Nagarjuna College of Engineering and Technology

(An Autonomous Institution under VTU, Accredited by NAAC "A+" Grade)

Beedaganahalli, Venkatagiri Kote Post, Bengaluru – 562103

NAGARJUNA COLLEGE OF ENGINEERING AND TECHNOLOGY
DEPARTMENT OF INFORMATION SCIENCE AND ENGINEERING
SE

Department of Information Science and Engineering

III Semester – 2023-24

Unix Shell Programming (22ISL38D)

Laboratory Manual

Introduction to Shell scripting:

- Use of Basic UNIX Shell Commands and options related to them:

  vi, ls, mkdir, rmdir, cd, cat, touch, file, wc, sort, cut, who, man etc.
- Commands related to inode, I/O redirection and piping.
- Shell Programming: Shell script exercises based on following:

  (i) Interactive shell scripts

  (ii) Positional parameters

  (iii) Arithmetic

  (iv) if-then-fi, if-then- else-fi, nested if-else

  (v) Logical operators

  (vi) else + if equals elif, case structure

  (vii) while, until, for loops, use of break

Laboratory Programs:

1. Write a shell script to check whether the entered username and password is valid or not.

In this script, we define a valid username and password using the variables `valid_username` and `valid_password`. Then, we prompt the user to enter their username and password using the `read` command. The `-s` option is used with `read` for password input, which hides the user's input on the terminal. After reading the input, we compare the entered username and password with the valid credentials using an `if` statement. If the entered username and password match the valid ones, it displays a message indicating that the credentials are valid. Otherwise, it displays a message indicating that the credentials are invalid.

You can save this script in a file, for example, `check_credentials.sh`, make it executable using the `chmod +x check_credentials.sh` command, and then run it using `./check_credentials.sh` in the terminal.

```
#!/bin/bash

# Define valid username and password

valid_username="myusername"

valid_password="mypassword"
```

```
# Read username from user
read -p "Enter username: " username
# Read password from user without displaying input
read -s -p "Enter password: " password
echo
# Check if the entered username and password are valid
if [ "$username" == "$valid_username" ]&& [ "$password" == "$valid_password" ]; then
echo "Valid username and password."
else
echo "Invalid username or password."
fi
```

2. Write a shell script to add, subtract, multiply, divide two numbers and add two strings.

In this script, we define several functions to perform the desired operations. The `add_numbers`, `subtract_numbers`, `multiply_numbers`, and `divide_numbers` functions take two numbers as arguments and perform addition, subtraction, multiplication, and division, respectively. The `concatenate_strings` function takes two strings as arguments and concatenates them.

We then read two numbers and two strings from the user using the `read` command. After that, we call the respective functions to perform the arithmetic operations and string concatenation.

You can save this script in a file, for example, `arithmetic_operations.sh`, make it executable using the `chmod +x arithmetic_operations.sh` command, and then run it using `./arithmetic_operations.sh` in the terminal.

```
#!/bin/sh
a=10
b=20
x=”hello”
y=”world”
echo “a+b: $(($a+$b))”
echo “a-b: $(($a-b))”
echo “a%b: $(($a%$b))”
echo "a/b: $(($a/b))"
c=$x$y
echo $c
if ($a==$b)
then
        echo "a is equal to b"
else
        echo "a is not equal to b"
fi
```

3. Write a linux shell program to perform convert lowercase to uppercase using tr statement.

Certainly! Here's an example of a Linux shell program that converts lowercase characters to uppercase using the `tr` command:

```bash

    var_name="THIS IS a TEST"
    $ echo "$VAR_NAME" | tr '[:upper:]''[:lower:]'
    $ movie="The Matrix"
    $ echo "$movie" | tr '[:lower:]''[:upper:]'
```

In this script, we use the `read` command to prompt the user to enter a string in lowercase. The input is stored in the `input_string` variable.

Then, we use the `echo` command to pass the `input_string` to the `tr` command. The `tr` command is used to translate or delete characters. In this case, we use the character class `[:lower:]` to represent all lowercase characters and `[:upper:]` to represent all uppercase characters. The `tr` command replaces all lowercase characters in the input string with their corresponding uppercase characters.

The resulting uppercase string is stored in the `uppercase_string` variable.

Finally, we display the uppercase string using the `echo` command.

You can save this script in a file, for example, `convert_lowercase_to_uppercase.sh`, make it executable using the `chmod +x convert_lowercase_to_uppercase.sh` command, and then run it using `./convert_lowercase_to_uppercase.sh` in the terminal.

4. Write a shell script to check the given file is a directory or not.

Certainly! Here's an example of a shell script that checks whether a given file is a directory or not:

~~~
```bash
#!/bin/bash

# Read file path from user
read -p "Enter a file path: " file_path

# Check if the file is a directory
if [ -d "$file_path" ]; then
echo "The file is a directory."
else
echo "The file is not a directory."
fi
```
~~~

In this script, we use the `read` command to prompt the user to enter a file path. The input is stored in the `file_path` variable.

Then, we use the `-d` flag with the square brackets `[ ]` to test whether the given file path represents a directory. The condition `[ -d "$file_path" ]` checks if the file path exists and is a directory. If the condition is true, it means the file is a directory, and the script displays a corresponding message. If the condition is false, it means the file is not a directory, and the script displays a different message.

You can save this script in a file, for example, `check_directory.sh`, make it executable using the `chmod +x check_directory.sh` command, and then run it using `./check_directory.sh` in the terminal.

5. Write a shell script to perform check pattern matching using grep command.

Certainly! Here's an example of a shell script that performs pattern matching using the `grep` command:

```bash
#!/bin/bash

# Read input string from user
read -p "Enter a string: " input_string

# Perform pattern matching using grep
if echo "$input_string" | grep -q "pattern"; then
echo "Pattern found in the input string."
else
echo "Pattern not found in the input string."
fi
```

In this script, we use the `read` command to prompt the user to enter a string. The input is stored in the `input_string` variable.

Then, we use the `grep` command to perform pattern matching on the input string. The `-q` option is used to make `grep` silent and only return an exit status. We provide the pattern to match as an argument to `grep`. In this example, "pattern" is used as a placeholder. You can replace it with your desired pattern.

If the pattern is found in the input string, `grep` returns a zero exit status, and the script displays a message indicating that the pattern is found. If the pattern is not found, `grep` returns a non-zero exit status, and the script displays a message indicating that the pattern is not found.

You can save this script in a file, for example, `pattern_matching.sh`, make it executable using the `chmod +x pattern_matching.sh` command, and then run it using `./pattern_matching.sh` in the terminal.

6. Write a non-recursive shell script that accepts any number of arguments and prints them in a reverse order.

Certainly! Here's an example of a non-recursive shell script that accepts any number of arguments and prints them in reverse order:

```bash
#!/bin/bash

# Store all arguments in an array
args=("$@")

# Get the number of arguments
num_args=$#

# Print arguments in reverse order
for (( i=num_args-1; i>=0; i-- )); do
echo "${args[$i]}"
done
```

In this script, we use an array `args` to store all the arguments passed to the script using `"$@"`. The `"$@"` notation represents all the command-line arguments as separate elements in the array.

We get the number of arguments using `$#`, which represents the count of arguments.

Then, we use a `for` loop to iterate over the array in reverse order. Starting from the last index (`num_args-1`), we decrement `i` until it reaches 0. Inside the loop, we use `${args[$i]}` to access and print the argument at index `$i`.

The loop effectively prints the arguments in reverse order.

You can save this script in a file, for example, `reverse_arguments.sh`, make it executable using the `chmod +x reverse_arguments.sh` command, and then run it using `./reverse_arguments.sh arg1 arg2 arg3 ...` in the terminal, providing any number of arguments you want to reverse and print.

7. Write a shell script to check whether the given year is Leap year or not.

Certainly! Here's an example of a shell script that checks whether a given year is a leap year or not:

```bash
#!/bin/bash

# Read year from user
read -p "Enter a year: " year

# Check if the year is a leap year
if (( year % 4 == 0 && year % 100 != 0 )) || (( year % 400 == 0 )); then
echo "The year $year is a leap year."
else
echo "The year $year is not a leap year."
fi
```

In this script, we use the `read` command to prompt the user to enter a year. The input is stored in the `year` variable.

We then check whether the given year is a leap year or not using an `if` statement. The condition in the `if` statement checks two conditions for a leap year:

1. The year is divisible by 4 but not divisible by 100, which is represented as `(( year % 4 == 0 && year % 100 != 0 ))`.
2. Alternatively, the year is divisible by 400, which is represented as `(( year % 400 == 0 ))`.

If any of the conditions are true, the year is considered a leap year, and the script displays a corresponding message. If both conditions are false, the year is not a leap year, and the script displays a different message.

You can save this script in a file, for example, `leap_year_check.sh`, make it executable using the `chmod +x leap_year_check.sh` command, and then run it using `./leap_year_check.sh` in the terminal

8. Write a shell script to compute GCD & LCM of two numbers.

Certainly! Here's an example of a shell script that computes the GCD (Greatest Common Divisor) and LCM (Least Common Multiple) of two numbers, as well as a basic function to find the GCD and LCM of multiple numbers:

```
echo "Enter first  nuumber: "
read n1
echo "Enter second  nuumber: "
read n2
m=$n1
n=$n2
r=$n2
```

```
while [ $r -ne 0  ];do

r=$(( n1%n2 ))

if [ $r -eq 0 ];then

break

else

((n1=$n2))

((n2=$r))

fi

done

echo  "GCD of %d and %d is %d \n" $m $n $n2

echo  "LCM Of %d and %d is %d \n" $m $n $((($m*$n)/$n2))
```

```
ouput:

Enter first  nuumber:

60

Enter second  nuumber:

24

GCD of %d and %d is %d \n 60 24 12

LCM Of %d and %d is %d \n 60 24 120
```

In this script:

- The `gcd()` function computes the GCD of two numbers using the Euclidean algorithm. It takes two numbers as arguments and iteratively finds the remainder until the remainder becomes zero, at which point it returns the last non-zero remainder.

- The `lcm()` function computes the LCM of two numbers using the GCD. It takes two numbers as arguments and uses the formula `LCM(a, b) = (a * b) / GCD(a, b)` to compute the LCM.

- The script computes the GCD and LCM of two numbers (`num1` and `num2`) using the `gcd()` and `lcm()` functions and displays the results.

- The `gcd_multiple()` function computes the GCD of multiple numbers. It takes an array of numbers as arguments and iteratively calls the `gcd()` function to compute the GCD of all the numbers.

- The `lcm_multiple()` function computes the LCM of multiple numbers. It takes an array of numbers as arguments and iteratively calls the `lcm()` function to compute the LCM of all the numbers.

- The script computes the GCD and LCM of multiple numbers (`numbers`) using the `gcd_multiple()` and `lcm_multiple()` functions and displays the results.

You can save this script in a file, for example, `gcd_l

9. Write a shell script to find whether a given number is prime.

Certainly! Checking whether a large number is prime can be computationally intensive. Here's an example of a shell script that uses a common algorithm called the "Trial Division" to determine whether a given number is prime:

~~~
```bash
#!/bin/bash


# Function to check if a number is prime
is_prime() {
number=$1
if ((number < 2)); then
return 1
fi


# Check divisibility up to the square root of the number
for ((i = 2; i * i<= number; i++)); do
if ((number % i == 0)); then
return 1
~~~

```
fi
done

return 0
}

# Read the number from the user
read -p "Enter a number: " input_number

# Check if the number is prime
if is_prime "$input_number"; then
echo "$input_number is a prime number."
else
echo "$input_number is not a prime number."
fi
```

```

In this script:

- The `is_prime()` function checks whether a given number is prime or not. It takes the number as an argument and uses the trial division method.

- The function checks if the number is less than 2, in which case it is not prime.

- It then iterates from 2 to the square root of the number and checks if the number is divisible by any of these values. If it is divisible, the number is not prime.

- If the number passes all the checks, it is considered prime.

You can save this script in a file, for example, `check_prime.sh`, make it executable using the `chmod +x check_prime.sh` command, and then run it using `./check_prime.sh` in the terminal. The script will prompt you to enter a number, and it will determine whether the number is prime or not based on the trial division algorithm.
```

10. Write a shell script to check whether the given string is palindrome or not.

Certainly! Here's an example of a shell script that checks whether a given string is a palindrome or not:

```bash
#!/bin/bash

# Read string from user
read -p "Enter a string: " input_string

# Remove non-alphanumeric characters and convert to lowercase
clean_string=$(echo "$input_string" | tr -dc '[:alnum:]' | tr '[:upper:]''[:lower:]')

# Reverse the string
reverse_string=$(echo "$clean_string" | rev)

# Check if the string is a palindrome
if [ "$clean_string" = "$reverse_string" ]; then
echo "The string '$input_string' is a palindrome."
else
echo "The string '$input_string' is not a palindrome."
fi
```

In this script:

- We use the `read` command to prompt the user to enter a string. The input is stored in the `input_string` variable.

- We clean the string by removing non-alphanumeric characters using the `tr` command with the `-dc '[:alnum:]'` option. We also convert the string to lowercase using the `tr` command with the `'[:upper:]''[:lower:]'` option.

- We reverse the string using the `rev` command.

- Finally, we check if the cleaned string is equal to the reversed string. If they are equal, the script displays a message indicating that the string is a palindrome. Otherwise, it displays a message indicating that the string is not a palindrome.

You can save this script in a file, for example, `check_palindrome.sh`, make it executable using the `chmod +x check_palindrome.sh` command, and then run it using `./check_palindrome.sh` in the terminal.

________________________________ $ $ $ ______________________________________