Quick intro for HW

AWK/NAWK

awk :

[Aho, Kernighan, Weinberger]

new-awk = nawk

Powerful pattern-directed scanning and processing language.

So powerful that we will devote a lot of time to it in the future.

One of the most used Unix tools.

For now we will present the bare basics that will allow us to start processing data.

nawk reads a file and processes it a line at a time.

The input line is parsed into fields separated by spaces or tabs.

The fields are addressed as $1, $2, etc.

$0 is the while line.

Here is the basic syntax for performing simple nawk processing from the command line.

nawk '[/regex/...] {print $n, $m, ...}' file

Where /regex/ is an optional (the []s) regular expression contained within forward slashes "/"s. (There can be more than one {the ...}, combined logically &&, ||, !).

$n, $m, etc. are the columns of the file to print out

file specifies the input file.

A basic, and useful, nawk example.

Print out a number of columns of a file (say the lat and long for plotting by GMT).

Here is the file

```
CAT     YEAR  MO DA ORIG TIME    LAT    LONG    DEP  MAGNITUDE
PDE     1973  01 05 123556.50   46.47 -112.73
PDE     1973  01 07 225606.10   37.44  -87.30   15   3.2
PDE     1973  01 08 091136.80   33.78  -90.62    7   3.5
```

The lat and long are in columns 6 and 7, respectively. For now use an editor to remove the header line.

If a regular expression is specified, nawk will only print out the lines which contain a match to the regular expression.

```
%nawk '/good data/ {print $7, $6}' mydatafile.dat
```

Prints out the seventh and sixth column for all lines in the file mydatafile.dat containing the character string "good data".

```
%nawk ' {print $7, $6}' mydatafile.dat
```

Prints out the seventh and sixth column for all lines in the file mydatafile.dat.

The input to nawk can also be piped or redirected.

```
%cat mydatafile.dat | nawk ' {print $7, $6}'

%nawk ' {print $7, $6}' << END
`cat mydatafile.dat`
END
```

(Although one would never do either of the above in practice!! Why?)

This is enough awk/nawk knowledge to do the homework.

Some useful commands

MISC COMMANDS

finger - find out information about users
w/who - who is currently on the machine and what are they doing
whoami – reports username
id - tells you who you are (username, uid, group, gid)
uname - tells you basic information about the system.
whois - information about hosts
which/whereis - locates commands locate - locates files
whatis - gives brief summary of a command
talk - chat with other users on the system
write/wall - send messages to users of the system

Basic scripting

SHELL SCRIPTING

Interpreted vs. Compiled Languages

End members of methods for changing what you write in a high-level language into the individual machine instructions the computer's CPU executes.

Compiler

The name "compiler" is primarily used for programs that translate source code written in a high-level programming language to a lower level language such as assembly language or machine code (everything has to end up as machine code eventually).

Typically produces most efficient (in terms of run time) implementation of a program.

Source to executable with compiled language.

Several steps.

-Edit the source code.

-Compile the source code into machine code.
-Link the machine code with libraries, etc. (oftentimes done in one command together with compile, but not necessarily).

- Run the program (which is itself another file).

Examples of some languages that are typically compiled.

FORTRAN
C
C++
ALGOL
PASCAL
Visual BASIC

Interpreter

The name "interpreter" is primarily used for programs that translate source code written in a high-level programming language to machine code at the time of execution.

Typically produces less efficient (in terms of run time) implementation of a program (for example, in a loop it has to reinterpret the instructions each time through the loop).

Source to executable with interpreted language.

Two steps.

-Edit the source code.

-Run the program

(It does the "compiling" and "linking", or translating to machine code, steps automatically.)

Examples of some languages that are typically interpreted

Shell scripts (always)
BASIC
MATLAB

- Compiled languages are compiled.

- Modern interpreted languages are typically hybrids (they will compile the code in a loop for instance, instead of interpreting it each pass).

Some languages, such as MATLAB, do both. It has interpreted parts, compiled as needed parts, pre-compiled parts, and you can compile your code.

When you run a compiled program again you skip the compile/link steps.

(the compile/link process produces an executable file, which is the file that is run/executed - not the source file.)

When you run an interpreted program again the computer has to redo the interpretation each time.

(So while modern interpreters may internally take shortcuts such as compiling a loop, it is local to each running of the program. Each time you run/execute the interpreted program it returns to the source file and starts from scratch.)

Shell scripts are strictly interpreted.

The philosophy of shell scripting is to

- develop a tool with the shell and

- then write the final, efficient, implementation of the tool in C (or other high level language).

The second step is typically skipped.

Interpreting vs. Compiling.

Compiling: good for medium, large-scale, complicated problems, number crunching, when you need the efficiency.

Interpreting: good for smaller scale, simpler problems, when not number crunching, when your efficiency is more important than the CPU's.

(It is not worth spending an hour of your time to save a microsecond of execution time on a program that will run once.)

-Olden days -

Computer was expensive, limited, resource.
Programmer – relatively less expensive.
Lots of effort to write small, efficient programs.

-Today -

Abundance of inexpensive computer resources.
Programmer - very expensive.
(Buy another gigabyte of memory!)

- Take home lesson -

Use the appropriate tool.

What is a shell script?

It is a program that is written using shell commands
(the same things you type to do things in the shell).

When to use a shell script?

Shell scripts are used most often for combining existing programs to accomplish some small, specific job, typically one you want to run often/multiple times.

Once you've figured out how to get the job done, you put the commands into a file, or script, which you can then run directly.

Why use shell scripts?

- Repeatability -

why bother retyping a series of common commands?

Why use shell scripts?

- Portability -

Once you have a useful tool, you can move you shell script from one machine/flavor Unix to another.

POSIX standard - formal standard describing a portable operating environment.

IEEE Std 1003.2 current POSIX standard.

Why use shell scripts?

Simplicity

Simple example

```
%more 0.5/run.csh
mkdir 0.5_3
tomoDD2.pwave tomoDD.3.inp
mv red* 0.5_3
cp tomo* 0.5_3
mv Vp* 0.5_3

mkdir 0.5_20
tomoDD2.pwave tomoDD.20.inp
mv red* 0.5_20
cp tomo* 0.5_20
mv Vp* 0.5_20
```

Use more command to see what is in shell script (file) run.csh.

This csh script simply runs a series of tomographic inversions using different parameter setups.

Simple example

```
%more 0.5/run.csh
mkdir 0.5_3
tomoDD2.pwave tomoDD.3.inp
mv red* 0.5_3
cp tomo* 0.5_3
mv Vp* 0.5_3

mkdir 0.5_20
tomoDD2.pwave tomoDD.20.inp
mv red* 0.5_20
cp tomo* 0.5_20
mv Vp* 0.5_20
```

When we run the script, it runs the commands in the file - so it runs the program tomoDD2, and moves the output files to specially named directories.
It then does it again with a different input data set.

Simple example

```
%more 0.5/run.csh
mkdir 0.5_3
tomoDD2.pwave tomoDD.3.inp
mv red* 0.5_3
cp tomo* 0.5_3
mv Vp* 0.5_3

mkdir 0.5_20
tomoDD2.pwave tomoDD.20.inp
mv red* 0.5_20
cp tomo* 0.5_20
mv Vp* 0.5_20
```

The prep work allows us to save time and effort later.

Simple example

```
%more 0.5/run.csh
mkdir 0.5_3
tomoDD2.pwave tomoDD.3.inp
mv red* 0.5_3
cp tomo* 0.5_3
mv Vp* 0.5_3

mkdir 0.5_20
tomoDD2.pwave tomoDD.20.inp
mv red* 0.5_20
cp tomo* 0.5_20
mv Vp* 0.5_20
```

This is an example only.
If we really wanted to run the same program multiple times, we would write this as some sort of loop.
This way we would only write the commands once, and pass the info that changes to the commands.

Standard example

Create a file (typically with an editor), make it executable, run it.

```
%vim hello.sh
i#!/bin/bash
echo 'hello world.'
a=`echo "hello world." | wc`
echo "This phrase contains $a lines, words and
characters"<Esc>
:wq
%chmod ug+x hello.sh
%./hello.sh
hello.sh
hello world.
This phrase contains 1 2 13 lines, words and characters
%
```

(i and <Esc> etc. above in blue don't show up on screen.)

New Unix construct

-Command substitution -

Invoked by using the back or grave (French) quotes (actually accent, `).

```
a=`echo "hello world." | wc`
```

What is this?

Command substitution tells the shell to run what is inside the back quotes and substitute the output of that command for what is inside the quotes.

So the shell runs the commmand

```
echo hello world. | wc

(%echo hello world. | wc
1 2 13)
(why is it 2 lines and 13 characters?)
```

takes the output (the "1 2 13" above), substitutes it for what is in the back quotes (*echo "hello world.*), and sets the shell variable equal to it

```
a=`echo "hello world." | wc`
```

does this (is as if you typed)

```
a='1 2 13'
```

Compare/contrast this to what the pipe ("|") does.

This is a very useful and powerful feature.

It completes the standard set of Unix features.

The #! in the first line (known as shebang).

The first line can be used to tell the system what language (shell) to use for command interpretation.

It is a very specific format

```
#!/bin/bash
```

or

```
#!/bin/csh –f
```

(the -f is optional - for "fast" initialization - see man page (-f Fast start. Reads neither the .cshrc file, nor the .login file (if a login shell) upon startup.))

If you want your shell script to use the same shell as the parent process you don't need to declare the shell with the shebang at the beginning.

BUT

You can't put a comment (indicated by #) in the first line.

So the first line has to be one of

#!/shell_to_use

or

command (not comment, not "/shell_to_use")

Scripting Etiquette

Most scripts are read by both a person and a computer.

Don't ignore the person using or revising your script (most likely you 6 months later – when you will not remember what you did, or why you did it that way.)

Advice

1. Use comments to tell the reader what they need to know. The # denotes a comment in bash and csh.
2. Use indentation to mark the various levels of program control. (loops, if-then-else blocks)
3. Use meaningful names for variables and develop a convention that helps readers identify their function.
4. Avoid unnecessary complexity…keep it readable.

Usually you will find the obvious stuff will be commented and described fully.

The stuff the original author did not understand that well – but somehow got to work – will generally not be commented (or usefully commented, may even be wrong).

Header

Adding a set of comments at the beginning that provides information on

1. Name of the script
2. How the script is called
3. What arguments the script expects
4. What does the script accomplish
5. Who wrote the script and when
6. When was it revised and how

```
#!/usr/bin/bash -f
#Script: prepSacAVOdata.pl
#Usage:  $script <unixDir> <dataDir> <staFile> <phaseFile> <eventFile>
#-------------------------------------
#Purpose: To prepare SAC AVAO data for further processing
#    (1) generate event information file and add the event info
#          (name, event location) to the SAC headers
#    (2) generate event phase file and add the phase info
#          (time and weights) to the SAC headers
#Original Author (prepSacData.pl: Wen-xuan Du, Date: Mar. 18, 2003
#            Modified: May 21, 2004
#
#Last Modified by Heather DeShon  Nov. 30, 2004
#       A) Reads AVO archive event format directly (hypo71):
#              subroutines rdevent and rdphase
#       B) Reads SAC KZDATA and KZTIME rather than NZDTTM, which is
#              not set in AVO SAC data
. . .
```

Do (at least simple) error checking of call and print some sort of message for error.

```
if [ $# -ne 5 ]
then
        printf "Usage:\t\t$script <unixDir> <dataDirList>\
<staFile> <phaseFile> <eventFile>\n"
        printf "<unixDir>:\tdirectory in the unix system\
where pick file is stored;\n"
        printf "<dataDirList>:\tlist of data directories\
under  <unixDir>; 'dataDir.list';\n"
        printf "<staFile>:\tstation file;\n"
        printf "<phaseFile>:\tphase info file for program\
'ph2dt';\n"
        printf "<eventFile>:\tevent info file (one line for\
 one event);\n"
exit (-1)
fi
```

if does everything between "then" and "fi" (green box) if…

Check there are 5 input parameters.

$ to have shell return the value of a variable.

is the shell variable (shell gives you this variable when it starts a script) that contains number of parameters on input line.

-ne is the numerical test for not equal

(-eq is the numerical test for equal).

```
if [ $# -ne 5 ]
then
. . .
fi
```

So if the number of input parameters is not 5, it will do what is between "then" and "fi".

One of the things done in the error processing (in the green box) is the command "exit(-1)".

```
if [ $# -ne 5 ]
then
        . . .
        exit (-1)
fi
```

This returns a message, a numeric "return value" (in this case a -1) to the parent process.

NOTE

The formatting with respect to spaces, and lines of the "if [ ]", "then", ("else"), "fi" are very specific.

```
if [ . . . ]
then
      . . .
fi
```

It has to be written exactly as above - where the test and code to be performed replace the ". . ." . (look back at previous slide to see where spaces go)

The parent process can access this return value using the shell variable ?
(to obtain the value one uses $? (of course)).

This allows the parent process to get information about what happened in the daughter process.

This information can be used to control the execution of the parent process.
(does the parent process continue, quit, try to fix it, etc.?)

Many programs return a value of 0 (zero) upon successful completion.

From the ls man page -

```
EXIT STATUS
     0      All information was written successfully.

     >0     An error occurred.
```

So we can tell if it terminated successfully (but not what the error was if not).

Types of commands

Built-in commands: commands that the shell itself executes.

Shell functions: self-contained chunks of code, written in the shell language, that are invoked in the same way as a command.

External commands: commands that the shell runs by creating a separate process.

Variables

A variable is used to store some piece of (typically character string) information.

The $ tells the shell to return the value of the specified variable.

csh example

```
%set b = "Hello world."
%set a = `echo $b | wc`
%echo $a
1  2  13
```

bash example

```
%b="Hello world."
%a=`echo $b | wc`
%echo $a
1  2  13
```

cs/csh and sh/bash have different syntax for assigning the value of a shell variable.

(in bash cannot have spaces on either side of the equals sign, csh does not care, works with our without spaces.)

Constants

A constant is used to store some piece of (typically character string) information that is not expected to change.

In bash, variables are made constants by using the readonly command.

bash example

```
%x = 2
%readonly x
%x = 4
```

bash: x: readonly variable

Quote syntax

‘ . . .’: Single quotes tell the shell to take everything within the quotes (the open/close distinction does not show up in the editor/shell script) without interpretation and pass it on literally (as is).

(Double quotes do the same, but the shell expands variables and command substitutions.)
(This is the standard quote behavior - Unix think.)

Quote syntax

What happens if you forget the quotes depends on the shell.

In csh/tcsh, the variable b below would be set to Hello, and the shell would ignore the string world.

```
alpaca.ceri.memphis.edu545:> echo $0
-tcsh
alpaca.ceri.memphis.edu546:> set b=hello world
alpaca.ceri.memphis.edu547:> echo $b
hello
```

In sh/bash, the variable b below would be set to Hello, and the shell would try to run the command world.

```
alpaca.ceri.memphis.edu501:> /bin/sh
$ b='hello world'
$ echo $b
hello world
$ b=hello world
world: not found
$
```

Single vs. double quotes.

```
alpaca.ceri.memphis.edu503:> set a = A
alpaca.ceri.memphis.edu504:> echo $a
A
alpaca.ceri.memphis.edu505:> set b = 'letter $a'
alpaca.ceri.memphis.edu506:> echo $b
letter $a
```

Shell did not expand variable a to its value. Treated literally ($a).

```
alpaca.ceri.memphis.edu507:> set c = "letter $a"
alpaca.ceri.memphis.edu508:> echo $c
letter A
alpaca.ceri.memphis.edu509:>
```

Shell expanded variable a to its value, A, and passed value on.

Works same in csh/tcsh and sh/bash.

Not limited to use in shell scripts, can also use in shell.

`. . .`: backquotes are used for command substitution.

The . . . between the quotes above represent a command.

The output of backquotes can go into a variable, switch, redirected input (<<), etc.

Cannot go into a pipe (why?).

Trivial example of substituting into a switch.

```
alpaca.ceri.memphis.edu509:> set options='-la'
alpaca.ceri.memphis.edu510:> ls $options
total 2203891
-rw-rw-rw-  1 rsmalley user 54847 Mar  7  2009 CHARGE-2002-107
drwxr-xr-x 94 rsmalley user 31232 Sep 19 17:46 .
dr-xr-xr-x  3 root     root     3 Sep 19 17:51 ..
drwxr-xr-x 2 rsmalley user    512 Oct  1  2004 .acrobat
-rw-r--r-- 1 rsmalley user    237 Oct  1  2004 .acrosrch

alpaca.ceri.memphis.edu511:> ls `echo $options`
total 2203891
-rw-rw-rw-  1 rsmalley user 54847 Mar  7  2009 CHARGE-2002-107
drwxr-xr-x 94 rsmalley user 31232 Sep 19 17:46 .
dr-xr-xr-x  3 root     root     3 Sep 19 17:51 ..
drwxr-xr-x  2 rsmalley user   512 Oct  1  2004 .acrobat
-rw-r--r--  1 rsmalley user   237 Oct  1  2004 .acrosrch
```

Variables are set to the final output of all commands within the back single quotes.

```
alpaca.ceri.memphis.edu509:> set options='-la'
alpaca.ceri.memphis.edu513:> set ops = `echo $options`
alpaca.ceri.memphis.edu514:> echo $ops
-la
alpaca.ceri.memphis.edu515:>
```

Reading command line arguments.

You can send your script input from the command line just like you do with built-in commands. It also gets environment variables from the shell.

```
alpaca.ceri.memphis.edu517:> vi hi.sh
"hi.sh" [New file]
i #!/bin/bash
echo Hello, my name is $HOST. Nice to meet you $1.<Esc>
:wq
"hi.sh" [New file] 2 lines, 63 characters
alpaca.ceri.memphis.edu518:> x hi.sh
alpaca.ceri.memphis.edu519:> hi.sh Bob
Hello, my name is alpaca.ceri.memphis.edu. Nice to meet you
Bob.
alpaca.ceri.memphis.edu520:>
```

think of the command line as an array whose index starts with 0.
When you enter

```
%command arg1 arg2 arg3 arg4 . . . arg10 arg11 . . . Arg_end
```

The shell produces the following array that is passed to the shell script.

array[0]=command
array[1]=arg1
array[2]=arg2
. . .
array[end]=arg_end

Within the script, access to this array is accomplished using the syntax $n, where n is the array index.

$0=command
$1=arg1
$2=arg2
. . .
$9=arg9
${10}=arg10
${11}=arg11

note the format for numbers ≥10, the braces are required (they are optional for numbers ≤9)

Remember the discussion of identifying the shell you are running?

```
%echo $0
```

The shell is (just) a program.

Your shell receives these variables from its parent process, just like any other program.

So apply Unix think.

Reading user input

(even though it goes against the grain of Unix filter/think philosophy)

read: reads screen input into the specified variable.

Script - introduce.sh

```
#!/bin/bash
echo Please, enter your firstname and lastname
read FN LN
echo "Hi! $FN, $LN !"
```

Running it

```
alpaca.ceri.memphis.edu528:> introduce.sh
Please, enter your firstname and lastname
Bob Smalley
Hi! Bob Smalley !
alpaca.ceri.memphis.edu529:>
```

Reading (sucking in) multiple lines.
Use the syntax "<< eof".

Where eof defines the (character string) end-of-file delimiter.

This syntax redirects standard-in to the shell script (or the terminal if you are typing) until it finds the characters specified in the eof field.
(You have to be sure those characters are not in the file/text being sucked in – else it will stop there.)

Example.

File - my_thoughts.dat

```
I have a thousand thoughts in my head
and one line of text is not enough to get them
all out.  Hello world.
```

Script - suckitin.sh

```
#!/bin/bash
cat << END
`cat my_thoughts.dat`
END
```

Run it

```
alpaca.ceri.memphis.edu540:> suckitin.sh
I have a thousand thoughts in my head
and one line of text is not enough to get them
all out.  Hello world.
alpaca.ceri.memphis.edu541:>
```

Note - we would never program something this way.

We could have just done

```
alpaca.ceri.memphis.edu540:> cat my_thoughts.dat
```

But we are trying to demonstrate how command substitution works.

How does this script work?

```
#!/bin/bash
cat << END
`cat my_thoughts.dat`
END
```

The cat command reads standard-in, which is redirected, by the <<, to the lines that follow in the shell script (or the keyboard if not in a shell script).

We then use command substitution to produce input to the cat command from the file my_thoughts.dat.
Finally we terminate the input redirection with the string "END"

This is a very powerful way to process data.

```
my_processing_program << END
`my_convert_program input file1`
`cat input file2`
END
```

If we only needed to process file 1 or file2, we could have used a pipe

```
my_convert_program input file1 | my_processing_program
cat input file2 | my_processing_program
```

But there is no way (we have seen so far) to pipe both outputs into the program (the pipe is serial, not parallel).

Another example

```
my_processing_program << END
class example
10.3
41
`my_convert_program input file1`
`cat input file2`
END
```

Here we have a character string input,
"class example",
some numbers,
followed by the other data.

Again we can not use a pipe.

(Also notice that, following the Unix philosophy, the program is not "interactive", it is not prompting for the inputs. You have to know what it wants and how it wants it.)

Another example

```
my_processing_program inputvari1 inputvari2 << END
echo $1
class example
10.3
41
`my_convert_program input file1`
`cat input file2`
echo $2
END
```

Now we have added two inputs from the command line.

further examples: command substitution in conjunction with the gmt psxy command

```
#!/bin/sh
#missing beginning  and end of script. This command alone will
not work
psxy -R$REGN -$PROJ$SCALE $CONT -W1/$GREEN << END >> $OUTFILE
-69.5 -29.5
-65 -29.5
-65 -33.5
-69.5 -33.5
-69.5 -29.5
`cat my_map_file.dat`
END
```

This will read the data between the psxy command and the END and plot it on the map that is being constructed (the redirected, appended output).

further examples of <<: running sac from within a script.

```
# Script to pick times in sac file using taup
#  Usage: picktimes.csh [directory name]
#
sacfile=$1

sac << EOF >&! sac.log
r $sacfile
sss
traveltime depth &1,evdp picks 1 phase P S Pn pP Sn sP sS
qs
w over
q
EOF
```

Something new, what does >&! mean?

What does >&! mean?

We have already seen the > (it means redirect output) and ! (it means clobber any existing files with the same name).

So far we have discussed standard-in and standard-out.

But there is another standard output stream - introducing

standard-error.

```
carpincho:ESCI7205 smalley$ ls nonexitantfile
ls: nonexitantfile: No such file or directory
```

The message above shows up on the screen, but is actually standard-error, not standard-out.

```
carpincho:ESCI7205 smalley$ ls nonexitantfile > filelist
ls: nonexitantfile: No such file or directory
carpincho:ESCI7205 smalley$ ls -l filelist
-rw-r--r--  1 smalley  staff  0 Sep 21 16:01 filelist
carpincho:ESCI7205 smalley$
```

Can see this by redirecting standard-out into a file. The error message still shows up and the file with the redirected output is empty.

(it has 0 bytes, our standard Unix output, ready for the next command in pipe.)

>& is the csh/tcsh syntax for redirecting both standard-out and standard-error.

(else standard-error it goes to the screen)

Append standard-out and standard-error

>>&

You can't handle standard-error alone.

(With what we have seen so far, in csh/tcsh.)

In tcsh the best you can do is (Unix think)

(command > stdout_file) >& stderr_file

which runs "command" in a subshell.

stdout is redirected inside the subshell to stdout_file.
both stdout and stderr from the subshell are redirected to stderr_file, but by this point stdout has already been redirected to a file, so only stderr actually winds up in stderr_file.

Subshells can be used to group outputs together into a single pipe.

sh/bash

```
(command 1;command 2; command 3) | command
```

(when a program starts another program [more exactly, when a process starts another process], the new process runs as a subprocess or child process. When a shell starts another shell, the new shell is called a *subshell*.)

So in our earlier example using command substitution we could have done

```
(my_convert_program input file1; cat input file2) |\
my_processing_program << END
```

Where we are using the \ to continue the command on the second line.

The semi-colon ";", allows us to enter multiple commands, to be executed in order, in the sub-shell

(In typical Unix fashion, the ";" works in the shell and shell scripts also. Try it.).

The sh/bash syntax uses 1 to [optionally] identify standard-out and 2 to identify standard-error.

To redirect standard-error in sh/bash use

```
2>
```

To redirect standard-error to standard-out

```
2>&1
```

(! has usual meaning - clobber)

To pipe standard-out and standard-error

```
2>&1|
```

Redirect standard-error to file

```
$ls nonexitantfile > filelist 2> errreport
$cat errreport
ls: nonexitantfile: No such file or directory
$
```

Redirect standard-error to standard-out into a file. Can't do second redirect to a file. Use subshell command format, redirect output subshell to file. combofile has both standard-out and standard-error.

```
$(ls a.out nonexistantfile 2>&1)>combofile
$more combofile
nonexistantfile: No such file or directory
a.out
$
```