CSCI 330
The UNIX System

Shell Programming Part 2

bash Control Structures

- if-then-else
- case
- loops
  - for
  - while
  - until
  - select

if statement

```
if command
then
   statements
fi
```

(not an expression)

- statements are executed only if **`command`** succeeds, i.e. has return status “0”

test command

Syntax:

```
test expression
[ expression ]
```

- evaluates ‘expression’ and returns true or false

Example:

```
if test $name = "Joe"          (note spaces)
then
  echo "Hello Joe"
fi
```

The simple if statement

```
if [ condition ]; then
   statements
fi
```

- executes the statements only if **`condition`** is true
- Notice ; after] Needed when “then” is on same line

The if-then-else statement

```
if [ condition ]; then
     statements-1
else
  statements-2
fi
```

- executes statements-1 if condition is true
- executes statements-2 if condition is false

The if…statement

```
if [ condition ]; then
     statements
elif [ condition ]; then
    statements
else
    statements
fi
```

- **elif** stands for “else if” : it is part of the if statement and cannot be used by itself

Relational Operators

Meaning	Numeric	String
Greater than	-gt	
Greater than or equal	-ge	
Less than	-lt	
Less than or equal	-le	
Equal	-eq	=
Not equal	-ne	!=
String length is zero		-z str
String length is non-zero		-n str
file1 is newer than file2		file1 -nt file2
file1 is older than file2		file1 -ot file2

Compound logical expressions

`! expression`

true if expression is false

`expression –a expression`

true if both expressions are true

`expression –o expression`

true if one of the expressions is true

also:

`&&`	and
`||`	or

and, or must be enclosed within

[[ ]]

Example: Using the ! Operator

```
#!/bin/bash

read -p "Enter years of work: " Years
if [ ! "$Years" -lt 20 ]; then
   echo "You can retire now."
else
   echo "You need 20+ years to retire"
fi
```

(Why quotes around **$Years** ?)

Example: Using the && Operator

```
#!/bin/bash

Bonus=500
read -p "Enter Status: " Status
read -p "Enter Shift: " Shift
if [[ "$Status" = "H" && "$Shift" = 3 ]]
then
   echo "shift $Shift gets \$$Bonus bonus"
else
   echo "only hourly workers in"
   echo "shift 3 get a bonus"
fi
```

Example: Using the || Operator

```
#!/bin/bash

read -p "Enter calls handled:" CHandle
read -p "Enter calls closed: " CClose
if [[ "$CHandle" -gt 150 || "$CClose" -gt 50 ]]
   then
   echo "You are entitled to a bonus"
else
   echo "You get a bonus if the calls"
   echo "handled exceeds 150 or"
   echo "calls closed exceeds 50"
fi
```

File Testing operators

	Meaning
-d file	True if ‘file’ is a directory
-f file	True if ‘file’ is a regular file
-r file	True if ‘file’ is readable
-w file	True if ‘file’ is writable
-x file	True if ‘file’ is executable
-s file	True if length of ‘file’ is nonzero

Example: File Testing

```
#!/bin/bash
read –p "Enter a filename: " filename
if [ ! –r "$filename" ]; then
   echo "File is not read-able"
   exit 1
fi
```

Example: File Testing

```
#!/bin/bash

if [ $# -lt 1 ]; then
        echo "Usage: filetest filename"
        exit 1
fi
if [[ ! -f "$1" || ! -r "$1" || ! -w "$1" ]]
then
  echo "File $1 is not accessible"
  exit 1
fi
```

Example: if..elif... Statement

```
#!/bin/bash

read -p "Enter Income Amount: " Income
read -p "Enter Expenses Amount: " Expense

Net=$(($Income-$Expense))

if [ "$Net" -eq "0" ]; then
   echo "Income and Expenses breakeven"
elif [ "$Net" -gt "0" ]; then
   echo "Profit of: " $Net
else
   echo "Loss of: " $Net
fi
```

The case Statement

- use the case statement for a decision that is based on multiple choices

Syntax:

```
case word in
   pattern1) command-list1
   ;;
   pattern2) command-list2
   ;;
   patternN) command-listN
   ;;
esac
```

case pattern

- checked against word for match
- may also contain: (wildcards)

  `*`

  `?`

  `[ … ]`

  `[:class:]`

- multiple patterns can be listed via:

  `|`

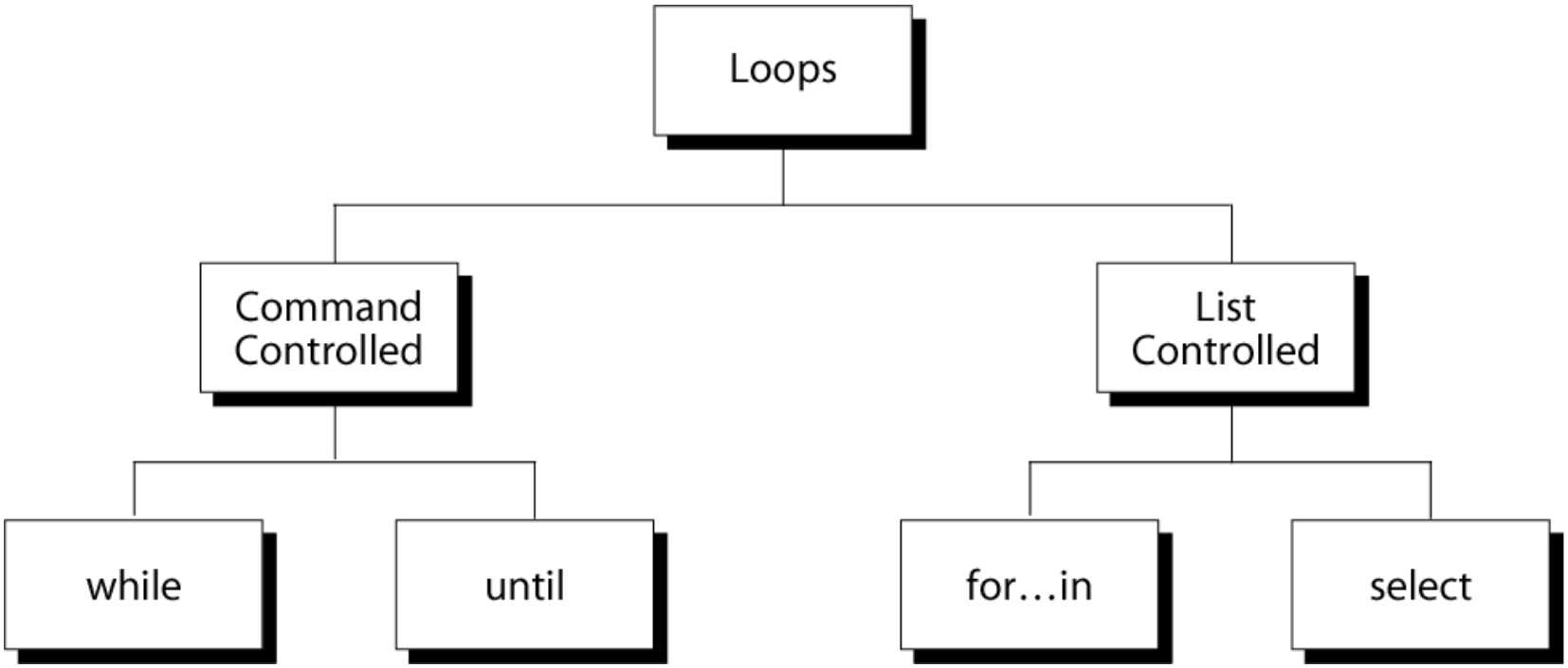
Example: case Statement

```
#!/bin/bash
echo "Enter Y to see all files including hidden files"
echo "Enter N to see all non-hidden files"
echo "Enter Q to quit"

read -p "Enter your choice: " reply

case "$reply" in
  Y|YES) echo "Displaying all (really…) files"
         ls -a ;;
  N|NO)  echo "Display all non-hidden files..."
         ls ;;
  Q)     exit 0 ;;

  *) echo "Invalid choice!"; exit 1 ;;
esac
```

Example: case Statement

```
#!/bin/bash
ChildRate=3
AdultRate=10
SeniorRate=7
read -p "Enter your age: " age
case "$age" in
  [1-9]|[1][0-2])   # child, if age 12 and younger
     echo "your rate is" '$'"$ChildRate.00" ;;
    # adult, if age is between 13 and 59 inclusive
  [1][3-9]|[2-5][0-9])
     echo "your rate is" '$'"$AdultRate.00" ;;
  [6-9][0-9])       # senior, if age is 60+
     echo "your rate is" '$'"$SeniorRate.00" ;;
esac
```

Repetition Constructs



The while Loop

- Purpose:

  execute “command-list” as long as

  “test-command” evaluates successfully

Syntax:

```
while test-command
do
   command-list
done
```

Example: Using the while Loop

```
#!/bin/bash
COUNTER=0
while [ $COUNTER -lt 10 ]
do
    echo The counter is $COUNTER
    COUNTER=$(( $COUNTER+1 ))
done
```

Example: Using the while Loop

```
#!/bin/bash
# script shows user’s active processes
cont="y"
while [ $cont = "y" ]; do
  ps -fu $USER
  read -p "again (y/n)? " cont
done
echo "done"
```

Example: Using the while Loop

```
#!/bin/bash
# copies files from home- into the webserver- directory
# A new directory is created every hour

PICSDIR=/home/carol/pics
WEBDIR=/var/www/carol/webcam
while true; do
   DATE=`date +%Y%m%d`
   HOUR=`date +%H`
   mkdir $WEBDIR/$DATE
   while [ $HOUR -ne "00" ]; do
      mkdir $WEBDIR/$DATE/$HOUR
      mv $PICSDIR/*.jpg $WEBDIR/$DATE/$HOUR
      sleep 3600
      HOUR=`date +%H`
   done
done
```

The until Loop

- Purpose:

  execute “command-list” as long as

  “test-command” does not evaluate successfully

Syntax:

```
until test-command
do
   command-list
done
```

Example: Using the until Loop

```
#!/bin/bash

COUNTER=20
until [ $COUNTER -lt 10 ]
do
    echo $COUNTER
    COUNTER=$(( COUNTER -1 ))
done
```

Example: Using the until Loop

```
#!/bin/bash
# script shows user’s active processes
stop="n"
until [ $stop = "y" ]; do
  ps -fu $USER
  read -p "done (y/n)? " stop
done
echo "done"
```

The for Loop

- Purpose:

  execute “commands” as many times as the number of words in the “word-list”

Syntax:

```
for variable in word-list
do
   commands
done
```

Example 1: for Loop

```
#!/bin/bash

for index in 7 9 2 3 4 5
do
    echo $index
done
```

Example 2: Using the for Loop

```
#!/bin/bash
# compute average weekly temperature
TempTotal=0
for num in 1 2 3 4 5 6 7
do
   read -p "Enter temp for $num: " Temp
   TempTotal=$(($TempTotal+$Temp))
done
AvgTemp=$(($TempTotal/7))
echo "Average temperature: " $AvgTemp
```

Example 3: Using the for Loop

```
#!/bin/bash
# compute average weekly temperature
TempTotal=0
for day in Mon Tue Wed Thu Fri Sat Sun
do
   read -p "Enter temp for $day: " Temp
   TempTotal=$(($TempTotal+$Temp))
done
AvgTemp=$(($TempTotal/7))
echo "Average temperature: " $AvgTemp
```

Example 4: Using the for Loop

```
#!/bin/bash
# compute average weekly temperature
TempTotal=0
for day in `cat day-file`
do
   read -p "Enter temp for $day: " Temp
   TempTotal=$(($TempTotal+$Temp))
done
AvgTemp=$(($TempTotal/7))
echo "Average temperature: " $AvgTemp
```

looping over arguments

- simplest form will iterate over all command line arguments:

```
#! /bin/bash
for parm
do
        echo $parm
done
```

break and continue

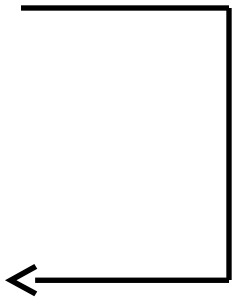
- Interrupt for, while or until loop

- The break statement
  - terminate execution of the loop
  - transfers control to the statement AFTER the done statement

- The continue statement
  - skips the rest of the current iteration
  - continues execution of the loop

The break command

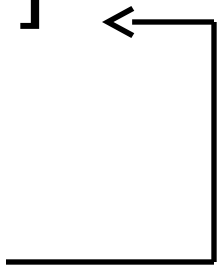
```
while [ condition ]
do
    cmd-1
    break
    cmd-n
done
echo "done"
```



This iteration is over and there are no more iterations

The continue command

```
while [ condition ]
do
     cmd-1
     continue
     cmd-n
done
echo "done"
```



This iteration is over; do the next iteration

Example:

```
for index in 1 2 3 4 5 6 7 8 9 10
do
        if [ $index –le 3 ]; then
              echo "continue"
              continue
        fi
        echo $index
        if  [ $index –ge 8 ]; then
              echo "break"
              break
        fi
done
```

Shell Functions

- A shell function is similar to a shell script
  - stores a series of commands for execution later
  - shell executes a shell function in the same shell that called it
- Function features:
  - parameters
  - local variables
  - return command to set return status
- Remove a function
  - Use unset built-in

Example: function

```
#!/bin/bash

funky () {
   # This is a simple function
   echo "This is a funky function."
   echo "Now exiting funky function."
}

# declaration must precede call:

funky
```

Function parameters

- Need not be declared
- Arguments provided via function call are accessible inside function as $1, $2, $3, ...

$#	reflects number of parameters
$0	still contains name of script (not name of function)

Example: function with parameter

```
#!/bin/bash
testfile() {
  if [ $# -gt 0 ]; then
     if [[ -f $1 && -r $1 ]]; then
        echo $1 is a readable file
     else
        echo $1 is not a readable file
     fi
  fi
}

testfile .
```

Example: function with parameters

```
#!/bin/bash
checkfile() {
   for file
   do
      if [ -f "$file" ]; then
         echo "$file is a file"
      else
         if [ -d "$file" ]; then
            echo "$file is a directory"
         fi
      fi
   done
}
checkfile . funtest
```

Local Variables in Functions

- Variables defined within functions are global,

  i.e. their values are known throughout the entire shell program

- keyword “local” inside a function definition makes referenced variables “local” to that function

Example: function

```
#!/bin/bash

funky () {
   # This is a simple function
   echo "This is a funky function."
   echo "Now exiting funky function."
}

# declaration must precede call:

funky
```

return from function

- special command:

  return [status]

- ends execution of function
- optional numeric argument sets

  return status $?

return example

```
#!/bin/bash
testfile() {
  if [ $# -gt 0 ]; then
     if [ ! -r $1 ]; then
        return 1
      fi
  fi
}

if testfile funtest; then
   echo "funtest is readable"
fi
```

Summary

- Decisions
- Repetition
- Functions