# Shell Programming and Unix Utilities

- Creating Shell Scripts
- Variables, Arguments, and Expressions
- Shell I/O and Debugging
- Testing Conditions and Control Statements
  - test, if, case, while, until, for, break, continue
- Exit Status
- Command Substitution
- Regular Expressions and Utilities
  - grep, wc, touch, awk, tr, sort, gtbl, groff, ghostview, cut, head, tail, sed, gnuplot

# Shell Scripts (35.2)

- Advantages of schell scripts.
  - Can very quickly setup a sequence of commands to avoid a repetitive task.
  - Can easily make several programs work together to meet a set of goals.
- Disadvantage of shell scripts.
  - Little support for programming in the large.
  - Shell scripts are much slower since they are interpreted.

# What Shell to Use? (27.3)

- The csh (C) shell and its derivative (tcsh) are recommended for use at the command line.
- The sh (Bourne) shell and its derivatives (ksh, bash) are recommended for writing shell scripts.
  - All shell script examples in this course are given using the Bourne shell syntax.

# Finding Information about the Bourne Shell and Its Commands

- Type “man <command>” or “info <command>” where <command> is the Bourne shell command of interest.
- Type “man sh” and look in the manual page.
- Look up information in the text or use other texts.
- See me or the TA.

# Creating a Shell Script (35.1, 36.2)

- General convention is to use a .sh extension (<name>.sh) for shell scripts.
- Type in the first line of each shell script file:

  #!/bin/sh

- Type in comments after this to indicate the purpose of the shell script.

  # This shell script is used to create a new
  # schema.

- Change the permissions on the schell script so it can be executed by typing the name of the file.

  chmod +x <name>.sh

# Printing a Line to Standard Output

- Use the echo command to print a line to stdout.

  form: echo <zero or more values>

  exs: echo “Hello World!”

  echo “Please enter a schema name:”

# Implementation of Echo

linprog4:~whalley/test/src

```
static char *sccsid = "@(#)echo.c       4.1 (Berkeley) 10/1/80";
#include <stdio.h>

main(argc, argv)
int argc;
char *argv[];
{
        register int i, nflg;

        nflg = 0;
        if(argc > 1 && argv[1][0] == '-' && argv[1][1] == 'n') {
                nflg++;
                argc--;
                argv++;
        }
        for(i=1; i<argc; i++) {
                fputs(argv[i], stdout);
                if (i < argc-1)
                        putchar(' ');
        }
        if(nflg == 0)
                putchar('\n');
        exit(0);
}
                                        1,1          All
```

# Shell Variables (35.9)

- Are just used and not declared.
- General convention is to use lowercase letters for shell variable names and uppercase letters for environment variable names.
- Shell variables are assigned strings, which can be interpreted as numeric values. Note that there can be no blanks before or after the '='.

  form: <name>=<value>

  exs: var=0

  var=”Good”

# Shell Variables (cont.)

- Shell variables are initialized to be the empty string by default.
- Shell variables can be dereferenced.

form: $<name>

exs: var1=$var2

echo “====” $testcase “====”

echo “deleted” $num “records from” $schema

# Reading Values into a Shell Variable (35.18)

- Use the read statement to read a line of standard input, split the line into fields of one or more strings, and assign these strings to shell variables. All leftover strings in the line are assigned to the last variable.

form: read <var1> <var2> ... <varm>

exs: read num

read name field1 field2

read name1 name2 < input.txt

# Shell Arguments (35.20)

- Can pass arguments on the command line to a shell script, just like you can pass command line arguments to a program.

  nd.sh employee Intel

- $1, $2, ..., $9 can be used to refer to up to nine command line arguments (argv[1] ... argv[9]).
- $0 contains the name of the script (argv[0]).

# Example Using Shell Arguments

- Example shell script called prompt.sh:

```
#!/bin/sh
echo “Please enter fields for the” $1 “schema.”
```

- Example usage:

  prompt.sh Intel

- Example output:

  Please enter fields for the Intel schema.

# More on Cmd Line Args (35.20)

- $# contains the number of command line arguments.
- $@ will be replaced with a string containing the command line arguments.
- Example shell script called echo.sh.

```
#!/bin/sh
echo "The" $# "arguments entered were:" $@
```

- Example usage:

```
echo.sh cat dog horse
```

- Example output:

```
The 3 arguments entered were: cat dog horse
```

# Debugging Shell Scripts (35.25, 37.1)

- Shell scripts are interpreted by the shell.
  - Syntax errors are not detected until that line is interpreted at run time.
  - There is no debugger.
- You can invoke your shell script with command line flags to assist in debugging.
  - The -v flag indicates the verbose option and the shell prints the commands as they are read.
  - The -x flag causes the shell to print the interpolated commands after they are read.

# Debugging Shell Scripts (cont.)

- Invoke your shell script as follows so that each line of the script is echoed as it is interpreted.

```
form:       sh -xv <scriptname>
ex usage:   sh -xv echo.sh a b c
ex output:  #!/bin/sh
            echo "The" $# "arguments entered were:" $@
            + echo The 3 arguments entered were: a b c
            The 3 arguments entered were: a b c
```

- You can also set these options in the shell script itself.
  - The “set -xv” will turn on these options.
  - The “set +xv” will turn off these options.

# Testing Conditions (35.26)

- Can test for various conditions. There are two general forms.

```
test <condition> or [ <condition> ]
```

- I think the latter is easier to read. Be sure to include a space before and after the brackets.
- To test the result of a command, just use the command, which will return an exit status.
- Can reverse a condition by putting '!' before it.

```
[ ! <condition> ]
```

- A ':' command always returns true.

# Testing File Attributes (35.26)

- Test if a file exists and is readable.

```
[  -r schemas.txt  ]
[  -r $1.db  ]
```

- Test if a file doesn't exist or is not writeable.

```
[  ! -w databases.txt  ]
```

- Test if a file exists and is executable.

```
[  -x ns.sh  ]
```

- Test if a file exists and is not a directory.

```
[  -f tmp.txt ]
```

# Numeric Tests

- The following {-eq, -ne, -gt, -ge, -lt, -le } operators can be used for numeric tests.

```
[  $1 -lt $2  ]
[  $1 -gt 0  ]
[  $1 -eq $# ]
```

# Exit (24.4, 35.12, 35.16)

- General form. The exit command causes the current shell script to terminate. There is an implicit exit at the end of each shell script. The status indicates the success of the script. If the status is not given, then the script will exit with the status of the last command.

```
exit
```

or

```
exit <status>
```

# Simple If Statement (35.13)

- General form.

```
if condition
then
        one-or-more-commands
fi
```

- Example.

```
if [ -r “tmp.txt” ]
then
        echo “tmp.txt is readable.”
fi
```

# Testing Strings

- Can perform string comparisons. It is good practice to put the shell variable being tested inside double quotes (“$v” instead of $v).

  [ “$1” = “yes” ]

- The following will give a syntax error when $1 is empty since:

  [ $1 != “yes” ]

- becomes

  [ != “yes” ]

# Testing Strings (cont.)

- Can check if a string is empty to avoid a syntax error.

```
[ -z $1 ]
```

- Can also check if a string is nonempty.

```
[ -n $1 ]
```

# Testing Strings (cont.)

- Can check if a string is empty to avoid a syntax error.

```
[ -z $1 ]
```

# Quoting Rules (27.12)

- 'xxx' disables all special characters in xxx.
- “xxx” disables all special characters in xxx except $, `, and \.
- \x disables the special meaning of character x.

# Quoting Examples

```
animal=”horse”
echo '$animal'          # prints:  $animal
echo “$animal”          # prints:  horse
cost=2000
echo 'cost: $cost'      # prints:  cost: $cost
echo “cost: $cost”      # prints:  cost: 2000
echo “cost: \$cost”     # prints:  cost: $cost
echo “cost: \$$cost”    # prints:  cost: $2000
```

# String Relational Operators

- The string relational operators are the operators:

  =, !=, >, >=, <, <=

- The (>, >=, <, <=) assume an ASCII ordering when performing comparisons. They have to be used with the expr command, which will be covered later. The backslash has to be placed before these operators so they are not confused with I/O redirection.

# Testing with Multiple Conditions (35.14)

- Can check if multiple conditions are all met.

  [ “$1” = “yes” ] && [ -r $2.txt ]

- Can check if one of a set of conditions are met.

  [ “$2” = “no” ] || [ ! -r $1.db ]

# General If Statement (35.13)

- General form. The “if condition”, initial “then”, and “fi” are required. Can have zero or more elif's. The else is also optional.

```
if  condition
then
    one-or-more-commands
elif condition
then
    one-or-more-commands
...
else
    one-or-more-commands
fi
```

# If Statement Examples

```
If  [ ! -r $1 ]
then
    echo “file” $1 “not readable”
fi

if  [  $1 = $2  ]
then
    echo $1 “is equal to” $2
else
    echo $1 “is not equal to” $2
fi
```

# If Statement Examples (cont.)

```
if  [ $var1 -lt $var2  ]
then
    echo  $var1 “is less than” $var2
elif  [ $var1 -gt $var2 ]
then
    echo  $var1 “is greater than” $var2
else
    echo $var1 “is equal to” $var2
fi
```

# Case Statement (35.10)

- General form.  Compares stringvalue to each of the strings in the patterns.  On first match it does the corresponding commands. ;; indicates to jump to the statement after esac. *) means the default case.

```
case stringvalue in
pattern1) one-or-more-commands ;;
pattern2) one-or-more-commands ;;
...
        *) one-or-more-commands ;;
esac
```

# Case Statement Example

```
echo “Please answer yes or no.”
read answer
case $answer in
“yes”) echo “We are pleased you agree.”
        ...
        ;;
 “no”) echo “We are sorry you disagree.”
        ...
        ;;
      *) echo “Please enter \”yes\” or \”no\”.”
esac
```