An Introduction to Shell Programming

Last Edit
March 23, 1992

Reg Quinton <reggers@julian.uwo.ca>
Computing and Communications Services
The University of Western Ontario
London, Ontario N6A 5B7
Canada

1. Bourne Shell

A shell is a *command line interpretor* (cf. DCL on VAX/VMS). It takes commands and executes them. As such, it implements a programming language. The Bourne shell is used to create *shell scripts* — ie. programs that are interpreted/executed by the shell. You can write shell scripts with the C-shell; however, this is *not* recommended.

There are lots of shell scripts on the system. For example

```
[10:45am julian] cd /bin
[10:46am julian] file * | grep /bin/sh
basename:    executable script for  /bin/sh
dirname:     executable script for  /bin/sh
echo:        executable script for  /bin/sh
false:       executable script for  /bin/sh
test:        executable script for  /bin/sh
tplot:       executable script for  /bin/sh
true:        executable script for  /bin/sh
```

etc...

Other good examples in the boot sequence — **/etc/rc.*** or **/etc/init.d/***. On CCS systems look at examples in **/usr/local/share/bin**.

2. Creating a Script

Suppose you often type the command

find . -name *file* -print

and you'd rather type a simple command, say

sfind *file*

Create a shell script

```
[11:01am julian] cd ~/bin
[11:01am julian] ed sfind etc...
[11:03am julian] page sfind
find . -name $1 -print
[11:03am julian] chmod a+x sfind
[11:03am julian] rehash
[11:04am julian] cd /usr/src/usr.local
[11:04am julian] sfind tcsh
./shells/tcsh
```


2.1. Observations

This quick example is far from adequate but some observations:

(1) Shell scripts are simple text files created with an editor.

(2) Shell scripts are marked as **executeable** —

[11:14am julian] chmod a+x sfind

(3) Should be located in your search path and **~/bin** should be in your search path.

(4) You likely need to **rehash** if you're a Csh user (but not again when you login).

(5) Arguments are passed from the command line and referenced. For example, as **$1**.

(6) Within a shell script — any command!

3. A Real Example

We have many shell scripts we've made.

```
[11:45am julian] man day

DAY(Local)        EP/IX Reference Manual

NAME
    day - convert date string to day of the week

SYNOPSIS
    day date month year
```

etc...

```
[11:44am julian] day 19 Feb 92
Wed
[11:44am julian] which day
/usr/local/share/bin/day
[11:49am julian] cd /usr/local/share/bin
[11:49am julian] file day
day:     executable script for /bin/sh
```

Note: installed utilities should have a manual page, should conform to Unix conventions, etc.

3.1. #!/bin/sh

All Bourne Shell scripts should begin with the sequence

```
[12:52pm julian] page day
#!/bin/sh
```

etc...

From **exec(2)**:

> "On the first line of an interpreter script, following the "#!", is the name of a program which should be used to interpret the contents of the file. For instance, if the first line contains "#! /bin/sh", then the contents of the file are executed as a shell script."

You can get away without this, but you shouldn't. All good scripts state the interpretor explicitly. Long ago there was just one (the Bourne Shell) but these days there are many interpretors — Cshell, Ksh, Bash, and others.

3.2. Comments/RCS header

All good Bourne Shell scripts should begin with the sequence

```
[12:58pm julian] page day
#!/bin/sh
#
# $Author: reggers $
# $Date: 91/10/30 14:31:19 $
# $Header: /ccs/export/share/ftp/pub/unix/uti...
#
#       Usage: day 19 Oct 91
#       reports back the day of the week
#
# January 1, 1901 was a Tuesday; this is my ...
#
# Bugs: I'm not handling leap/noleap centur...
```

etc...

The Revision Control System, RCS, is a good tool for managing software projects. What version? When was it written? Where are the sources? This is recommended.

Comment your code as you build it. This should be required. Anyone should be able to read a shell script.

3.3. Search Path

All shell scripts should include a search path specification:

```
PATH=/usr/ucb:/usr/bin:/bin; export PATH
```

A PATH specification is recommended — often times a script will fail for some people because they have a different or incomplete search path.

The Bourne Shell does **not export** environment variables to children unless explicitly instructed to do so.

Beware: of "." in the search path, this opens a big hole for trojan horses.

```
PATH=:/usr/ucb:/usr/bin:/bin:; export PATH
```

This example is a **very big mistake**! Watch out for the leading and trailing colon, don't make the mistake.

3.4. Argument Checking

A good shell script should verify that the arguments supplied (if any) are correct.

```
if [ $# -ne 3 ]; then
        echo 1>&2 Usage: $0 19 Oct 91
        exit 127
fi
```

This script requires three arguments and gripes accordingly.

Some more argument checking (note the caveat):

```
# check range of day (this is quick, not perfect)

case "$day" in
        [1-9]|[123][0-9])
                ;;
        *)
                echo 1>&2 Day \"$day\" out of range ...
                exit 127
                ;;
esac
```


3.5. Exit status

All Unix utilities should return an exit status.

```
# is the year out of range for me?

if [ $year -lt 1901  -o  $year -gt 2099 ]; then
        echo 1>&2 Year \"$year\" out of range
        exit 127
fi

etc...

# All done, exit ok

exit 0
```

A non-zero exit status indicates an error condition of some sort while a zero exit status indicates things worked as expected.

On BSD systems there's been an attempt to categorize some of the more common exit status codes. See **/usr/include/sysexits.h**.

3.6. Using exit status

Exit codes are important for those who use your code. Many constructs test on the exit status of a command.

The conditional construct is:

```
if command; then
        command
fi
```

For example,

```
if tty -s; then
        echo Enter text end with \^D
fi
```

Your code should be written with the expectation that others will use it. Making sure you return a meaningful exit status will help.

3.7. Stdin, Stdout, Stderr

Standard input, output, and error are file descriptors 0, 1, and 2. Each has a particular role and should be used accordingly:

```
# is the year out of range for me?

if [ $year -lt 1901  -o  $year -gt 2099 ]; then
        echo 1>&2 Year \"$year\" out of my range
        exit 127
fi

etc...

# ok, you have the number of days since Jan 1, ...

case `expr $days % 7` in
0)
        echo Mon;;
1)
        echo Tue;;

etc...
```

Error messages should appear on stderr not on stdout!

Output should appear on stdout. As for input/output dialogue (from **purge(l)**):

```
# give the fellow a chance to quit

if tty -s ; then
        echo This will remove all files in $* since ...
        echo $n Ok to procede? $c;      read ans
        case "$ans" in
                n*|N*)
                        echo File purge abandoned;
                        exit 0   ;;
        esac
        RM="rm -rfi"
else
        RM="rm -rf"
fi
```

Note: this code behaves differently if there's a user to communicate with (ie. if the standard input is a tty rather than a pipe, or file, or etc. See **tty(1)**).

4. Language Constructs

4.1. For loop iteration

Substitute values for variable and perform task:

for *variable* in *word* ...
do
        command
done

For example, from **syslogd.daily**:

```
for i in `cat $LOGS`
do
        mv $i $i.$TODAY
        cp /dev/null $i
        chmod 664 $i
done
```

Alternatively you may see:

for *variable* in *word* ...; do *command*; done

4.2. Case switch

Switch to statements depending on pattern match

```
case word in
[ pattern [ | pattern ... ] )
        command ;; ] ...
esac
```

For example, from **day(l)**:

```
case "$year" in

[0-9][0-9])
        year=19${year}
        years=`expr $year - 1901`
        ;;
[0-9][0-9][0-9][0-9])
        years=`expr $year - 1901`
        ;;
*)
        echo 1>&2 Year \"$year\" out of range ...
        exit 127
        ;;
esac
```


4.3. Conditional Execution

Test exit status of command and branch

```
if command
then
        command
[ else
        command ]
fi
```

For example, from **day(l)**:

```
if [ $# -ne 3 ]; then
        echo 1>&2 Usage: $0 19 Oct 91
        exit 127
fi
```

Alternatively you may see:

```
if command; then command; [ else command; ] fi
```

4.4. While/Until Iteration

Repeat task while command returns good exit status.

```
{while | until} command
do
    command
done
```

For example, from **purge(l)**:

```
# for each argument mentioned, purge that directory

while [ $# -ge 1 ]; do
    _purge $1
    shift
done
```

Alternatively you may see:

```
while command; do command; done
```


4.5. Variables

Variables are sequences of letters, digits, or underscores beginning with a letter or underscore.

Numeric variables (eg. like $1, etc.) are positional variables for argument communication.

4.5.1. Variable Assignment

Assign a value to a variable by *variable*=*value*. For example:

```
PATH=/usr/ucb:/usr/bin:/bin; export PATH
```

or

```
TODAY=`(set \`date\`; echo $1)`
```

4.5.2. Exporting Variables

Variables are **not** exported to children unless explicitly marked. From **xdm/Xsession**:

```
# We MUST have a DISPLAY environment variable

if [ "$DISPLAY" = "" ]; then
        if tty -s ; then
                echo "DISPLAY (‘hostname‘:0.0)? \c";
                read DISPLAY
        fi
        if [ "$DISPLAY" = "" ]; then
                DISPLAY=‘hostname‘:0.0
        fi
        export DISPLAY
fi
```

Likewise, for variables like the **PRINTER** which you want honored by **lpr(1)**. From a users **.profile** (which we don't support):

```
PRINTER=PostScript; export PRINTER
```

Note: that the Cshell exports all environment variables.

4.5.3. Referencing Variables

Use **$variable** (or, if necessary, **${variable}**) to reference the value.

```
# Most user’s have a /bin of their own

if [ "$USER" != "root" ]; then
        PATH=$HOME/bin:$PATH
else
        PATH=/etc:/usr/etc:$PATH
fi
```

The braces are required for concatenation constructs.

$p_01
The value of the variable “p_01”.

${p}_01
The value of the variable “p” with “_01” pasted onto the end.

4.5.4. Conditional Reference

${*variable***-***word***}**

If the variable has been set, use it's value, else use *word*. From **xdm/Xsession**:

```
POSTSCRIPT=${POSTSCRIPT-PostScript};
        export POSTSCRIPT
```

${*variable***:-***word***}**

If the variable has been set and is not null, use it's value, else use *word*.

These are useful constructions for honoring the user environment. Ie. the user of the script can override variable assignments. Cf. programs like **lpr(1)** honor the **PRINTER** environment variable, you can do the same trick with your shell scripts.

${*variable***:?***word***}**

If variable is set use it's value, else print out *word* and exit. Useful for bailing out.

4.5.5. Arguments

Command line arguments to shell scripts are positional variables:

$0, $1, ...

The command and arguments. With **$0** the command and the rest the arguments.

$#

The number of arguments.

$*, $@

All the arguments as a blank separated string. Watch out for "$*" vs. "$@".

And, some commands:

shift

Shift the postional variables down one and decrement number of arguments.

set *arg arg ...*

Set the positional variables to the argument list.

Command line parsing uses **shift** (see also **getopt(1)**):

```
# parse argument list

while [ $# -ge 1 ]; do
        case $1 in
        process arguments...
        esac
        shift
done
```

A use of the **set** command (from **syslogd.daily**):

```
# figure out what day it is

TODAY=`(set \`date\`; echo $1)`

cd $SPOOL

for i in `cat $LOGS`
do
        mv $i $i.$TODAY
        cp /dev/null $i
        chmod 664 $i
done
```


4.5.6. Special Variables

$$

Current process id. This is very useful for constructing temporary files. From **calendar(1)**:

```
tmp=/tmp/cal0$$
trap "rm -f $tmp /tmp/cal1$$ /tmp/cal2$$"
trap exit 1 2 13 15
/usr/lib/calprog >$tmp
```

$?

The exit status of the last command. From **ctc(1)**:

```
$command
# Run target file if no errors and ...

if [ $? -eq 0 ]
then
        etc...
fi
```

4.6. Quotes/Special Characters

Special characters to terminate words:

```
; & ( ) | ^ < > new-line space tab
```

These are for command sequences, background jobs, etc. To *quote* any of these use a backslash (****) or bracket with quote marks (**""** or **’’**).

Single Quotes

Within single quotes *all* characters are quoted -- including the backslash. The result is one word.

```
grep :${gid}: /etc/group | awk -F: '{print $1}'
```

Double Quotes

Within double quotes you have variable subsitution (ie. the dollar sign is interpreted) but no file name generation (ie. ***** and **?** are quoted). The result is one word.

```
if [ ! "${parent}" ]; then
        parent=${people}/${group}/${user}
fi
```


Back Quotes

Back quotes mean run the command and substitute the output.

```
if [ "`echo -n`" = "-n" ]; then
        n=""
        c="\c"
else
        n="-n"
        c=""
fi
```

and

```
TODAY=`(set \`date\`; echo $1)`
```

4.7. Functions

Functions are a powerful feature that aren't used often enough. Syntax is

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

For example, from **purge(l)**:

```
# Purge a directory

_purge()
{
	# there had better be a directory

	if [ ! -d $1 ]; then
		echo $1: No such directory 1>&2
		return
	fi

	etc...
}
```


Within a function the positional parmeters $0, $1, etc. are the arguments to the function (not the arguments to the script).

Within a function use **return** instead of **exit**.

Functions are good for encapsulations. You can pipe, redirect input, etc. to functions. For example, from **addusers(l)**

```
# deal with a file, add people one at a time

do_file()
{
	while parse_one

	etc...
}

etc...

# take standard input (or a specified file) and do it.

if [ "$1" != "" ]; then
	cat $1 | do_file
else
	do_file
fi
```

4.8. Sourcing commands

You can execute shell scripts from within shell scripts. A couple of choices:

sh *command*
: This runs the shell script as a separate shell. For example, on Sun machines in **/etc/rc**

```
sh /etc/rc.local
```

. *command*
: This runs the shell script from within the current shell script. For example, on NeXT machine in **/etc/rc**

```
# Read in configuration information
. /etc/hostconfig
```

What are the virtues of each? What's the difference?

The second form is useful for configuration files where environment variable are set for the script.

For example, from **backup(l)**

```
for HOST in $HOSTS; do

 # is there a config file for this host?

 if [ -r ${BACKUPHOME}/${HOST} ]; then
         . ${BACKUPHOME}/${HOST}
 fi

etc...
```

Using configuration files in this manner makes it possible to write scripts that are automatically tailored for different situations.

5. Some Tricks

5.1. Test

The most powerful command is **test(1)**.

    if test *expression*; then

        etc...

and (note the matching bracket argument)

    if [*expression*]; then

        etc...

On System V machines this is a builtin (check out the command **/bin/test**).

On BSD systems (like the Suns) compare the command **/usr/bin/test** with **/usr/bin/[**.

Useful expressions are:

test { -w, -r, -x, -s, ... } *filename*
is file writeable, readable, executeable, empty, etc?

test *n1* **{ -eq, -ne, -gt, ... }** *n2*
are numbers equal, not equal, greater than, etc.?

test *s1* **{ =, != }** *s2*
Are strings the same or different?

test *cond1* **{ -o, -a }** *cond2*
Binary **or**; binary **and**; use **!** for unary negation.

For example

```
if [ $year -lt 1901  -o  $year -gt 2099 ]; then
        echo 1>&2 Year \"$year\" out of range
        exit 127
fi
```

Learn this command inside out! It does a lot for you.

5.2. String matching

The test command provides limited string matching tests. A more powerful trick is to match strings with the **case** switch.

```
# parse argument list

while [ $# -ge 1 ]; do
        case $1 in
        -c*)    rate=‘echo $1 | cut -c3-‘;;
        -c)     shift;  rate=$1 ;;
        -p*)    prefix=‘echo $1 | cut -c3-‘;;
        -p)     shift;  prefix=$1 ;;
        -*)     echo $Usage; exit 1 ;;
        *)      disks=$*;       break   ;;
        esac

        shift

done
```

Of course **getopt** would work much better.

5.3. SysV vs BSD echo

On BSD systems to get a prompt you’d say:

```
echo -n Ok to procede?;  read ans
```

On SysV systems you’d say:

```
echo Ok to procede? \c; read ans
```

In an effort to produce portable code we’ve been using:

```
# figure out what kind of echo to use

if [ "‘echo -n‘" = "-n" ]; then
        n="";   c="\c"
else
        n="-n";    c=""
fi

etc...

echo $n Ok to procede? $c; read ans
```

5.4. Is there a person?

The Unix tradition is that programs should execute as quietly as possible. Especially for pipelines, cron jobs, etc.

User prompts aren't required if there's no user.

```
# If there's a person out there, prod him a bit.

if tty -s; then
	echo Enter text end with \^D
fi
```

The tradition also extends to output.

```
# If the output is to a terminal, be verbose

if tty -s <&1; then
	verbose=true
else
	verbose=false
fi
```


Beware: just because stdin is a tty that doesn't mean that stdout is too. User prompts should be directed to the user terminal.

```
# If there's a person out there, prod him a bit.

if tty -s; then
	echo Enter text end with \^D >&0
fi
```

Have you ever had a program stop waiting for keyboard input when the output is directed elsewhere?

5.5. Creating Input

We're familiar with redirecting input. For example (from **addusers(l)**):

```
# take standard input (or a specified file) and do it.

if [ "$1" != "" ]; then
	cat $1 | do_file
else
	do_file
fi
```

alternatively, redirection from a file:

```
# take standard input (or a specified file) and do it.

if [ "$1" != "" ]; then
	do_file < $1
else
	do_file
fi
```


You can also construct files on the fly. From **signon(l)**:

```
rmail bsmtp <<EOF
helo news
mail from:<$1@newshost.uwo.ca>
rcpt to:<listserv@$3>
data
from: <$1@newshost.uwo.ca>
to: <listserv@$3>
Subject: Signon $2

subscribe $2 Usenet Feeder at UWO
.
quit
EOF
```

Note: that variables are expanded in the input.

5.6. String Manipulations

One of the more common things you'll need to do is parse strings. Some tricks

```
TIME=‘date | cut -c12-19‘

TIME=‘date | sed ’s/.* .* .* \(.*\) .* .*/\1/’ ‘

TIME=‘date | awk ’{print $4}’ ‘

TIME=‘set \‘date\‘; echo $4‘

TIME=‘date | (read u v w x y z; echo $x)‘
```


With some care, redefining the input field separators can help.

```
#!/bin/sh
# convert IP number to in-addr.arpa name

name()
{	set ‘IFS=".";echo $1‘
	echo $4.$3.$2.$1.in-addr.arpa
}

if [ $# -ne 1 ]; then
	echo 1>&2 Usage: bynum IP-address
	exit 127
fi

add=‘name $1‘

nslookup <<EOF | grep "$add" | sed ’s/.*= //’
set type=any
$add
EOF
```

5.7. Debugging

The shell has a number of flags that make debugging easier:

sh -n *command*
read the shell script but don’t execute the commands. Ie. check syntax

sh -x *command*
Display commands and arguments as they’re executed.

In a lot of my shell scripts you’ll see

```
# Uncomment the next line for testing
# set -x
```