples are

```bash
for number in 1 2 3 4 5 6 7 8 9 10
   for name in John Jacob Judy Jocelyn
   for file in `ls .`
```

As you can see, this can be very powerful. As with the while-loop, the list of statements is any list of statements, but the intention is that the variable plays a role in this list. For example, the script

```bash
let sum=0
for number in 1 2 3 4 5 6 7 8 9 10
do
   let square=$number*$number
   let sum=$sum+$number
   echo The square of $number is $square
done
echo The sum of the numbers is $sum.
```

displays ten lines showing the squares of the first ten positive integers and then displays their sum. Notice how the sum is calculated.

The second form of the for-loop does not need an argument list. It automatically assigns to the variable the successive words from the command line arguments of the script when it is run:

```bash
for name
do
   if [ 0 -lt ` who | grep -c $name` ]
      then
echo $name is logged in.
   fi
done
```

The if-statement tests whether the output of the pipe is greater than zero. If it is, then the name is a user who is logged in at least once.
Tasks

1. Write a bash script that takes a list of login names as its command line arguments and displays these login names and the full names of the users who own these login names (as shown in the network password file which is displayed with the ypcat passwd command.) The format of your output lines should be

   login-name: real-name

with one line per entered name. If there is no such name in the network password file, do not display any line for that name. Name this script realname. I should be able to run a command such as

   $ realname sweiss tbw
   sweiss: Stewart Weiss
   tbw: Tom Walter

2. For this task, assume that we have a file that contains a list of file names, one per line. Write a bash script that takes the name of such a file as its command line argument. Its output will be, for each file named in that file, the file name and the number of bytes in that file, in the format

   file: 93345

If the file does not exist or cannot be opened, there should be no line in the output for that file. After all files and their sizes have been displayed, the script should display the total number of bytes in the “large” files on a single line. For this purpose, a file is considered large if its size is greater than 100,000 bytes. To be clear, the sum is only the sum of those files whose size is at least 100,000 bytes. Name this script filesize.

Submitting the Solution

Create a directory in /data/biocs/b/student.accounts/cs132/projects/project4 named hwk4_username. Make sure that you use only lowercase letters in the directory name so that it matches your username exactly. Remember, this is a directory, not a file. Give your directory permission rwx------ (700). Put your two scripts into this directory, with the names specified above, e.g., realname and filesize. These scripts should have permission rwxr-xr-x. (No one can see them since they are protected by the permissions on the directory.) Make sure that each script has your name in a comment line at the top.