Lecture 6

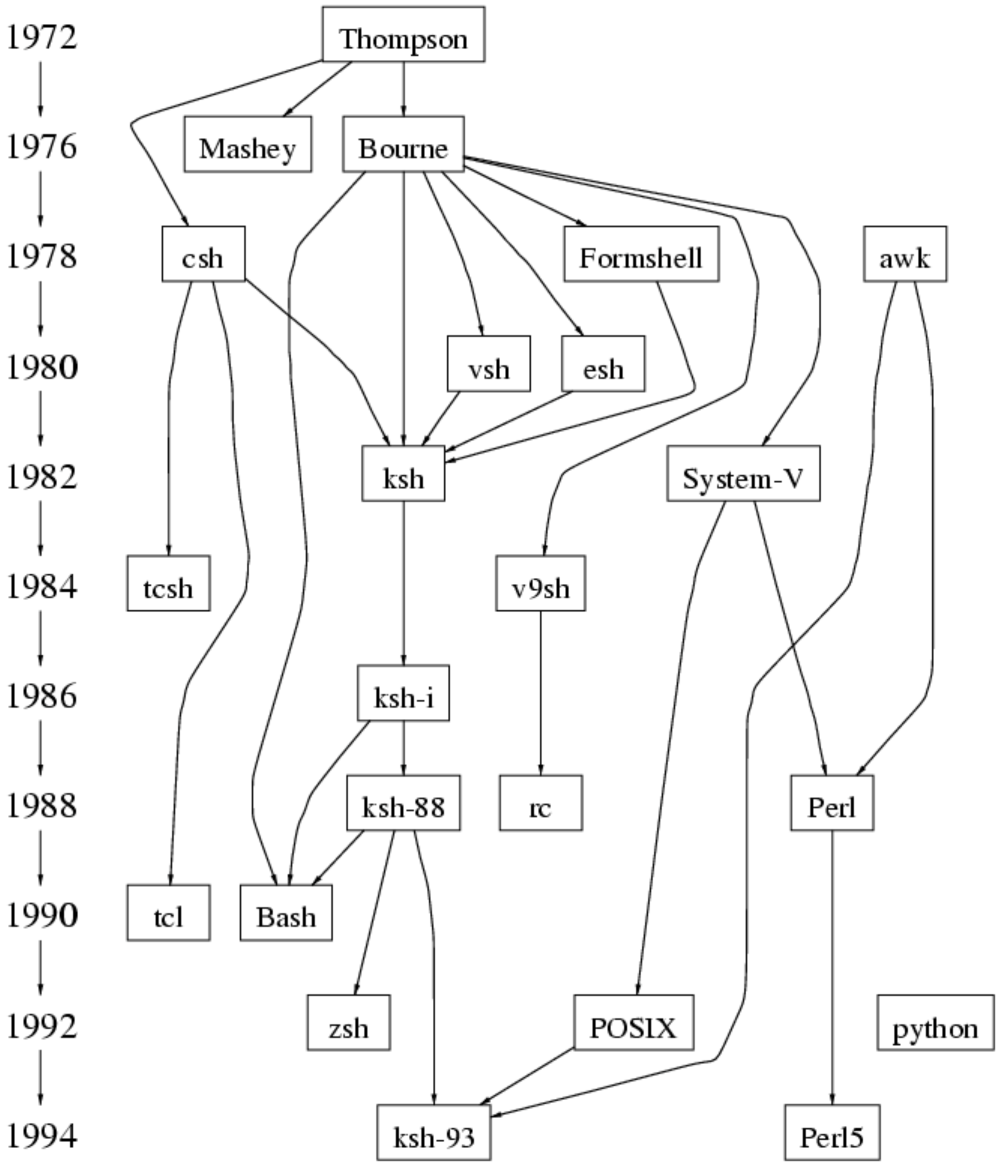
Shell Scripting

What is a shell?

- The user interface to the operating system
- Functionality:
  - Execute other programs
  - Manage files
  - Manage processes
- Full programming language
- A program like any other
  - This is why there are so many shells

Shell History

- There are many choices for shells
- Shell features evolved as UNIX grew



Most Commonly Used Shells

– /bin/csh	C shell
– /bin/tcsh	Enhanced C Shell

– /bin/sh	The Bourne Shell / POSIX shell
– /bin/ksh	Korn shell
– /bin/bash	Korn shell clone, from GNU

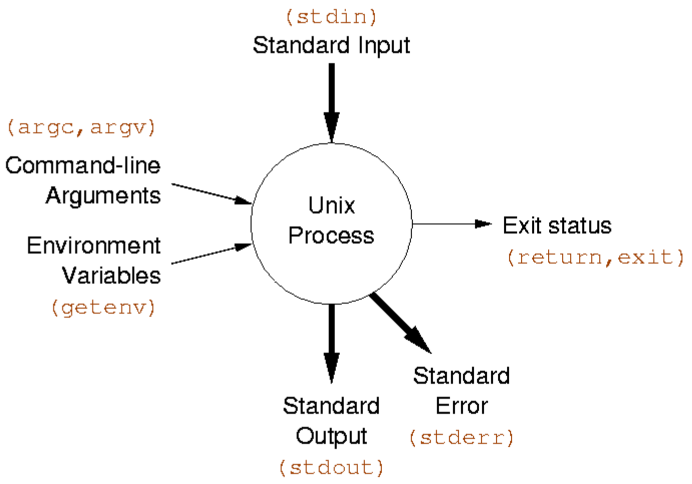
Ways to use the shell

- **Interactively**
  - When you log in, you interactively use the shell

- **Scripting**
  - A set of shell commands that constitute an executable *program*

Review: UNIX Programs

- **Means of input**:
  - Program arguments [control information]
  - Environment variables [state information]
  - Standard input [data]



- **Means of output**:
  - Return status code [control information]
  - Standard out [data]
  - Standard error [error messages]

Shell Scripts

- A shell script is a regular text file that contains shell or UNIX commands
  - Before running it, it must have execute permission:
    - **`chmod +x`** ***`filename`***
- A script can be invoked as:
  - **`ksh name`** ***`[ arg … ]`***
  - **`ksh < name`** ***`[ args … ]`***
  - **`name`** ***`[ arg …]`***

Shell Scripts

- When a script is run, the **kernel** determines which shell it is written for by examining the first line of the script
  - If 1st line starts with **`#!`*`pathname-of-shell`***, then it invokes *pathname* and sends the script as an argument to be interpreted
  - If **`#!`** is not specified, the current shell assumes it is a script in its own language
    - leads to problems

Simple Example

```
#!/bin/sh

echo Hello World
```

Scripting vs. C Programming

- Advantages of shell scripts
  - Easy to work with other programs
  - Easy to work with files
  - Easy to work with strings
  - Great for prototyping. No compilation
- Disadvantages of shell scripts
  - Slow
  - Not well suited for algorithms & data structures

The C Shell

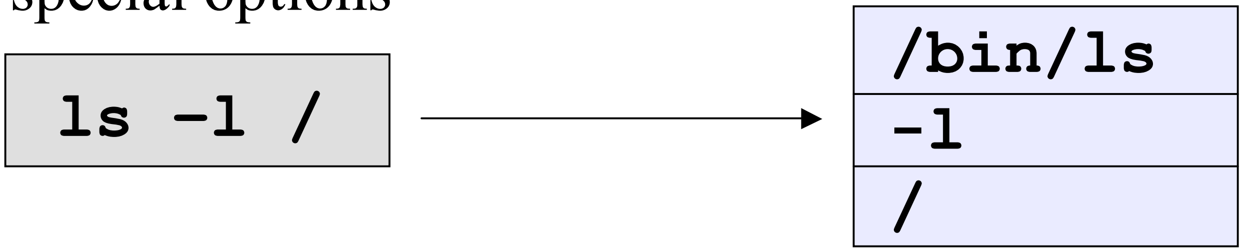
- C-like syntax (uses `{  }`'s)
- **Inadequate for scripting**
  - Poor control over file descriptors
  - Can't mix flow control and commands
  - Difficult quoting `"I say \"hello\""` doesn't work
  - Can only trap SIGINT
- Survives mostly because of interactive features.
  - Job control
  - Command history
  - Command line editing, with arrow keys (**tcsh**)

The Bourne Shell

- Slight differences on various systems
- Evolved into standardized POSIX shell
- Scripts will also run with **ksh**, **bash**
- Influenced by ALGOL

Simple Commands

- ***simple command***: sequence of non blanks arguments separated by blanks or tabs.
- 1st argument (numbered zero) usually specifies the name of the command to be executed.
- Any remaining arguments:
  - Are passed as arguments to that command.
  - Arguments may be filenames, pathnames, directories or special options



```
ls -l /
```

```
/bin/ls
-l
/
```

Background Commands

- Any command ending with "&" is run in the background.

```
netscape &
```

- **`wait`** will block until the command finishes

Complex Commands

- The shell's power is in its ability to hook commands together
- We've seen one example of this so far with pipelines:

```
cut –d: -f2 /etc/passwd | sort | uniq
```

- We will see others

Redirection of input/ouput

- Redirection of output: >
  - example: `$ ls -l > my_files`
- Redirection of input: `<`
  - example: `$ cat <input.data`
- Append output: >>
  - example: `$ date >> logfile`
- Arbitrary file descriptor redirection: *fd*>
  - example: `$ ls –l 2> error_log`

Multiple Redirection

- **`cmd 2>file`**
  - – send standard error to file
  - – standard output remains the same
- **`cmd > file 2>&1`**
  - – send both standard error and standard output to file
- **`cmd > file1 2>file2`**
  - – send standard output to file1
  - – send standard error to file2

Here Documents

- Shell provides alternative ways of supplying standard input to commands (an *anonymous file*)
- Shell allows in-line input redirection using << called here documents
- **format**

```
command [arg(s)] << arbitrary-delimiter
command input
 :
 :
arbitrary-delimiter
```

- `arbitrary-delimiter` should be a string that does not appear in text

Here Document Example

```
#!/bin/sh

mail steinbrenner@yankees.com <<EOT
  You guys really blew it
  Monday.  Good luck next year.
  Yours,
  $USER
EOT
```

Shell Variables

- Write
  ```
  name=value
  ```

- Read: `$var`

- Turn local variable into environment:
  ```
        export variable
  ```

Variable Example

```
#!/bin/sh

MESSAGE="Hello World"
echo $MESSAGE
```

Environmental Variables

NAME	MEANING
$HOME	Absolute pathname of your home directory
$PATH	A list of directories to search for
$MAIL	Absolute pathname to mailbox
$USER	Your login name
$SHELL	Absolute pathname of login shell
$TERM	Type of your terminal
$PS1	Prompt

Parameters

- A parameter is one of the following:
  - A variable
  - A *positional parameter*, starting at 1
  - A *special* parameter
- To get the value of a parameter: **`${param}`**
  - Can be part of a word (**`abc${foo}def`**)
  - Works within double quotes
- The **`{}`** can be omitted for simple variables, special parameters, and single digit positional parameters.

Positional Parameters

- The arguments to a shell script
  - `$1, $2, $3 …`
- The arguments to a shell function
- Arguments to the **set** built-in command
  - `set this is a test`
    - `$1=this, $2=is, $3=a, $4=test`
- Manipulated with **shift**
  - `shift 2`
    - `$1=a, $2=test`
- Parameter 0 is the name of the shell or the shell script.

Example with Parameters

```
#!/bin/sh

# Parameter 1: word
# Parameter 2: file
grep $1 $2 | wc -l
```

```
$ countlines ing /usr/dict/words
3277
```

Special Parameters

• **`$#`**	Number of positional parameters
• **`$-`**	Options currently in effect
• **`$?`**	Exit value of last executed command
• **`$$`**	Process number of current process
• **`$!`**	Process number of background process
• **`$*`**	All arguments on command line
• **`"$@"`**	All arguments on command line individually quoted **`"$1" "$2" ...`**

Command Substitution

- Used to turn the output of a command into a string
- Used to create arguments or variables
- Command is placed with grave accents `   ` to capture the output of command

```
$ date
Wed Sep 25 14:40:56 EDT 2001
$ NOW=`date`

$ sed "s/oldtext/`ls | head -1`/g"

$ PATH=`myscript`:$PATH
$ grep `generate_regexp` myfile.c
```

File name expansion

- Wildcards (patterns)
  - *   matches any string of characters
  - ?   matches any single character
  - [**list**] matches any character in **list**
  - **[lower-upper]** matches any character in range **lower-upper** inclusive
  - **[!list]** matches any character not in list

File Expansion

- If multiple matches, all are returned and treated as separate arguments:

```
$ /bin/ls
file1 file2
$ cat file1
a
$ cat file2
b
$ cat file*
a
b
```

- Handled by the shell (exec never sees the wildcards)
  - argv[0]: /bin/cat
  - argv[1]: file1
  - argv[2]: file2

NOT

  - argv[0]: /bin/cat
  - argv[1]: file*

Compound Commands

- Multiple commands
  - Separated by semicolon or newline
- Command groupings
  - pipelines
- Subshell

```
( command1; command2 ) > file
```

- Boolean operators
- Control structures

Boolean Operators

- Exit value of a program (**exit** system call) is a number
  - 0 means success
  - anything else is a failure code
- *cmd1* **`&&`** *cmd2*
  - executes cmd2 if cmd1 is successful
- *cmd1* **`||`** *cmd2*
  - executes cmd2 if cmd1 is not successful

```
$ ls bad_file > /dev/null && date
$ ls bad_file > /dev/null || date
Wed Sep 26 07:43:23 2001
```

Control Structures

```
if expression
then
   command1
else
   command2
fi
```

What is an expression?

- Any UNIX command. Evaluates to true if the exit code is 0, false if the exit code > 0
- Special command **`/bin/test`** exists that does most common expressions
  - String compare
  - Numeric comparison
  - Check file properties
- **`/bin/[`** often linked to **`/bin/test`** for syntactic sugar (or builtin to shell)
- Good example UNIX tools working together

Examples

```
if test "$USER" = ″mohri"
then
      echo "I know you"
else
      echo "I dont know you"
fi
```

```
if [ -f /tmp/stuff ] && [ `wc –l < /tmp/stuff` -gt 10
]
then
      echo "The file has more than 10 lines in it"
else
      echo "The file is nonexistent or small"
fi
```

test Summary

- **String based tests**

`-z string`	Length of string is 0
`-n string`	Length of string is not 0
`string1 = string2`	Strings are identical
`string1 != string2`	Strings differ
`string`	String is not NULL

- **Numeric tests**

`int1 -eq int2`	First int equal to second
`int1 -ne int2`	First int not equal to second
`-gt`, `-ge`, `-lt`, `-le`	greater, greater/equal, less, less/equal

- **File tests**

`-r file`	File exists and is readable
`-w file`	File exists and is writable
`-f file`	File is regular file
`-d file`	File is directory
`-s file`	file exists and is not empty

- **Logic**

`!`	Negate result of expression
`-a`, `-o`	and operator, or operator
`( expr )`	groups an expression

Arithmetic

- No arithmetic built in to **`/bin/sh`**
- Use external command **`/bin/expr`**
- **`expr expression`**
  - Evaluates expression and sends the result to standard output
  - Yields a numeric or string result

```
expr 4 "*" 12

expr "(" 4 + 3 ")" "*" 2
```

Control Structures Summary

- `if … then … fi`
- `while … done`
- `until … do … done`
- `for … do … done`
- `case … in … esac`

for loops

- Different than C:

```
for var in list
do
 command
done
```

- Typically used with positional params or a list of files:

```
sum=0
for var in "$@"
do
  sum=`expr $sum + $var`
done
echo The sum is $sum

for file in *.c ; do echo "We have $file"
done
```

Case statement

- Like a C switch statement for strings:

```
case $var in
  opt1) command1
        command2
        ;;
  opt2) command
        ;;
  *)    command
        ;;
 esac
```

- ***** is a catch all condition

Case Example

```
#!/bin/sh

echo "Say something."
while true
do
    read INPUT_STRING
    case $INPUT_STRING in
        hello)
            echo "Hello there."
            ;;
        bye)
            echo "See ya later."
            ;;
        *)
            echo "I'm sorry?"
            ;;
    esac
done
echo "Take care."
```

Case Options

- **opt** can be a shell pattern, or a list of shell patterns delimited by `|`
- Example:

```
case $name in
    *[0-9]*)
        echo "That doesn't seem like a name."
        ;;
    J*|K*)
        echo "Your name starts with J or K, cool."
        ;;
    *)
        echo "You're not special."
        ;;
esac
```

Types of Commands

All behave the same way

- Programs
  - Most that are part of the OS in /bin
- Built-in commands
- Functions
- Aliases

Built-in Commands

- Built-in commands are internal to the shell and do not create a separate process. Commands are built-in because:
  - They are intrinsic to the language (**exit**)
  - They produce side effects on the process (**cd**)
  - They perform much better
    - No fork/exec
- Special built-ins
  - : . break continue eval exec export exit readonly return set shift trap unset

Important Built-in Commands

exec	:	replaces shell with program
cd	:	change working directory
shift	:	rearrange positional parameters
set	:	set positional parameters
wait	:	wait for background proc. to exit
umask	:	change default file permissions
exit	:	quit the shell
eval	:	parse and execute string
time	:	run command and print times
export	:	put variable into environment
trap	:	set signal handlers

Important Built-in Commands

continue	:	continue in loop
break	:	break in loop
return	:	return from function
:	:	true
.	:	read file of commands into current shell; like **`#include`**

Functions

Functions are similar to scripts and other commands except that they can produce side effects in the callers script. The positional parameters are saved and restored when invoking a function. Variables are shared between caller and callee.

Syntax:

```
name ()
{
    commands
}
```

Aliases

- Like macros (#define in C)
- Shorter to define than functions, but more limited
- Not recommended for scripts
- Example:

```
alias rm='rm -i'
```

Search Rules

- Special built-ins
- Functions
  - *command* bypasses search for functions
- Built-ins not associated with PATH
- PATH search
- Built-ins associated with PATH
- Executable images

Parsing and Quoting

How the Shell Parses

- Part 1: Read the command:
  - Read one or more lines a needed
  - Separate into *tokens* using space/tabs
  - Form commands based on token types
- Part 2: Evaluate a command:
  - Expand word tokens (command substitution, parameter expansion)
  - ***Split words into fields***
  - Setup redirections, environment
  - Run command with arguments

Useful Program for Testing

/home/unixtool/bin/showargs

```
#include <stdio.h>
int main(int argc, char *argv[])
{
  int i;
  for (i=0; i < argc; i++) {
    printf("Arg %d: %s\n", i, argv[i]);
  }
  return(0);
}
```

Shell Comments

- Comments begin with an unquoted **#**
- Comments end at the end of the line
- Comments can begin whenever a token begins
- Examples

```
# This is a comment
# and so is this
grep foo bar # this is a comment
grep foo bar# this is not a comment
```

Special Characters

- The shell processes the following characters specially unless quoted:

  `| & ( ) < > ; " ' $ `` ` `` *space tab newline*

- The following are special whenever patterns are processed:

  `* ? [ ]`

- The following are special at the beginning of a word:

  `# ~`

- The following is special when processing assignments:

  `=`

Token Types

- The shell uses spaces and tabs to split the line or lines into the following types of tokens:
  - Control operators
  - Redirection operators
  - Reserved words
  - Assignment tokens
  - Word tokens

Operator Tokens

- Operator tokens are recognized everywhere unless quoted. Spaces are optional before and after operator tokens.
- I/O Redirection Operators:

```
> >> >| >& < << <<- <&
```

  - Each I/O operator can be immediately preceded by a single digit
- Control Operators:

```
| & ; ( ) || && ;;
```

Shell Quoting

- Quoting causes characters to loose special meaning.
- `\` Unless quoted, `\` causes next character to be quoted. In front of new-line causes lines to be joined.
- `'`…`'` Literal quotes. Cannot contain `'`
- `"`…`"` Removes special meaning of all characters except **`$`**, `"`, `\` and `` ` ``. The `\` is only special before one of these characters and new-line.

Quoting Examples

```
$ cat file*
a
b
```

```
$ cat "file*"
cat: file* not found
```

```
$ cat file1 > /dev/null
$ cat file1 ">" /dev/null
a
cat: >: cannot open
```

```
FILES="file1 file2"
$ cat "$FILES"
cat: file1 file2 not found
```

Simple Commands

- A simple command consists of three types of tokens:
  - Assignments (must come first)
  - Command word tokens
  - Redirections: *redirection-op* + *word-op*
  - The first token must not be a reserved word
  - Command terminated by new-line or `;`
- Example:

  -
    ```
    foo=bar z=`date`
    echo $HOME
    x=foobar > q$$ $xyz z=3
    ```

Word Splitting

- **After parameter expansion**, **command substitution**, and arithmetic expansion, the characters that are generated as a result of these expansions that are not inside double quotes are checked for split characters
- Default split character is *space* or *tab*
- Split characters are defined by the value of the **`IFS`** variable (**`IFS=""`** disables)

Word Splitting Examples

```
FILES="file1 file2"
cat $FILES
a
b

IFS=
cat $FILES
cat: file1 file2: cannot open
```

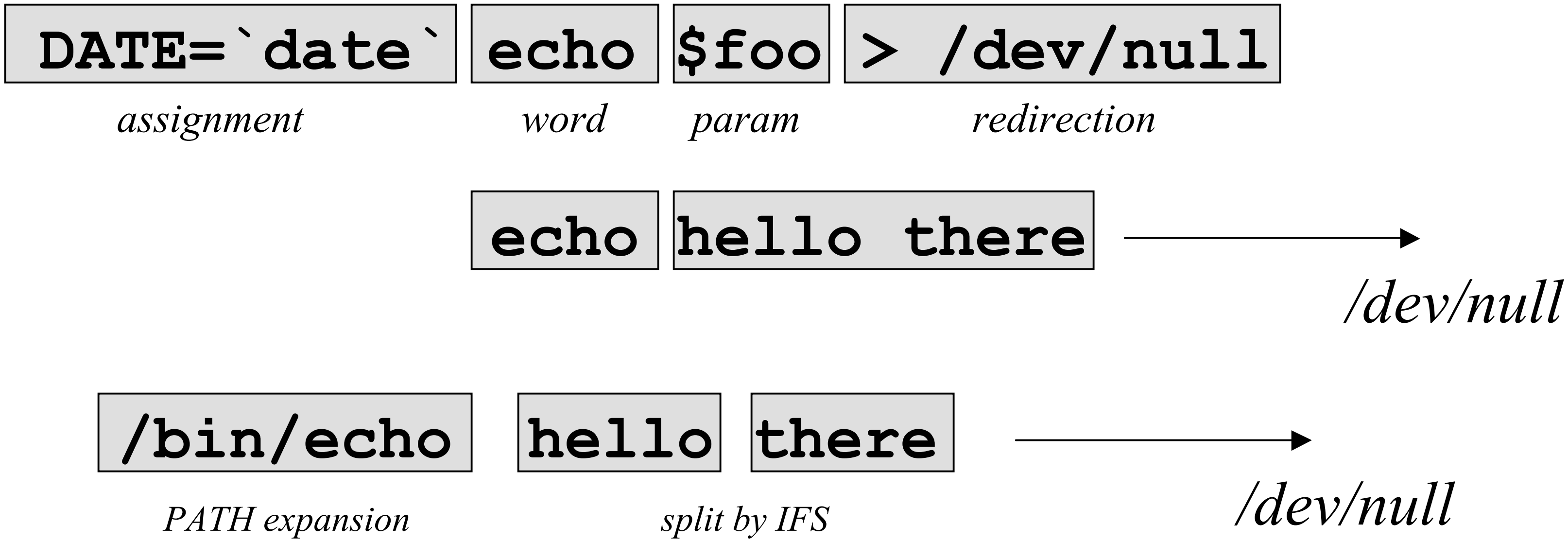
```
IFS=x v=exit
echo exit $v "$v"
exit e it exit
```

Pathname Expansion

- **After** word splitting, each field that contains pattern characters is replaced by the pathnames that match
- Quoting prevents expansion
- **`set –o noglob`** disables
  - Not in original Bourne shell, but in POSIX

Parsing Example

```
DATE=`date` echo $foo > \
     /dev/null
```



`DATE=`date`` `echo` `$foo` `> /dev/null`

assignment *word* *param* *redirection*

`echo` `hello there` → */dev/null*

`/bin/echo` `hello` `there` → */dev/null*

PATH expansion *split by IFS*

The eval built-in

- **`eval`** ***`arg`*** …
  - Causes all the tokenizing and expansions to be performed again

trap command

- **trap** specifies command that should be **eval**ed when the shell receives a signal of a particular value.
- **`trap [ [`*`command`*`] {`*`signal`*`}+]`**
  - If ***command*** is omitted, signals are ignored
- Especially useful for cleaning up temporary files

```
trap 'echo "please, dont interrupt!"' SIGINT

trap 'rm /tmp/tmpfile' EXIT
```

Reading Lines

- **read** is used to read a line from a file and to store the result into shell variables
  - **read –r** prevents special processing
  - Uses **IFS** to split into words
  - If no variable specified, uses **REPLY**

```
read
read –r NAME
read FIRSTNAME LASTNAME
```

Script Examples

- Rename files to lower case
- Strip CR from files
- Emit HTML for directory contents

Rename files

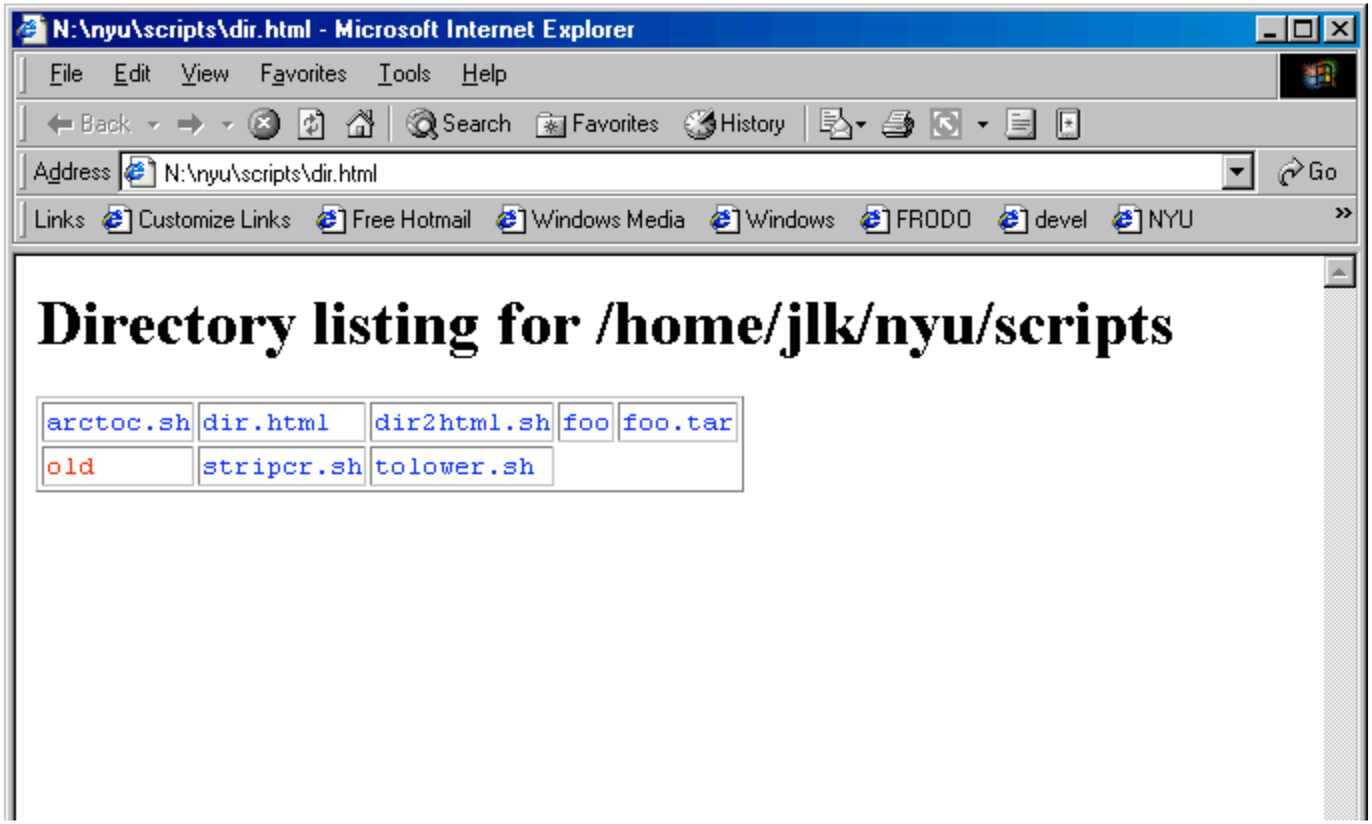
```
#!/bin/sh

for file in *
do
        lfile=`echo $file | tr A-Z a-z`
        if [ $file != $lfile ]
        then
                mv $file $lfile
        fi
done
```

Remove DOS Carriage Returns

```
#!/bin/sh

TMPFILE=/tmp/file$$

if [ "$1" = "" ]
then
        tr -d '\r'
        exit 0
fi

trap 'rm -f $TMPFILE' 1 2 3 6 15

for file in "$@"
do
        if tr -d '\r' < $file > $TMPFILE
        then
                mv $TMPFILE $file
        fi
done
```

Generate HTML

```
$ dir2html.sh > dir.html
```


N:\nyu\scripts\dir.html - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites History
Address N:\nyu\scripts\dir.html Go
Links Customize Links Free Hotmail Windows Media Windows FRODO devel NYU »

Directory listing for /home/jlk/nyu/scripts

arctoc.sh	dir.html	dir2html.sh	foo	foo.tar
old	stripcr.sh	tolower.sh		

The Script

```
#!/bin/sh

[ "$1" != "" ] && cd "$1"

cat <<HUP
 <html>
 <h1> Directory listing for $PWD </h1>
 <table border=1>
 <tr>
HUP
num=0
for file in *
do
    genhtml $file   # this function is on next
page
done
cat <<HUP
 </tr>
 </table>
 </html>
HUP
```

Function genhtml

```
genhtml()
{
    file=$1
    echo "<td><tt>"
    if [ -f $file ]
    then    echo "<font color=blue>$file</font>"
    elif [ -d $file ]
    then    echo "<font color=red>$file</font>"
    else    echo "$file"
    fi
    echo "</tt></td>"
    num=`expr $num + 1`
    if [ $num -gt 4 ]
    then
        echo "</tr><tr>"
        num=0
    fi
}
```

Korn Shell / bash Features

Command Substitution

- Better syntax with $(*command*)
  - Allows nesting
  - **`x=$(cat $(generate_file_list))`**
- Backward compatible with ` … ` notation

Expressions

- Expressions are built-in with the **`[[ ]]`** operator

  `if [[ $var = "" ]]` …
- Gets around parsing quirks of **/bin/test,** allows checking strings against *patterns*
- Operations:
  - *string* **`==`** *pattern*
  - *string* **`!=`** *pattern*
  - *string1* **`<`** *string2*
  - *file1* **`-nt`** *file2*
  - *file1* **`-ot`** *file2*
  - *file1* **`-ef`** *file2*
  - **`&&`**, **`||`**

Patterns

- Can be used to do string matching:

```
if [[ $foo = *a* ]]
if [[ $foo = [abc]* ]]
```

- Similar to regular expressions, but different syntax

Additional Parameter Expansion

- `${#`*param*`}` – Length of *param*
- `${`*param*`#`*pattern*`}` – Left strip min *pattern*
- `${`*param*`##`*pattern*`}` – Left strip max *pattern*
- `${`*param*`%`*pattern*`}` – Right strip min *pattern*
- `${`*param*`%%`*pattern*`}` – Right strip max *pattern*
- `${`*param*`-`*value*`}` – Default *value* if *param* not set

Variables

- Variables can be arrays
  - `foo[3]=test`
  - `echo ${foo[3]}`
- Indexed by number
- **`${#arr}`** is length of the array
- Multiple array elements can be set at once:
  - `set -A foo a b c d`
  - `echo ${foo[1]}`
  - Set command can also be used for positional params: `set a b c d;  print $2`

Printing

- Built-in **print** command to replace echo
- Much faster
- Allows options:

  -u#   print to specific file descriptor

Functions

- Alternative function syntax:

```
function name {
    commands
  }
```

- Allows for local variables
- $0 is set to the name of the function

Additional Features

- Built-in arithmetic: Using $((*expression*))
  - e.g., **`print $(( 1 + 1 * 8 / x ))`**
- Tilde file expansion

~	$HOME
~**user**	home directory of user
~+	$PWD
~**-**	$OLDPWD

KornShell 93

Variable Attributes

- By default attributes hold strings of unlimited length
- Attributes can be set with typeset:
  - readonly (-r) – cannot be changed
  - export (-x) – value will be exported to env
  - upper (-u) – letters will be converted to upper case
  - lower (-l) – letters will be converted to lower case
  - ljust (-L *width*) – left justify to given width
  - rjust (-R *width*) – right justify to given width
  - zfill (-Z *width*) – justify, fill with leading zeros
  - integer (-I [*base*]) – value stored as integer
  - float (-E [*prec*]) – value stored as C double
  - nameref (-n) – a name reference

Name References

- A name reference is a type of variable that references another variable.
- **nameref** is an alias for **typeset -n**
  - Example:

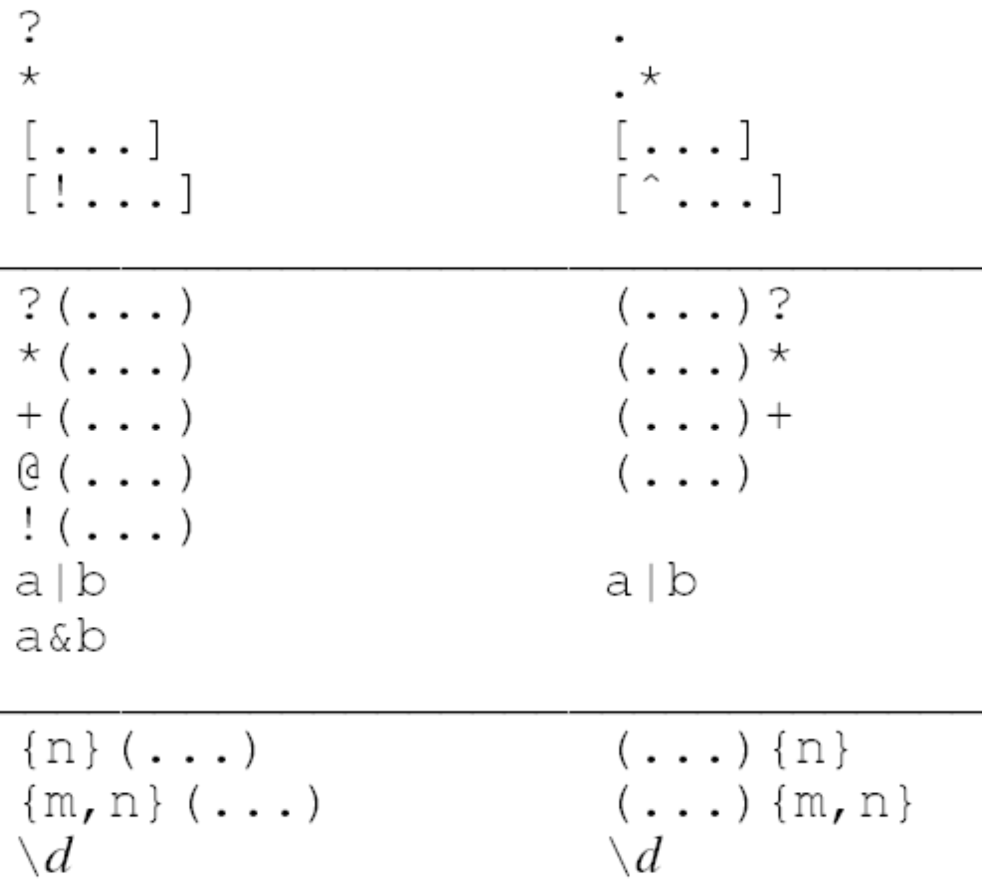
```
user1=”mehryar"
user2="adam"
typeset -n name="user1"
print $name
mehryar
```

New Parameter Expansion

- ${param/pattern/str} – Replace first pattern with str
- ${param//pattern/str} – Replace all patterns with str
- ${param:offset:len} – Substring with offset

Patterns Extended

- Additional pattern types so that shell patterns are equally expressive as regular expressions
- Used for:
  - file expansion
  - **[[ ]]**
  - case statements
  - parameter expansion



Patterns	*Regular Expressions*
?	.
*	.*
[...]	[...]
[!...]	[^...]
?(...)	(...)?
*(...)	(...)*
+(...)	(...)+
@(...)	(...)
!(...)	
a|b	a|b
a&b	
{n}(...)	(...){n}
{m,n}(...)	(...){m,n}
\d	\d

ANSI C Quoting

- **`$'`…`'`** Uses C escape sequences

  `$'\t'   $'Hello\nthere'`

- **printf** added that supports C like printing:

  `printf "You have %d apples" $x`

- Extensions
  - `%b` – ANSI escape sequences
  - `%q` – Quote argument for reinput
  - `\E` – Escape character (033)
  - `%P` – convert ERE to shell pattern
  - `%H` – convert using HTML conventions
  - `%T` – date conversions using date formats

Associative Arrays

- Arrays can be indexed by string
- Declared with **`typeset -A`**
- Set: **`name["foo"]="bar"`**
- Reference **`${name["foo"]}`**
- Subscripts: **`${!name[@]}`**

Corresponding Shell Features

- Standard input, output, error
  - Redirection
  - Here documents
  - Pipelines
  - Command substitution
- Exit status
  - $?
  - &&, ||, if, while
- Environment
  - export, variables
- Arguments
  - Command substitution
  - Variables
  - Wildcards