Shell Scripting

Introduction

The first script that we will use will be a simple script that will output the text "Hello Unix". I would suggest using **vi** to create and edit the file. To create and edit the file, run the following command:

```
vi hello.sh
```

If you don't know how to use vi, I would suggest that you read a quick tutorial by searching for "vi tutorial" with the search bar above. To enter input mode in vi, press "i". Now, for the code in this first script, enter the following code in the file:

```
#!/bin/sh
# This is my first script.

echo "Hello Unix"
```

Now save the file and close it by hitting Escape followed by ":wq" and Return. The first line of the file tells unix which shell to use to execute the file. /bin/sh is the default location for the **bourne shell**. In Linux this will normally point to the **bourne again shell**, which is a remake of the original unix shell and works pretty much the same. The second line of the file is just a simple comment. Comments are ignored by the shell interpreter but are very useful when developing large and complex scripts. Everyone forgets what their original logic or intention was when coding a script and it's much easier to read a comment than it is to try to understand large and complex sections of code. Before we can run this script, we must first make it executable. To do this we will use the unix chmod command:

```
chmod u+x hello.sh
```

Now our script is executable. This command basically tells unix to set the x (executable) flag for the user level access of the file. Now we are able to run the file. If you don't have "." in your unix PATH environment variable, then you will need to proceed the name of the script with "./" to execute it. It is generally considered to be a security risk to put "." in your PATH evironment variable, so we will assume that you don't have it. Now you can execute your script by using the following command:

```
./hello.sh
```

You will see the text "Hello Unix" output to the console, congratulations, you have created your first unix script! Now you can move on to the next topic where we will discuss using variables in a unix script.

Variables

Variables are an important part of any program or script. A variable is a simple way to refer to a chunk of data in memory that can be modified. A variable in a unix script can be assigned any type of value, such as a text string or a number. In unix to create a variable, we simply put in our script:

```
VARIABLE_NAME=value
```

Note that we do not have to put the variable name in uppercase, but it is the standard way of naming variables in unix. The text "VARIABLE_NAME" can be anything you want, as long as it only contains numbers, letters and/or an underscore "_". A variable name also cannot start with a number. After the equal sign you put the value, there must be no space between the variable name and the equal sign. To use the variable, we simply put a dollar sign "$" before the name of the variable in our script code. Let's revise our original script to use the two words in two variables as such:

```
#!/bin/sh
# This is my second script.
VAR_1=Hello
VAR_2=Unix

echo "$VAR_1 $VAR_2"
```

This gives the same output as the original script but uses two variables. This is not a very exciting use of variable I will admit, let's try something more interesting. Let's write a program that will open a file and read the head and tail of it. This can be useful if you want to see the first items in your log and the last items in your log but you don't want to see the whole log file, which could be very large. To write this program, we will make use of the unix commands **head** and **tail**. Our script will use a predefined variable called **$1**. This variable will display the first item in the list of command line arguments to your script. You can also access any other command line argument from **$1** to **$9**. You can also use the variable **$@** to access all of the command line arguments in a single variable.

```
#!/bin/sh
# This program will print the head and tail of a file
# passed in on the command line.

echo "Printing head of $1..."
head $1

echo ""  #this prints an extra return...
echo "Printing tail of $1..."
tail $1
```

Give this program a name of ht.sh (for Head Tail shell script), be sure to give it execute permission and then run it with the following command:

```
./ht.sh WEB_LOG
```

The parameter **WEB_LOG** must point to an actual text file on your system, it will open the file and print the head and tail. Now let's assume that we want the user to be able to input the file name after the script has been run. We can read input from the console by using the unix **read** command. Let's modify the program and try it again:

```
#!/bin/sh
# This program will read the filename from user input.

echo "Enter the file: "
read FILENAME
echo "Printing head of $FILENAME..."
head $FILENAME

echo ""  #this prints an extra return...
echo "Printing tail of $FILENAME..."
```

```
tail $FILENAME
```

This way of printing a variable does have limitations, for instance you cannot print text without a space after the variable name. Suppose that we want to create a script that will read the head and tail of the file as before, but we know that the user will always be using files that end with "_LOG". If we tried to open the file $FILENAME_LOG, then it would look for a variable with that specific name and not a file named $FILENAME + _LOG. We can fix this by using another way of displaying and using a variable. We use the format **${VARIABLE_NAME}**. Let's modify our program again to automatically append "_LOG" to the end of the filename for our users:

```
#!/bin/sh
# This program will read the filename from user input.

echo "Enter the file: "
read FILENAME
echo "Printing head of ${FILENAME}_LOG..."
head ${FILENAME}_LOG

echo ""  #this prints an extra return...
echo "Printing tail of ${FILENAME}_LOG..."
tail ${FILENAME}_LOG
```

Now if you run the script with the command **./ht.sh WEB** it will open the file "WEB_LOG" for the user.

If/Else

In order for a script to be very useful, you will need to be able to test the conditions of variables. Most programming and scripting languages have some sort of if/else expression and so does the bourne shell. Unlike most other languages, spaces are very important when using an **if** statement. Let's do a simple script that will ask a user for a password before allowing him to continue. This is obviously not how you would implement such security in a real system, but it will make a good example of using **if** and **else** statements.

```
#!/bin/sh
# This is some secure program that uses security.

VALID_PASSWORD="secret" #this is our password.

echo "Please enter the password:"
read PASSWORD

if [ "$PASSWORD" == "$VALID_PASSWORD" ]; then
        echo "You have access!"
else
        echo "ACCESS DENIED!"
fi
```

Remember that the spacing is very important in the if statement. Notice that the termination of the if statement is **fi**. You will need to use the **fi** statement to terminate an if whether or not use use an else as well. You can also replace the "==" with "!=" to test if the variables are NOT equal. There are other tokens that you can put in place of the "==" for other types of tests. The following table shows the different expressions allowed.

Comparisons:

-eq	equal to
-ne	not equal to
-lt	less than
-le	less than or equal to
-gt	greater than
-ge	greater than or equal to

File Operations:

-s	file exists and is not empty
-f	file exists and is not a directory
-d	directory exists
-x	file is executable
-w	file is writable
-r	file is readable

Let's try using a couple of these in a script. This next script will ask for a user name, if there is not a file that exists with the name "username_DAT", the script will prompt the user for their age, it will then make sure that they are old enough to use this program and then it will write their age to a file with the name "username_DAT". If the file already exists, it will just display the age of the user.

```
#!/bin/sh

# Prompt for a user name...
echo "Please enter your name:"
read USERNAME

# Check for the file.
if [ -s ${USERNAME}_DAT ]; then
        # Read the age from the file.
        AGE=`cat ${USERNAME}_DAT`
        echo "You are $AGE years old!"
else
        # Ask the user for his/her age
        echo "How old are you?"
        read AGE

        if [ "$AGE" -le 2 ]; then
                echo "You are too young!"
        else
                if [ "$AGE" -ge 100 ]; then
                        echo "You are too old!"
                else
                        # Write the age to a new file.
                        echo $AGE > ${USERNAME}_DAT
                fi
        fi
fi
```

Run this program a couple of times. First run it and give it the user name of "john". When it asks for an age, enter the age "1". Notice that it will say that you are too you and then exit. Now run the program again with the name "john" and the age 200. This time the script will tell you that you are too old and exit. Now run the the script again with the name of "john", enter the age 30. The script exits normally this time, the program created a file called "john_DAT" which contains the text "30". Finally run the program one more time and give it the name "john". This time it will not prompt you to enter an age, instead it will read the age from a file and say "Your are 30 years old!".

We introduced something else new in this script. On line 10 of the file, we see the code:

```
        AGE=`cat ${USERNAME}_DAT`
```

This is how you execute a command and put the text output from the command into a variable. The unix command **cat** reads the file named **${USERNAME}_DAT** and outputs it to the console. Instead of putting it to the console in our script, we wrap the command with the character **`**, this puts the text into our variable AGE.

You can test multiple expressions at once by using the **||** (or) operator or the **&&** (and) operator. This can save you from writing extra code to nest if statements. The above code has a nested if statement where it checks if the age is greater than or equal to 100. This could be changed as well by using **elif** (else if). The structure of **elif** is the same as the structure of **if**, we will use it in an example below. In this example, we will check for certain age ranges. If you are less than 20 or greater than 50, you are out of the age range. If you are between 20 and 30 you are in your 20's and so on.

```
#!/bin/sh

# Prompt for a user name...
echo "Please enter your age:"
read AGE

if [ "$AGE" -lt 20 ] || [ "$AGE" -ge 50 ]; then
        echo "Sorry, you are out of the age range."
elif [ "$AGE" -ge 20 ] && [ "$AGE" -lt 30 ]; then
        echo "You are in your 20s"
elif [ "$AGE" -ge 30 ] && [ "$AGE" -lt 40 ]; then
        echo "You are in your 30s"
elif [ "$AGE" -ge 40 ] && [ "$AGE" -lt 50 ]; then
        echo "You are in your 40s"
fi
```

Looping

While Loop

The **while** statement is used when you want to loop while a statement is true. This is the same in many other programming and scripting languages. The body of the loop is put between **do** and **done**. Suppose that we want to write a script to have the user guess what number we are thinking of. The best way to use this would be to use a **while** loop.

```
#!/bin/sh
# Guess the number game.

ANSWER=5          # The correct answer
CORRECT=false     # The correct flag

while [ "$CORRECT" != "true" ]
do
        # Ask the user for the number...
        echo "Guess a number between 1 and 10. "
        read NUM

        # Validate the input...
        if [ "$NUM" -lt 1 ] || [ "$NUM" -gt 10 ]; then
                echo "The number must be between 1 and 10!"
        elif [ "$NUM" -eq "$ANSWER" ]; then
                echo "You got the answer correct!"
                CORRECT=true
        else
                echo "Sorry, incorrect."
        fi
done
```

You can also loop while reading a variable to make the code simpler. Let's try the above example revised:

```
#!/bin/sh
# Guess the number game.  Version 2.0

ANSWER=5          # The correct answer

echo "Guess a number between 1 and 10. "

while read NUM
do
        # Validate the input...
        if [ "$NUM" -lt 1 ] || [ "$NUM" -gt 10 ]; then
                echo "The number must be between 1 and 10! Guess again. "
        elif [ "$NUM" -eq "$ANSWER" ]; then
                echo "You got the answer correct!"
                exit
        else
                echo "Incorrect, guess again. "
        fi
done
```

Another way to loop forever is to use the **:** in your **while** statement. Let's write a simple program that will print out how long it has been running until the user presses Ctrl+C to terminate the program.

```
#!/bin/sh

COUNTER=0

while :
do
```

```
        sleep 1
        COUNTER=`expr $COUNTER + 1`
        echo "Program has been running for $COUNTER seconds..."
done
```

This program will loop until the user presses Ctrl+C. Notice that we use something else new here. The unix **expr** command is used to evaluate a mathematical expression. If you want to increment a variable, this is the command that you will use.

For Loop

The **for** statement is used when you want to loop through a list of items. The body of the loop is put between **do** and **done**. Let's say that we want to write a program that will validate numbers in a given list. These numbers can be loaded from a file, hard coded, or manually entered by the user. For our example, we will ask the user for a list of numbers separated with spaces. We will validate each number and make sure that it is between 1 and 100. The best way to write a program like this would be to use a **for** loop.

```
#!/bin/sh
# Validate numbers...

echo "Please enter a list of numbers between 1 and 100. "
read NUMBERS

for NUM in $NUMBERS
do
        if [ "$NUM" -lt 1 ] || [ "$NUM" -gt 100 ]; then
                echo "Invalid Number ($NUM) - Must be between 1 and 100!"
        else
                echo "$NUM is valid."
        fi
done
```

Notice that we give the **for** statement a variable "NUM", this can be whatever variable name you want to use. It will loop through each item in the given variable (in this case $NUMBERS) and put that item in the $NUM variable. Run the file and give it the input "1 4 3 55 48 120 1000 4 1" and you will have the following output:

```
Please enter a list of numbers between 1 and 100.
1 4 3 55 48 120 1000 4 1
1 is valid.
4 is valid.
3 is valid.
55 is valid.
48 is valid.
Invalid Number (120) - Must be between 1 and 100!
Invalid Number (1000) - Must be between 1 and 100!
4 is valid.
1 is valid.
```

Cases

Many programming languages and scripting languages have the concept of a **case** or select statement.

This is generally used as a shortcut for writing if/else statements. The **case** statement is always preferred when there are many items to select from instead of using a large if/elif/else statement. It is usually used to implement menus in a script. The **case** statement is terminated with **esac** (case backwards). Here is a simple example of using a **case** statement:

```
#!/bin/sh

echo "Enter a number between 1 and 10. "
read NUM

case $NUM in
        1) echo "one" ;;
        2) echo "two" ;;
        3) echo "three" ;;
        4) echo "four" ;;
        5) echo "five" ;;
        6) echo "six" ;;
        7) echo "seven" ;;
        8) echo "eight" ;;
        9) echo "nine" ;;
        10) echo "ten" ;;
        *) echo "INVALID NUMBER!" ;;
esac
```

The **1**, **2**, etc. are the different values that could possibly be contained in the given variable ($NUM). The ***** indicates how you will handle any value that is unexpected. This application will keep track of how many people want chicken and how many want steak for a wedding reception. The user will also have an option to exit.

```
#!/bin/sh
# Wedding guest meals

# These variables hold the counters.
NUM_CHICKEN=0
NUM_STEAK=0
ERR_MSG=""

# This will clear the screen before displaying the menu.
clear

while :
do
        # If error exists, display it
        if [ "$ERR_MSG" != "" ]; then
                echo "Error: $ERR_MSG"
                echo ""
        fi

        # Write out the menu options...
        echo "Chicken: $NUM_CHICKEN"
        echo "Steak: $NUM_STEAK"
        echo ""
        echo "Select an option:"
        echo " * 1: Chicken"
        echo " * 2: Steak"
        echo " * 3: Exit"
        # Clear the error message
        ERR_MSG=""

        # Read the user input
        read SEL

        case $SEL in
                1) NUM_CHICKEN=`expr $NUM_CHICKEN + 1` ;;
                2) NUM_STEAK=`expr $NUM_STEAK + 1` ;;
                3) echo "Bye!"; exit ;;
                *) ERR_MSG="Please enter a valid option!"
        esac

        # This will clear the screen so we can redisplay the menu.
        clear
done
```

Since we clear the screen before displaying the menu each time, we must keep track of any error messages and display them if an error occurs before we draw the menu and after we clear the screen. The screen is cleared by using the unix **clear** command. Notice in option 3 that you can separate unix commands on a single line by using the **;** (semicolon).

Functions

When your scripts start to become very large, you may tend to notice that you are repeating code more often in your scripts. You have the ability to create functions inside of your script to help with code reuse. Writing the same code in multiple sections of your script can lead to severe maintenance problems. When you fix a bug in a section of code you need to be sure that all sections of code that are repeated will also have those fixes. A function is a block of code that can be called from other parts of your script. It can have parameters passed to it as if it were a separate script itself. As an example, we will create a function called **logit**, which will take two parameters, a level and a message. The level will be a number between 1 and 3 that will indicate the severity of the log message. The level of messages that you want to view will be passed in the command line of the script.

```
#!/bin/sh

# logit function declaration.
logit()
{
        MSG_LEVEL=$1

        # Shifts the position of the parameters over one place.
        shift

        if [ "$MSG_LEVEL" -ge 1 ] && [ "$MSG_LEVEL" -le 3 ]; then
                if [ "$LEVEL" -eq 1 ] && [ "$MSG_LEVEL" -ge 1 ]; then
                        echo "Msg Level $MSG_LEVEL: $@"
                elif [ "$LEVEL" -eq 2 ] && [ "$MSG_LEVEL" -ge 2 ]; then
                        echo "Msg Level $MSG_LEVEL: $@"
                elif [ "$LEVEL" -eq 3 ] && [ "$MSG_LEVEL" -ge 3 ]; then
                        echo "Msg Level $MSG_LEVEL: $@"
                fi
        fi
}

#Load the log level from the command line...
LEVEL=$1

# Call the function a couple of times.
logit 1 Logit Test one
logit 2 Logit Test two
logit 3 Logit Test three
logit 4 Logit Test four
```

Let's save this file and run it several times with different input levels...

```
$ test.sh 1
Msg Level 1: Logit Test one
Msg Level 2: Logit Test two
Msg Level 3: Logit Test three
$ test.sh 2
Msg Level 2: Logit Test two
Msg Level 3: Logit Test three
$ test.sh 3
Msg Level 3: Logit Test three
$ test.sh 4
$
```

In this script we use the unix command **shift** to shift the position of the **$@** pointer to the right. Remember that **$@** points to the entire command line. In the case of functions, the command line parameters are localized to the function, so for the first call to logit, the following parameters are passed "1 Logit Test one". Inside the logit function, the $@ variable contains the data "1 Logit Test one". We already read the first parameter ("1") into our MSG_LEVEL variable, so we want to use the other parameters are the text message. access them all together without the first parameter, we call **shift**, which moves the $@ variable one to the right so that it now contains the text "Logit Test one".

Another interesting thing in this script is that for the $LEVEL variable, the scope does not matter. Remember that in a unix script, a varible is just an environment variable and is accessible from anywhere.

You can write recursive functions in a shell script as well. Consider the factoral:

```
#!/bin/sh

fac()
{
        if [ "$1" -gt 1 ]; then
                NEXT=`expr $1 - 1`
                REC=`fac $NEXT`
                PROD=`expr $1 \* $REC`
                echo $PROD
        else
                echo 1
        fi
}

echo "Enter a number: "
read NUM
echo "$NUM! = `fac $NUM`"
```

A recursive function is a function that calls itself. Notice at line 7 in the script, it calls the fac function inside itself. Recursive functions have the possibility of going in an endless loop and crashing if you do not code them right, so only use a recursive function if you really understand what you are trying to do.

Searching & Sorting

grep

The unix **grep** command is a simple but powerful tool for using in your scripts. You can use it to search in files for certain words or even search by regular expression. Before we get into using these tools, let's define a file that we can manipulate. Create a file on your filesystem with the following contents:

```
root 192.168.1.1 10/11/2005 /usr/local/bin/one_app
root 192.168.1.1 10/12/2005 /usr/local/bin/two_app
root 192.168.1.1 10/12/2005 /var/logs/system.log
root 192.168.1.2 10/13/2005 /var/logs/approot.log
user1 192.168.1.3 10/13/2005 /usr/local/bin/one_app
user1 192.168.1.3 10/13/2005 /usr/local/src/file.c
user1 192.168.1.3 10/14/2005 /var/logs/system.log
user2 192.168.1.4 10/14/2005 /var/logs/approot.log
user2 192.168.1.5 10/15/2005 /usr/local/bin/two_app
user2 192.168.1.5 10/15/2005 /usr/local/bin/two_app
```

Save this file as "testfile". The file is an access log for the file system. It has four fields separated by spaces: user, ip address, date and filename. Files such as this can get very large and hard to find things when using just a file editor. Suppose that we wanted to find all rows that for the file "/usr/local/bin/one_app". To do this, we would use the following command and get the following results:

```
$ grep "/usr/local/bin/one_app" testfile
root 192.168.1.1 10/11/2005 /usr/local/bin/one_app
user1 192.168.1.3 10/13/2005 /usr/local/bin/one_app
```

This makes the file much easier to search through. You can also redirect the output of a command by using > **filename** after the command. For instance, let's say that we want to find all rows for the user name "root" and redirect it to a file. If we simply did a grep for "root", we would also pick up any rows that are for access to the file "/var/logs/app**root**.log". What we really want to do is find any line that starts with "root". To do this, we will use the regular expression character ^, which means "starts with". We will call this command and have it redirect the output to the file "output.txt", then we will use the unix **cat** command to display the output.txt file.

```
$ grep "^root" testfile > output.txt
$ cat output.txt
root 192.168.1.1 10/11/2005 /usr/local/bin/one_app
root 192.168.1.1 10/12/2005 /usr/local/bin/two_app
root 192.168.1.1 10/12/2005 /var/logs/system.log
root 192.168.1.2 10/13/2005 /var/logs/approot.log
```

awk

If you only want to view certain fields in the file, you will want to use the unix **awk** command. If you are using linux, this will most likely be called **gawk**, but for this tutorial I will use the unix name for the command. Let's say that we want to see all files that were accessed that end with the text "_app". We don't want to see the whole rows, we only want to see column number 4 in the file (the filename). In order to do this, we will need to use both **grep** and **awk**, then we will need to **pipe** the output from one command to the other. To find a line that ends with a certain text, we use the regular expression character **$** at the end of the text. See the following example:

```
$ grep "_app$" testfile | awk '{print $4}'
/usr/local/bin/one_app
/usr/local/bin/two_app
/usr/local/bin/one_app
/usr/local/bin/two_app
/usr/local/bin/two_app
```

We use the **|** character to pipe the output of the **grep** command to the input of the **awk** command. To print the fourth column of the input data, we give awk the script contents '{print $4}'. If we want to display the user name as well, we can print column **$1**, but we must separate the two columns with a comma.

```
$ grep "_app$" testfile | awk '{print $1, $4}'
root /usr/local/bin/one_app
root /usr/local/bin/two_app
user1 /usr/local/bin/one_app
user2 /usr/local/bin/two_app
user2 /usr/local/bin/two_app
```

sort

Another common need in scripts is the ability to sort input. Luckily unix has a **sort** command. All you need to do is pipe your output to the **sort** command and it will be sorted. If you want to see all files (column 4) in the file and you want them sorted, use the following command:

```
$ awk '{print $4}' testfile | sort
/usr/local/bin/one_app
/usr/local/bin/one_app
/usr/local/bin/two_app
/usr/local/bin/two_app
/usr/local/bin/two_app
/usr/local/src/file.c
/var/logs/approot.log
/var/logs/approot.log
/var/logs/system.log
/var/logs/system.log
```

Another common need related to sorting is to get only unique items. The unix **sort** command has a flag **-u** that tells it to only display unique items. Let's use this command to only see unique file names.

```
$ awk '{print $4}' testfile | sort -u
/usr/local/bin/one_app
/usr/local/bin/two_app
/usr/local/src/file.c
/var/logs/approot.log
/var/logs/system.log
```

Using these commands in a script

All of these commands can be used inside of your scripts and can make for a very powerful toolset for developing programs in unix. For an example of using these commands in a script, let's write a script that uses our current data file. The script will get all users that are in the file and will then display how many files that user accessed. We will also have the script get all files in the file and display how many times each file was accessed. There are much more efficient ways of doing these specific functions, but for this example we will do it to better show how you can use these commands in a script.

```
#!/bin/sh

# First let's get the list of unique users:
USERS=`awk '{print $1}' testfile | sort -u`

echo "Users:"

# Now loop through each user.
for USER in $USERS
do
        # Get the number of lines that start with the user name.
        NUM=`grep -c "^$USER" testfile`

        echo " - $USER: $NUM files accessed."
done

# Now let's get the list of unique files:
FILES=`awk '{print $4}' testfile | sort -u`

echo ""
echo "Files:"

# And loop through each file.
for FILE in $FILES
do
        # Get the number of lines that end with the file name.
        NUM=`grep -c "$FILE$" testfile`

        echo " - $FILE: $NUM accesses."
done
```

Notice that we use the command line parameter **-c** for grep, this returns a row count instead of a list of rows. Another thing to notice is that we are reading the whole file 2 + (num_users * 2) times. In our case, that's 8 times. A smarter program would be able to read the file once and get the data it needs to parse out, because I/O calls on a system (such as reading a file) are always slower than reading from memory (variables).

Now let's save the script as test.sh and run it. We get the following output:

```
$ ./test.sh
Users:
 - root: 4 files accessed.
 - user1: 3 files accessed.
 - user2: 3 files accessed.

Files:
 - /usr/local/bin/one_app: 2 accesses.
 - /usr/local/bin/two_app: 3 accesses.
 - /usr/local/src/file.c: 1 accesses.
 - /var/logs/approot.log: 2 accesses.
 - /var/logs/system.log: 2 accesses.
```

Advanced Topics

Substrings

Often times a programmer needs to be able to get a substring from a variable at a given position. In unix you can use the **expr** command to do this with the **substr** parameter. Let's say that we have the text string "5283username$$2384/" and we want to get the text "username". To do this we need to read from position 5 for a length of 8. The parameters for **substr** are the input string, the starting position, and the length. See the following example:

```
#!/bin/sh

INPUT="5283username$$2384/"

USER=`expr substr $INPUT 5 8`

echo "Sub: '$USER'"
```

Find in a string

Sometimes you need to find text in a string. Maybe you want to list files but print only the text appearing before the ".". So if the filename is asdf.txt, you would want to print only asdf. To do this, you will use **expr index**, and pass it the string followed by the text for which you are searching. Let's try an example:

```
#!/bin/sh

# Get the files:
FILES=`ls -1`

for FILE in $FILES
do
        IDX=`expr index $FILE .`

        if [ "$IDX" == 0 ]; then
                IDX=`expr length $FILE`
        else
                IDX=`expr $IDX - 1`
        fi

        SUB=`expr substr $FILE 1 $IDX`
        echo "Sub File: $SUB"
done
```

If the substring doesn't exist, 0 is returned. If 0 is returned, we want to make the IDX variable the length of the name so that we just display the whole filename. If a dot is found in the file, we want to subtract 1 from our **$IDX** variable because we don't want to display the dot.

To lower/upper case

If you want to transform a string to upper or lower case, you can do so with the unix **tr** command. Here's a simple example.

```
#!/bin/sh

STR_ORIGINAL=aBcDeFgHiJkLmNoP
STR_UPPER=`echo $STR_ORIGINAL | tr a-z A-Z`
STR_LOWER=`echo $STR_ORIGINAL | tr A-Z a-z`

echo "Original: $STR_ORIGINAL"
echo "Upper   : $STR_UPPER"
echo "Lower   : $STR_LOWER"
```

Editing a file with sed

If you want to edit a file from within your script, you can use the unix **sed** command. It will take a regular expression and a filename and put the file manipulations to standard output. For instance, let's say that we have a file with two fields "username" and "home directory". All the home directories start with "/home", but what if the admin changes the location of the "/home" directory to "/usr/local/home". We can have sed automatically update our file. Here is the file, save it as testfile2.

```
user1 /home/user1
root /home/root
user2 /home/user2
user3 /home/user3
```

We want our regular expression to search for "/home" and replace it with "/usr/local/home", a search expression is in the following format: "s/find/replace/", where "find" is the string you are searching for and "replace" is what you want to replace it with. Since the / character is a special character, we will need to escape it with a backslash in our find and replace strings. Here is the command we will use to do the file edit:

```
$ sed "s/\/home/\/usr\/local\/home/" testfile2 > tmp; cp tmp testfile2
$ cat testfile2
user1 /usr/local/home/user1
root /usr/local/home/root
user2 /usr/local/home/user2
user3 /usr/local/home/user3
```

Notice that we redirect the output of sed to a file named **tmp**, we then on the same line copy the tmp file over the testfile2 file. We cannot specify testfile2 to be the output since it is also being read from by sed during the command. On the next line we **cat** the output and you can see the file modifications.

Automating another application

Sometimes we may want to automate another program or script. If the other script expects user input, we may want to write a script to automatically fill in that information. First let's create a simple program that accepts a user name and password:

```
#!/bin/sh

#Grab user name:
echo "user: "
read USER

#Grab password:
echo "pass: "
read PWD

if [ "$USER" == "dreamsys" ] && [ "$PWD" == "soft" ]; then
        echo "Login Success!"
else
        echo "Login Failed!"
fi
```

Save this file as **up.sh**. Now we need to create a script to automate this script. To do this, all we need to do is output the user name followed by the password to the command line, we will pass these as two parameters:

```
#!/bin/sh
USER=$1
PWD=$2

echo $USER
echo $PWD
```

Now to run this automation script, we simply need to pipe the output to the **up.sh** script. First we will try to run it with an invalid user and password, then we will try to run it with the correct user and password:

```
$ ./auto.sh testuser testpass | ./up.sh
user:
pass:
Login Failed!
$ ./auto.sh dreamsys soft | ./up.sh
user:
pass:
Login Success!
```