Lecture 6

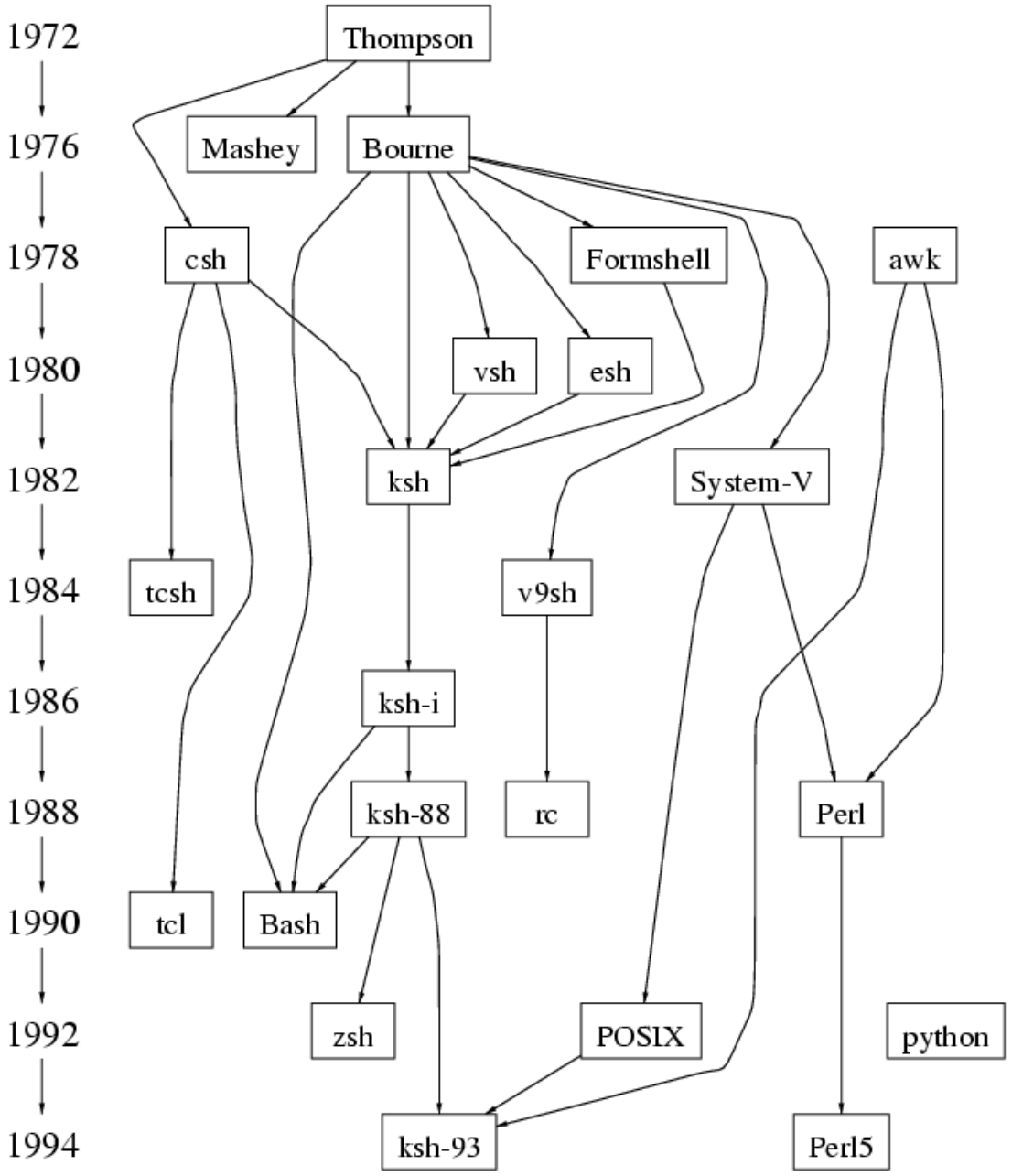
Shell Scripting

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

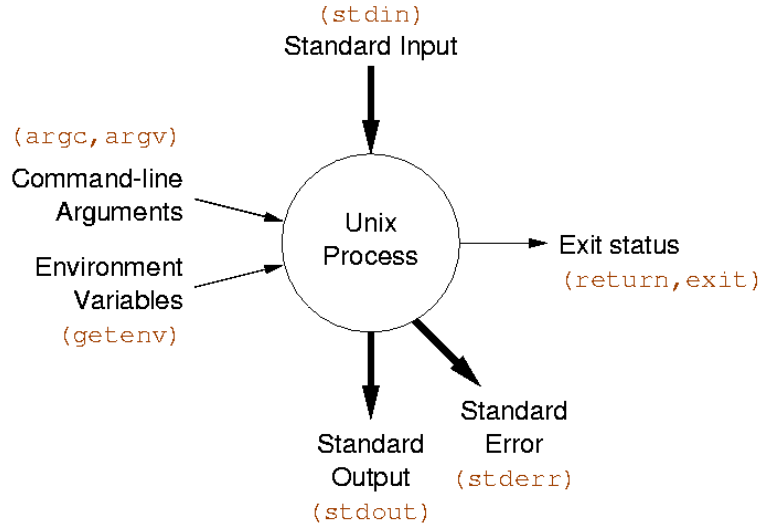
When you log in, you interactively use the shell

Scripting

A set of shell commands that constitute an executable program

UNIX Programs

- **Means of input**:
  - Program arguments [control information]
  - Standard input [data]
  - Environment variables [state information]
- **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:
  - **`sh name [ arg … ]`**
  - **`sh < 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
  - *Slower*
  - *Not well suited for algorithms & data structures*

The C Shell

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

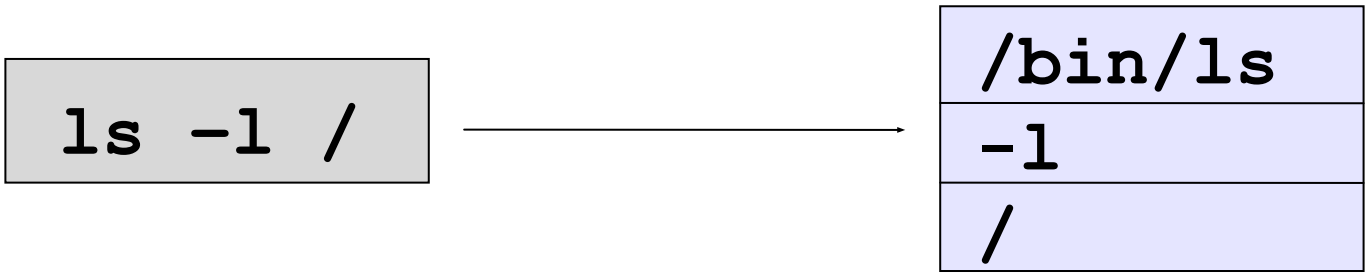
http://www.faqs.org/faqs/unix-faq/shell/csh-whynot

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*



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);
}
```

Background Commands

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

```
firefox &
```

- **`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
- **Syntax:**

```
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 billg@microsoft.com <<EOT
   I hope Windows 10 is almost as
   good as UNIX which I wrote 45
   years ago. Yours,
   Ken Thompson
EOT
```

Shell Variables

- To set:

  name=value

- Read: **`$var`**
- Variables can be local or environment. Environment variables are part of UNIX and can be accessed by child processes.
- 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
`$PWD`	Current directory
`$USER`	Your login name
`$SHELL`	Absolute pathname of login shell
`$TERM`	Type of your terminal
`$PS1`	Prompt

Here Documents Expand Vars

```
#!/bin/sh

mail student@nyu.edu <<EOT
   You got a $SCORE on assignment
   $ASSIGNMENT.
   Yours,
   $GRADER
EOT
```

Parameters

- A parameter is one of the following:
  - A variable
  - A positional parameter, starting from `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

- `$#` Number of positional parameters
- `$?` 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" "$2" ...`
- `$-` Options currently in effect

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

bash Arrays

- 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:

```
foo=(a b c d)
echo ${foo[1]}
```

- **set** command can also be used for positional params:

```
set a b c d;  print $2
```

Command Substitution

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

```
$ date
Wed Oct 8 14:40:56 EDT 2008
$ NOW=`date`

$ grep `date +%Y` myfile.c

$ sed "s/oldtext/`ls | head -1`/g"
$ PATH=`myscript`:$PATH
```

POSIX Command Substitution

- Better syntax with $(*command*)
- Allows nesting

```
x=$(cat $(generate_file_list))
```

- Backward compatible with `...` notation

File name expansion

- Used to generate a set of arguments from files
- Wildcards (***shell 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
- This is the same syntax that **find** uses
  - … but are not regular expressions!

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 (programs don't see 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 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 2> /dev/null && date
$ ls bad_file 2> /dev/null || date
Wed Oct 9 07:43:23 2013
```

Control Structures

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

What is an expression?

- Simply a UNIX command. Evaluates to true if the exit code is 0, false if the exit code is not 0.
- Special command **`/bin/test`** exists that does most common expressions:
  - String comparison
  - Numeric comparison
  - Check file properties
- **`[`** often a builtin version of **`/bin/test`** for syntactic sugar
- Great example UNIX tools working together

Examples

```
if test $USER = kornj
then
   echo "I know you"
else
   echo "I don't 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

bash 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*

  `&&, ||`

bash Patterns

- Can be used to do string matching:

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

- Same as patterns used for file expansion

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
```

  - Particularly useful with command substitution

```
X=$(expr $X + 2)
```

- **bash** has built-in arithmetic:

```
echo $(( 3 * 4 + 5 ))
```

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 parameters 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

for INPUT in "$@"
do
    case $INPUT 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 **|**

```
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 current process (**cd**)
- They perform faster
  - No fork/exec

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
export	run command and print times

Important Built-in Commands

time	put variable into environment
trap	set signal handlers
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:
  - They can produce side effects in the callers script.
  - Variables are shared between caller and callee.
  - The positional parameters are saved and restored when invoking a function.
- Syntax:

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

bash Functions

- Alternative function syntax:

```
function name {
   commands
}
```

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

Aliases

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

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

Command Search Rules

1. Special built-ins
2. Functions
   - `command` bypasses search for functions
3. Built-ins not associated with `PATH`
4. `PATH` search
5. Built-ins associated with `PATH`

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***
- File expansion
- Setup redirections, environment
- Run command with arguments

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
```

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 newline 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
```

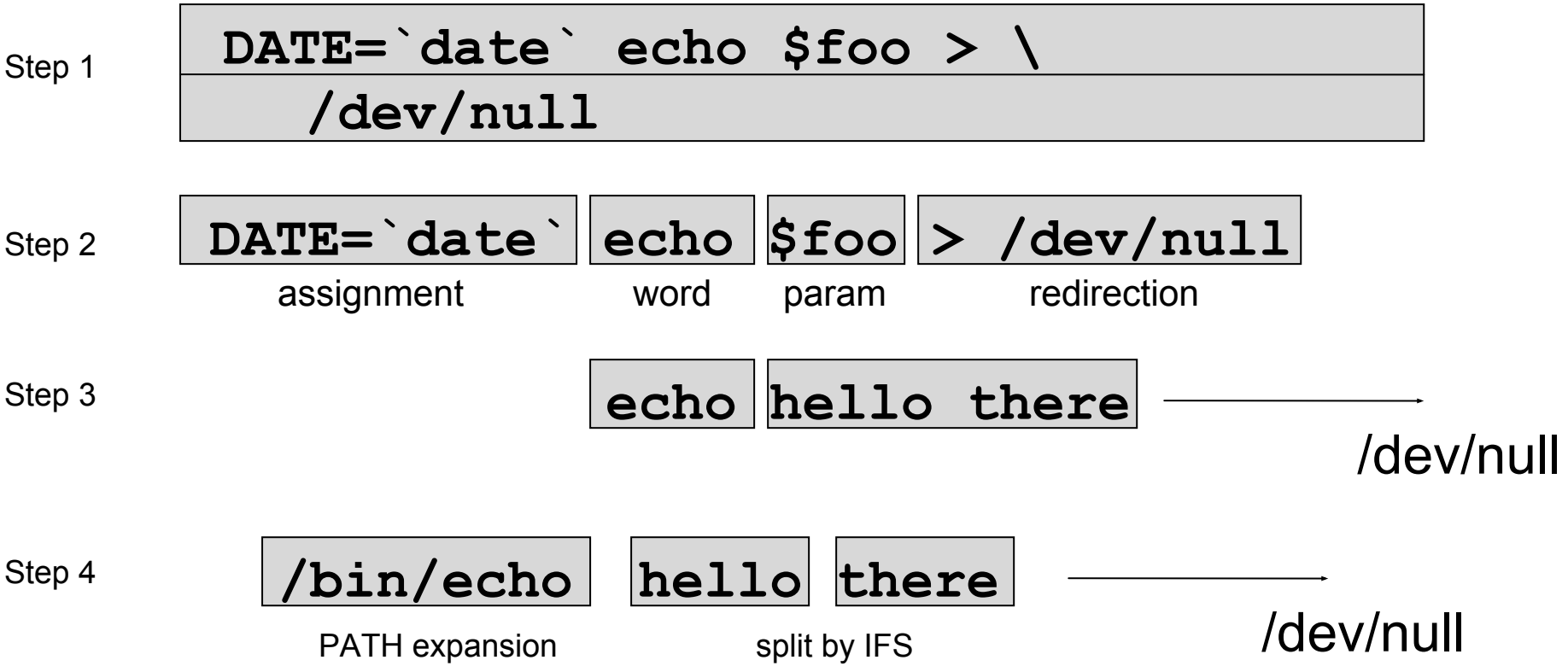
```
IFS=x v=exit
echo exit $v "$v"
exit e it exit
```

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 entirely

Parsing Example



Script Examples

1. Rename files to lower case
2. Strip CR from files
3. Emit HTML for directory contents

Rename files

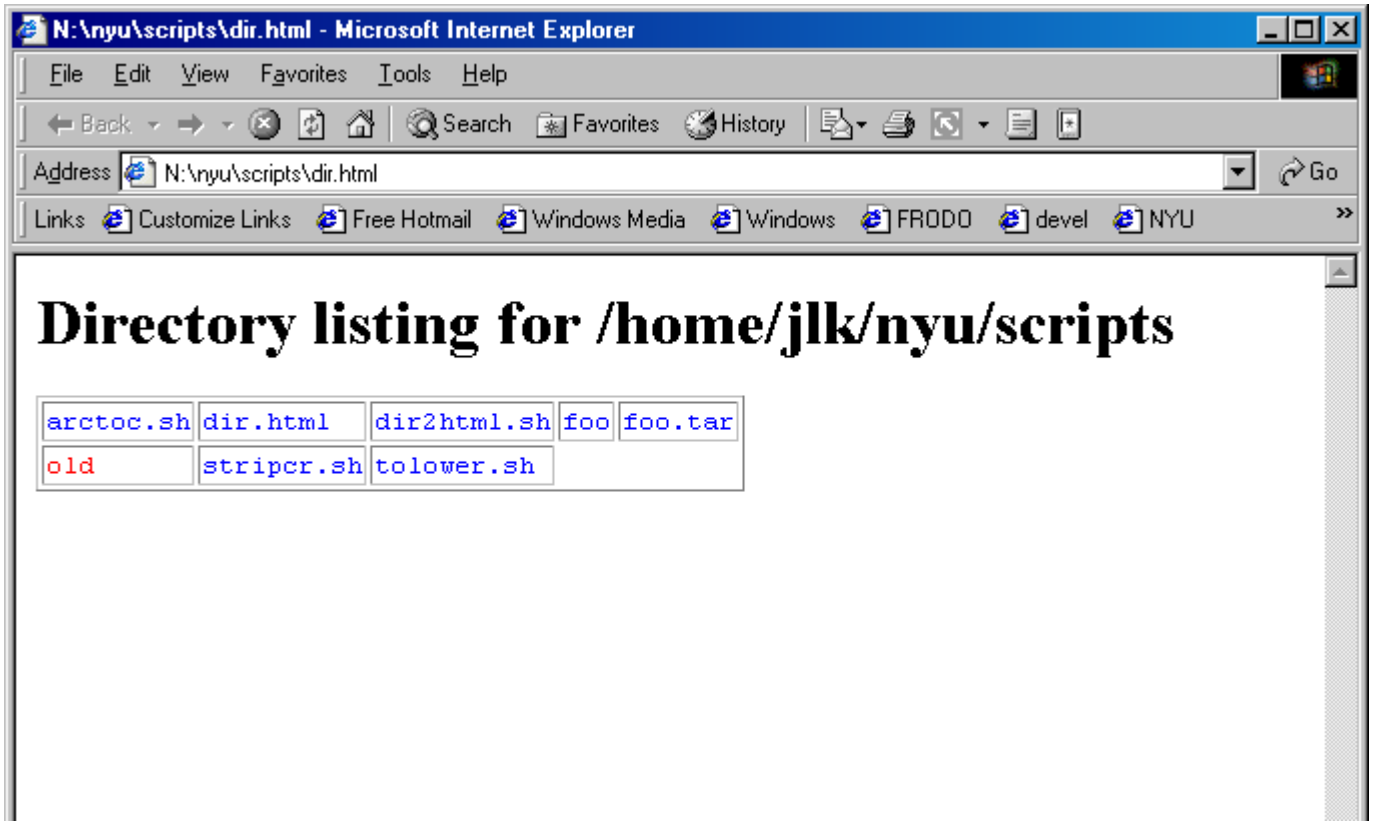
```
#!/bin/sh

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

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

Generate HTML

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



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
}
```