Command Interpreters

A *command interpreter* is a program that executes other programs.
Aim: allow users to execute the commands provided on a computer system.
Command interpreters come in two flavours:

- graphical (e.g. Windows or Mac desktop)
  - advantage: easy for naive users to start using system
- command-line (e.g. Unix shell)
  - advantage: programmable, powerful tool for expert users

On Unix/Linux, `bash` has become defacto standard shell.

What Shells Do

All Unix shells have the same basic mode of operation:

```
loop
   if (interactive) print a prompt
   read a line of user input
   apply transformations to line
   split line into words (/\s+/)
   use first word in line as command name
   execute that command,
      using other words as arguments
end loop
```

Note that "`line of user input`" could be a line from a file.
In that case, the shell is reading a "script" of commands and acting as a kind of programming language interpreter.

What Shells Do

The "`transformations`" applied to input lines include:

- variable expansion ... e.g. `$1 ${x-20}`
- file name expansion ... e.g. `*.c enr.07s?`

To "`execute that command`" the shell needs to:

- find file containing named program (`PATH`)
- start new process for execution of program

Command Search PATH

If we have a script called `bling` in the current directory, we might be able to execute it with any of these:

```
$ sh bling   # file need not be executable
$ ./bling    # file must be executable
$ bling      # file must be executable and . in $PATH
```

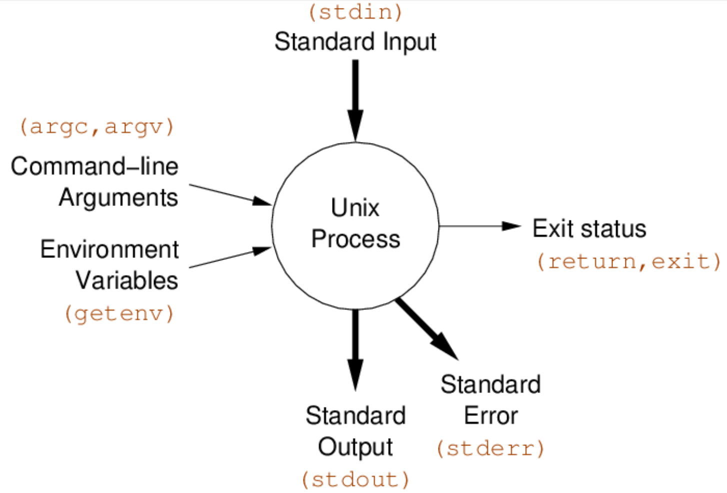
Shell searches for programs to run using the colon-separated list of directories in the variable PATH Beware only append the current directory to end of your path, e.g:

```
$ PATH=.:$PATH
$ cat >cat <<eof
#!/bin/sh
echo miaou
eof
$ chmod 755 cat
$ cat /home/cs2041/public_html/index.html
miaou
$
```

Nore ./cat is being run rather /bin/cat
Much hard to discover if it happens with another shell script which runs cat. Safer still: don't put . in your PATH.

Unix Processes

A Unix process executes in this environment



Unix Processes: C Program View

Components of process environment (C programmer's view):

- `char *argv[]` - command line "words"
- `int argc` - size of `argv[]`
- `char *env[]` - `name=value` pairs from shell
- `FILE *stdin` - input byte-stream, e.g. `getchar)`
- `FILE *stdout` - input byte-stream, e.g. (`putchar()`
- `FILE *stderr` - output byte-stream, e.g. `fputc(c, stderr)`
- `exit(int)` - terminate program, set exit status
- `return int` - terminate `main()`, set exit status

Shell as Interpreter

The shell can be viewed as a programming language interpreter.
As with all interpreters, the shell has:

- a state (collection of variables and their values)
- control (current location, execution flow)

Different to most interpreters, the shell:

- modifies the program code before finally executing it
- has an infinitely extendible set of basic operations

Shell as Interpreter

Basic operations in shell scripts are a sequence of *words*.

```
CommandName  Arg1  Arg2  Arg3  ...
```

A *word* is defined to be any sequence of:

- non-whitespace characters (e.g. `x, y1, aVeryLongWord`)
- characters enclosed in double-quotes (e.g. `"abc", "a b c"`)
- characters enclosed in single-quotes (e.g. `'abc', 'a b c'`)

We discuss the different kinds of quote later.

Shell Scripts

Consider a file called "`hello`" containing

```
#!/bin/sh

echo Hello, World
```

How to execute it?

```
    $ sh hello            # execute the script
```

or

```
    $ chmod +x hello      # make the file executable
    $ ./hello             # execute the script
```


Shell Scripts

The next simplest shell program: "Hello, *YourName*"

```
#!/bin/sh

echo -n "Enter your name: "
read name
echo Hello, $name
```

Shell variables:

```
$ read x     # read a value into variable x
$ y=John     # assign a value to variable y
$ echo $x    # display the {\it{value of}} variable x
$ z="$y $y"  # assign two copies of y to variable z
```

Note: spaces matter ... do *not* put spaces around the = symbol.

Shell Variables

More on shell variables:

- no need to declare shell variables; simply use them
- are local to the current execution of the shell.
- all variables have type string
- initial value of variable = empty string
- note that `x=1` is equivalent to `x="1"`

Examples:

```
$ x=5
$ y="6"
$ z=abc
$ echo $(( $x + $y ))
11
$ echo $(( $x + $z ))
5
```


Shell Variables

```
$ x=1
$ y=fred
$ echo $x$y
1fred
$ echo $xy         # the aim is to display "1y"

$ echo "$x"y
1y
$ echo ${x}y
1y
$ echo ${j-10}     # give value of j or 10 if no value
10
$ echo ${j=33}     # set j to 33 if no value (and give $j)
33
$ echo ${x:?No Value}   # display "No Value" if $x not set
1
$ echo ${xx:?No Value}  # display "No Value" if $xx not set
-bash: xx: No Value
```

Shell Scripts

Some shell built-in variables with pre-assigned values:

`$0`	the name of the command
`$1`	the first command-line argument
`$2`	the second command-line argument
`$3`	the third command-line argument
`$#`	count of command-line arguments
`$*`	all of the command-line arguments (together)
`$@`	all of the command-line arguments (separately)
`$?`	exit status of the most recent command
`$$`	process ID of this shell

The last one is useful for generating unique filenames.

Shell Scripts

Tip: debugging for shell scripts

- the shell transforms commands before executing
- can be useful to know what commands are executed
- can be useful to know what transformations produced
- `set -x` shows each command after transformation

i.e. execution trace

Quoting

Quoting can be used for three purposes in the shell:

- to group a sequence of words into a single "word"
- to control the kinds of transformations that are performed
- to capture the output of commands (back-quotes)

The three different kinds of quotes have three different effects:

single-quote (')	grouping, turns off all transformations
double-quote (")	grouping, no transformations except $ and `
backquote (`)	no grouping, capture command results

Quoting

Single-quotes are useful to pass shell meta-characters in args: e.g.
`grep 'S.*[0-9]+$' < myfile`
Use double-quotes to

- construct strings using the values of shell variables
  e.g. `"x=$x, y=$y"` like Java's `("x=" + x + ", y=" + y)`
- prevent empty variables from "vanishing"
  e.g. use `test "$x" = "abc"` rather than `test $x = "abc"`
  in case `$x` is empty
- for values obtained from the command line or a user
  e.g. use `test -f " $1"` rather than `test -f $1` in case
  `$1` contains a path with spaces (e.g. `C:/Program Files/app/data`

Back-quotes

Back-quotes capture output of command as shell values.
For ‘*Command*‘, the shell:

1. performs variable-substitution (as for double-quotes)
2. executes the resulting command and arguments
3. captures the standard output from the command
4. converts it to a single string
5. uses this string as the value of the expression

Back-quotes

Example: convert GIF files to PNG format.
Original and converted files share the same prefix
(e.g. `/x/y/abc.gif` is converted to `/x/y/abc.png`)

```
#!/bin/sh
# ungif - convert gifs to PNG format

for f in "$@"
do
    dir=`dirname "$f"`
    prefix=`basename "$f" .gif`
    outfile="$dir/$prefix.png"
    giftopnm "$f" | pnmtopng > "$outfile"
done
```


Connecting Commands

The shell provides *I/O redirection* to allow us to change where processes read from and write to.

`< infile`	connect `stdin` to the file `infile`
`> outfile`	connect `stdout` to the file `outfile`
`>> outfile`	apppend `stdout` to the file `outfile`
`2> outfile`	connect `stderr` to the file `outfile`
`2>&1 > outfile`	connect `stderr`+`stdout` to `outfile`

Beware: > truncates file before executing command.
Always have backups!

Connecting Commands

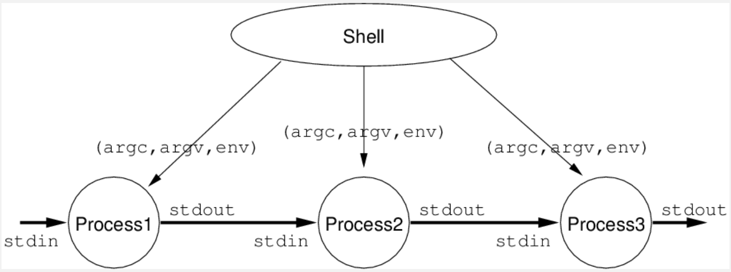
Many commands accept list of input files:
E.g. `cat file1 file2 file3`
These commands also typically adopt the conventions:

- read contents of `stdin` if no filename arguments
- treat the filename `-` as meaning `stdin`

E.g. `cat -n < file` and `cat a - b - c`
If a command does not allow this, use:
E.g. `cat file1 file2 file3 | Command`

Connecting Commands

The shell sets up the environment for each command in a pipeline and connects them together:



Exit Status and Control

Process exit status is used for control in shell scripts:

- zero exit status means command successful $\rightarrow$ true
- non-zero exit status means error occurred $\rightarrow$ false

Mostly, exit status is simply ignored (e.g. when interactive)
One application of exit statuses:

- AND lists cmd_1 && cmd_2 && ... && cmd_n
  (cmd_{i+1} is executed only if cmd_i succeeds (zero exit status))
- OR lists cmd_1 || cmd_2 || ... || cmd_n
  (cmd_{i+1} is executed only if cmd_i fails (non-zero exit status))

Testing

The `test` command performs a test or combination of tests and

- returns a zero exit status if the test succeeds
- returns a non-zero exit status if the test fails

Provides a variety of useful testing features:

- string comparison (`=` `!=`)
- numeric comparison (`-eq` `-ne` `-lt`)
- checks on files (`-f` `-x` `-r`)
- boolean operators (`-a` `-o` `!`)

Testing

Examples:

```
# does the variable msg have the value "Hello"?
test "$msg" = "Hello"

# does x contain a numeric value larger than y?
test "$x" -gt "$y"

# Error: expands to "test hello there = Hello"?
msg="hello there"
test $msg = Hello

# is the value of x in range 10..20?
test "$x" -ge 10 -a "$x" -le 20

# is the file xyz a readable directory?
test -r xyz -a -d xyz

# alternative syntax; requires closing ]
[ -r xyz -a -d xyz ]
```

Note: use of quotes, spaces around values/operators

Sequential Execution

Combine commands in pipelines and AND and OR lists.
Commands executed sequentially if separated by semicolon or newline.

cmd_1 ; cmd_2 ; ... ; cmd_n
cmd_1
cmd_2
...
cmd_n

Grouping

Commands can be grouped using `( ... )` or `{ ... }`
(cmd_1 ; ... cmd_n) are executed in a new sub-shell.
{cmd_1 ; ... cmd_n } are executed in the current shell.
Exit status of group is exit status of last command.
Beware: state of sub-shell (e.g. `$PWD`, other variables) is lost after `(...)`, hence

```
$ cd /usr/share
$ x=123
$ ( cd $HOME;  x=abc; )
$ echo $PWD $x
/usr/share 123
$ { cd $HOME;  x=abc; }
$ echo $PWD $x
/home/cs2041 abc
```


If Command

The `if-then-else` construct allows conditional execution:

```
if testList{1}
then
   commandList{1}
elif testList{2}
then
   commandList{2}
...
else
   commandList{n}
fi
```

Keywords `if`, `else` etc, are only recognised at the start of a command (after newline or semicolon).

If Command

Examples:

```
# Check whether a file is readable

if [ -r $HOME ]     # neater than:  if test -r $HOME
then
   echo "$0: $HOME is readable"
fi

# Test whether a user exists in passwd file

if grep "^$user" /etc/passwd > /dev/null
then
    ... do something if they do exist ...
else
   echo "$0: $user does not exist"
fi
```

Case command

case provides multi-way choice based on patterns:

```
case word in
pattern{1})  commandList{1} ;;
pattern{2}-2)  commandList{2}-2 ;;
...
pattern{n})  commandList{n} ;;
esac
```

The *word* is compared to each *pattern*$_i$ in turn.
For the first matching pattern, corresponding *commandList*$_i$ is executed and the statement finishes.
Patterns are those used in filename expansion (* ? []).

Case command

Examples:

```
# Checking number of command line args

case $# in
0)  echo "You forgot to supply the argument" ;;
1)   ... process the argument ...  ;;
*)  echo "You supplied too many arguments" ;;
esac

# Classifying a file via its name

case "$file" in
*.c)  echo "$file looks like a C source-code file" ;;
*.h)  echo "$file looks like a C header file" ;;
*.o)  echo "$file looks like a an object file" ;;
...
?)    echo "$file's name is too short to classify it" ;;
*)    echo "I have no idea what $file is" ;;
esac
```


Loop commands

while loops iterate based on a test command list:

```
while testList
do
   commandList
done
```

for loops set a variable to successive words from a list:

```
for var in wordList
do
   commandList  # ... generally involving  var
done
```


Loop commands

Examples of while:

```
# Check the system status every ten minutes

while true
do
   uptime ; sleep 600
done

# Interactively prompt the user to process files

echo -n "Next file: "
while read filename
do
   process < "$filename" >> results
   echo -n "Next file: "
done
```

Loop commands

Examples of `for`:

```
# Compute sum of a list of numbers from command line

sum=0
for n in "$@"    # use "$@" to preserve args
do
   sum=`expr $sum + "$n"`
done

# Process files in $PWD, asking for confirmation

for file in *
do
   echo -n "Process $file? "
   read answer
   case "$answer" in
      [yY]*) process < $file >> results ;;
      *)     ;;
   esac
done
```