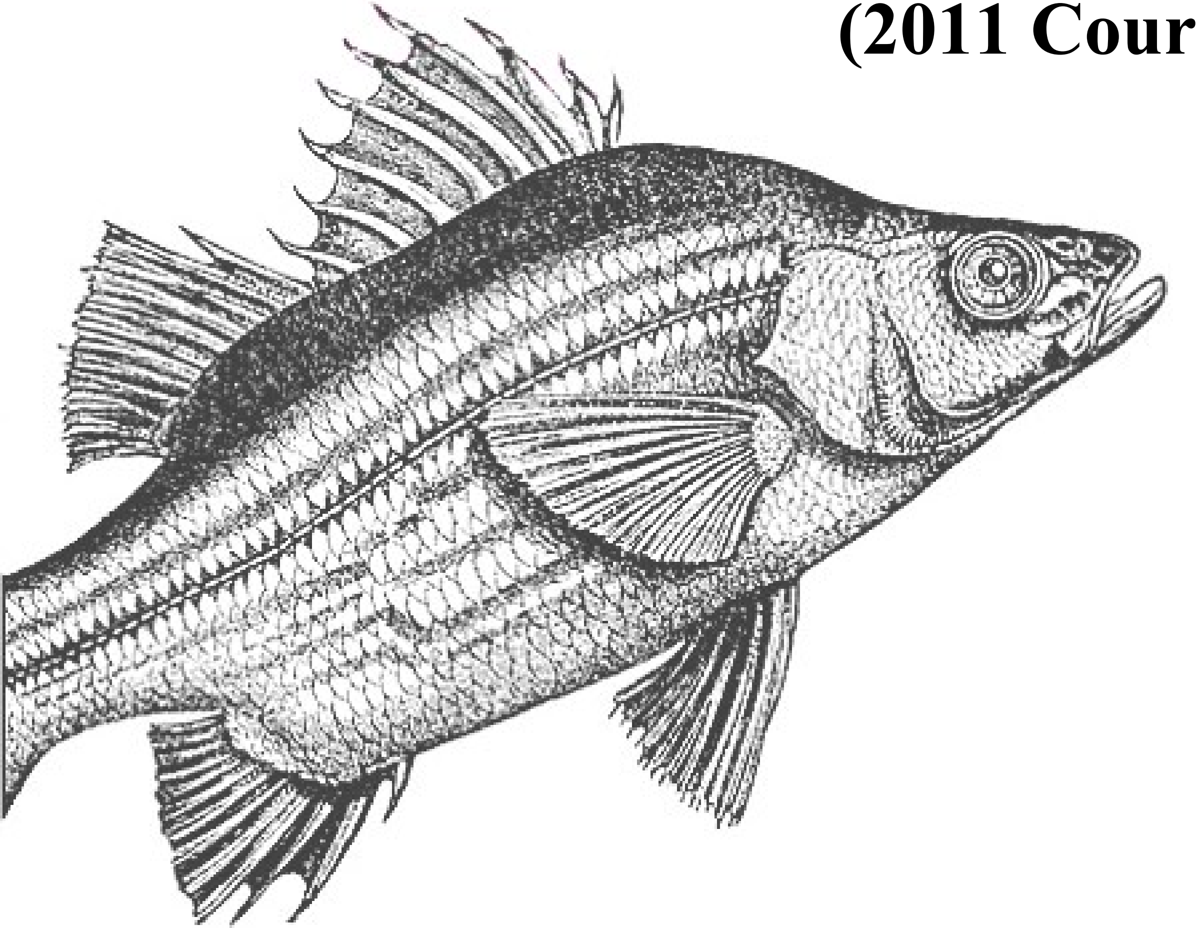
Introduction to Linux Extra Material (Part 2): Shell Scripting

(2011 Course)

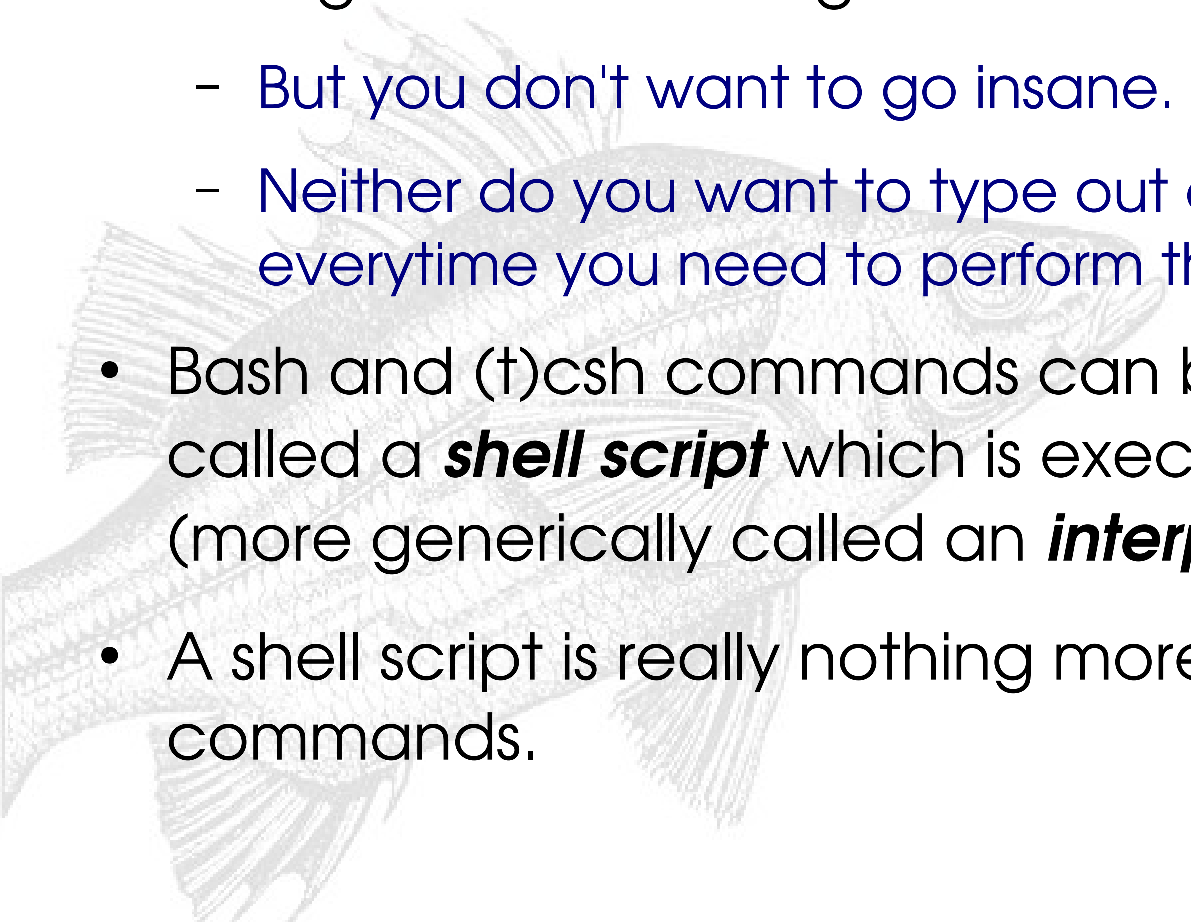


Tom Latham
Ben Morgan

THE UNIVERSITY OF WARWICK

Shell Scripts

- We saw earlier how basic commands could be linked and chained.
- In theory, one could perform most tasks by just piping, linking and redirecting.
  - But you don't want to go insane.
  - Neither do you want to type out a chain of 6 commands everytime you need to perform that one repetitive task.
- Bash and (t)csh commands can be written into a text file called a ***shell script*** which is executed by a given shell (more generically called an ***interpreter***).
- A shell script is really nothing more than a sequence of shell commands.



Hello World!

- In bash, the classic first program is written:

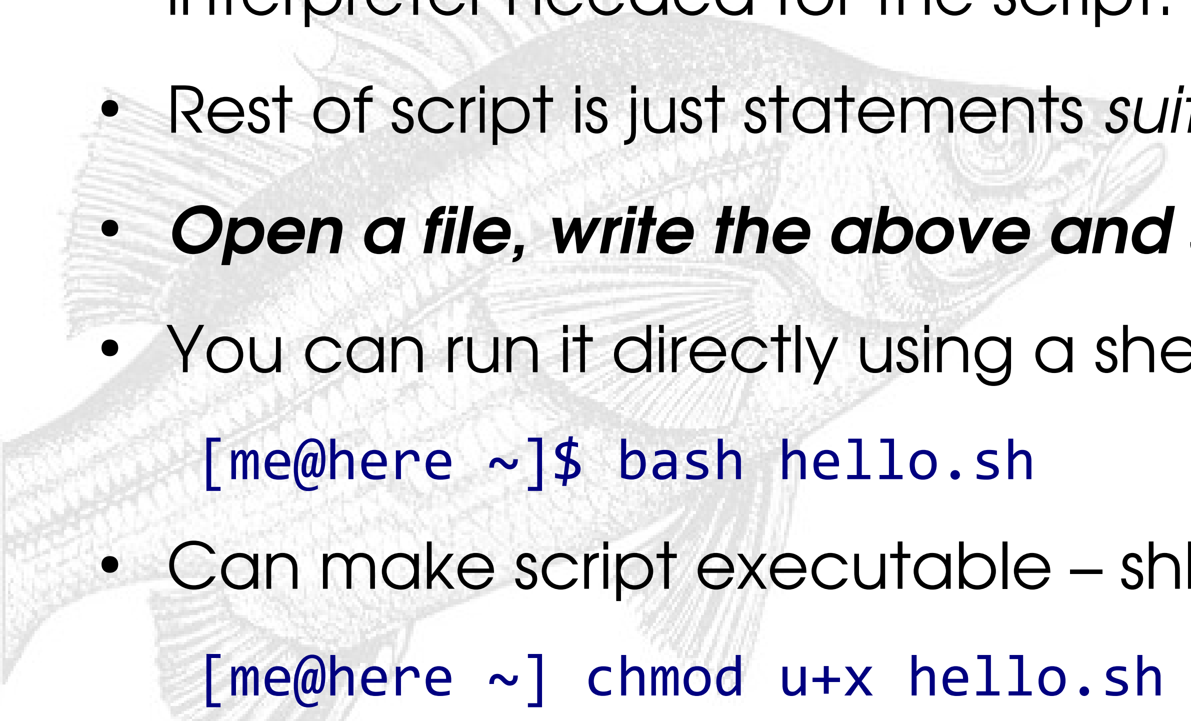
```
#!/bin/bash
echo “Hello world!”
```

- The first line specifies the interpreter to use. The `#!` is known as the 'shbang' and is followed by the full path to the interpreter needed for the script.
- Rest of script is just statements *suitable for that interpreter*.
- ***Open a file, write the above and save the file as hello.sh.***
- You can run it directly using a shell:

```
[me@here ~]$ bash hello.sh
```

- Can make script executable – shbang controls interpreter

```
[me@here ~] chmod u+x hello.sh
[me@here ~] ./hello.sh
```



Hello World! in (t)csh

- Because the shbang controls the interpreter, you could
  - Use bash day to day, write your scripts in csh.
  - Or vice versa.
- For instance, copy `hello.sh` to `hello.csh`, and modify it as follows:

```
#!/bin/tcsh
set message = “Hello world!”
echo $message
```

- Running it as

```
[me@here ~]$ tcsh hello.csh
```

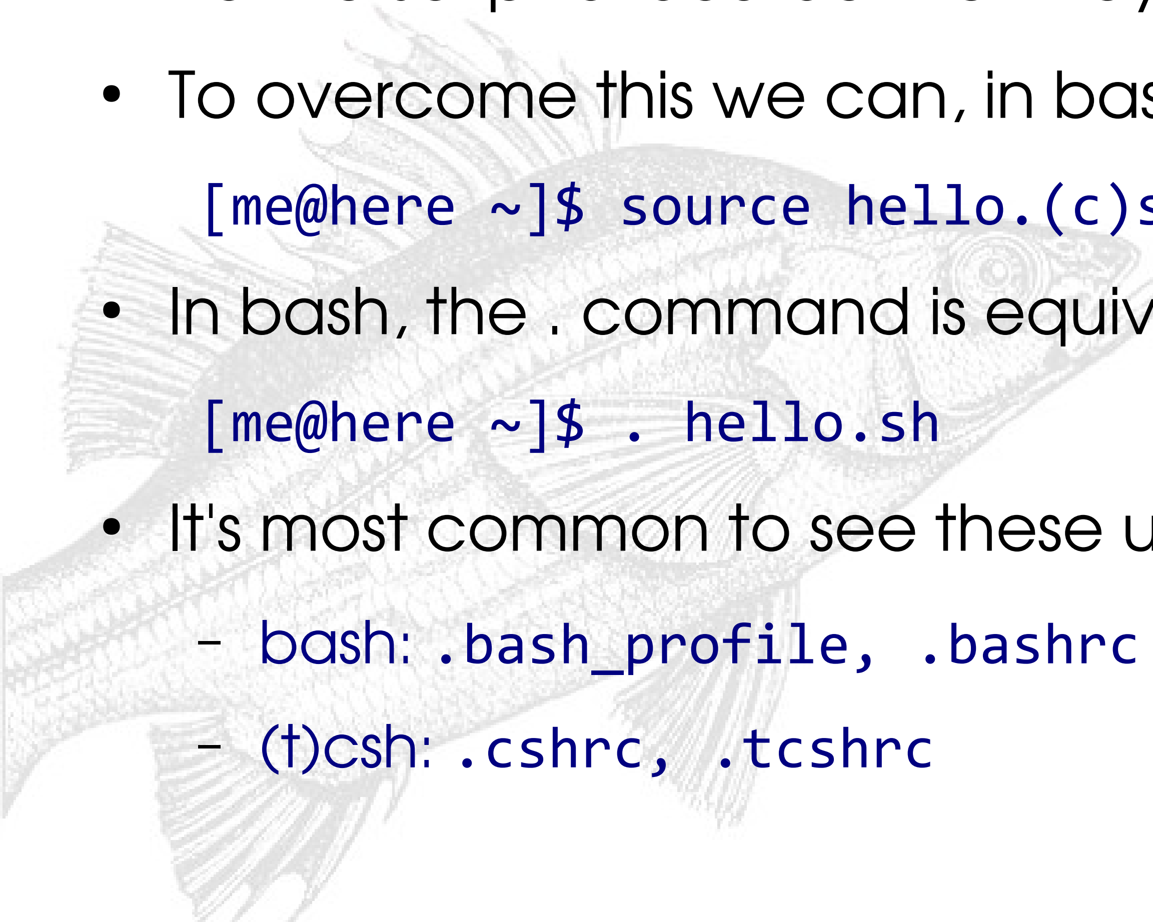
- works, but even if you're in a bash shell, you can also do

```
[me@here ~]$ ./hello.csh
```

Aside:
A fairly general practice is to give (shell) script files an extension matching the interpreter, e.g.

```
    .sh  for sh family
    .csh for csh family
    .py  for python
    .pl  for perl
```

`source` and `.`



- Running a script as an executable or argument to a shell ***executes commands in a subshell***.
- This means you can't affect current Environment Variables from a script executed that way.
- To overcome this we can, in bash AND (t)csh, use source

```
[me@here ~]$ source hello.(c)sh
```

- In bash, the . command is equivalent

```
[me@here ~]$ . hello.sh
```

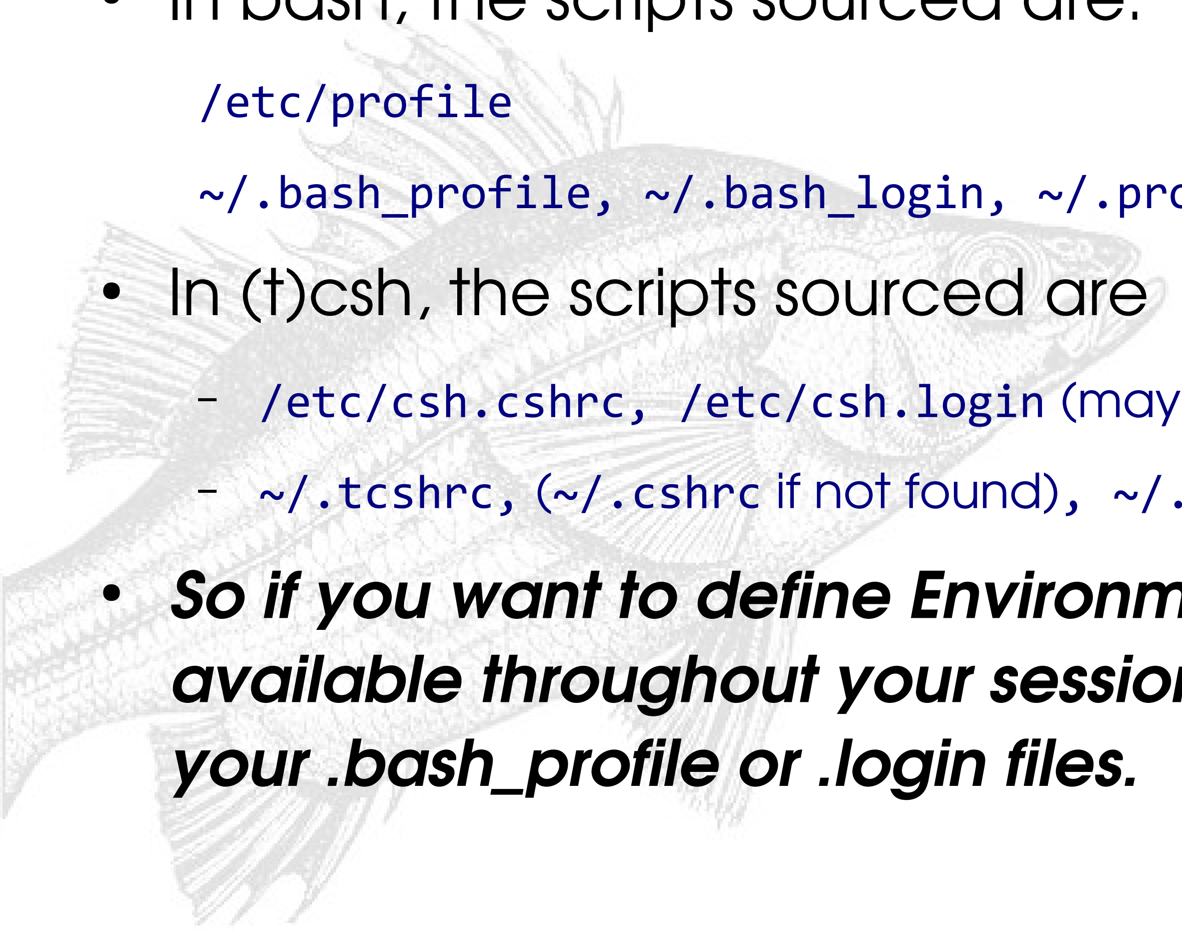
- It's most common to see these used in the 'login' scripts
  - bash: `.bash_profile, .bashrc`
  - (t)csh: `.cshrc, .tcshrc`

Login scripts

- A 'login shell' is a shell you obtain after authenticating to the system.
  - e.g. From graphical or virtual terminal login.
- In bash, the scripts sourced are:

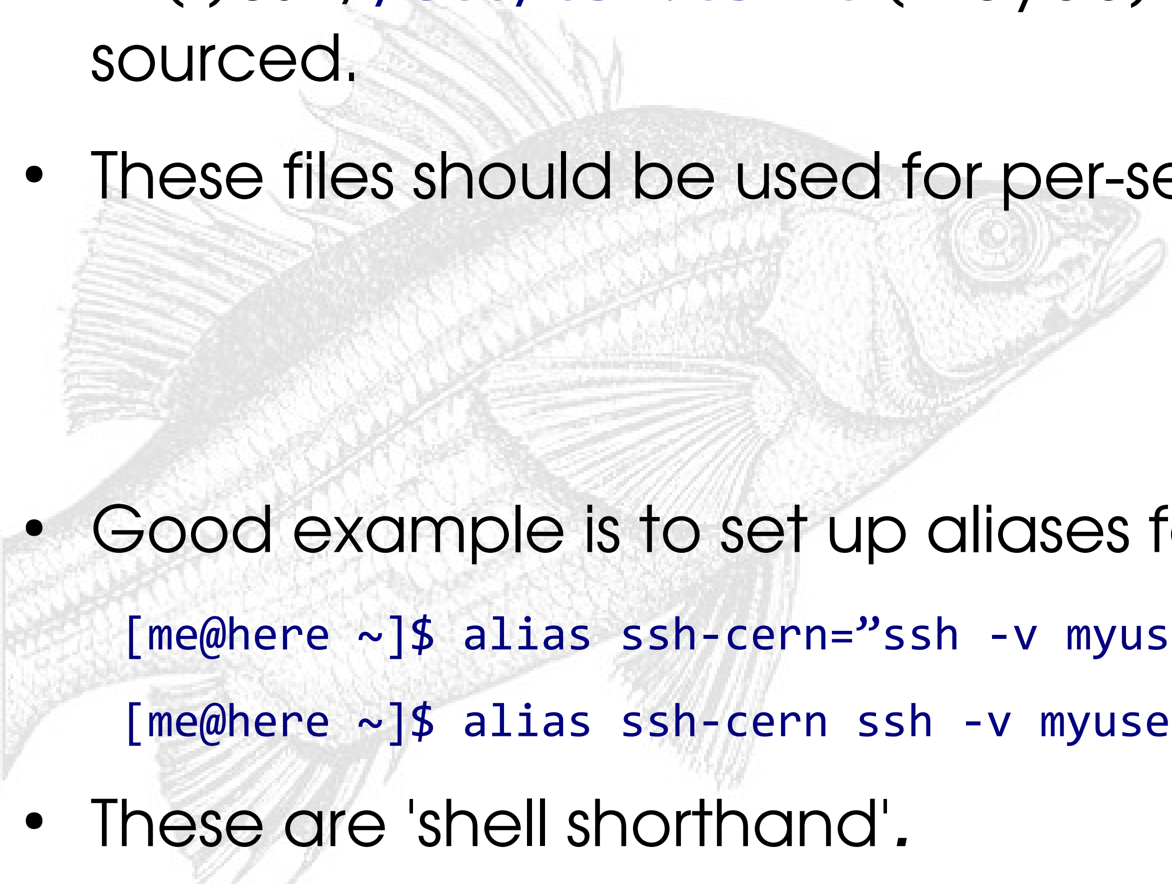
  `/etc/profile`

  `~/.bash_profile, ~/.bash_login, ~/.profile` (1st of these found readable)
- In (t)csh, the scripts sourced are
  - `/etc/csh.cshrc, /etc/csh.login` (maybe)
  - `~/.tcshrc,` (`~/.cshrc` if not found), `~/.history, ~/.login, ~/.cshdirs`
- ***So if you want to define Environment Variables that will be available throughout your session, you should define them in your .bash_profile or .login files.***



Startup scripts

- In a non-login ***interactive*** shell, e.g. a terminal started in the GUI, running of scripts may be different.
- In bash, only `.bashrc` is sourced.
- In (t)csh, `/etc/csh.cshrc` (maybe) and `.tcshrc` or `.cshrc` are sourced.
- These files should be used for per-session tasks.



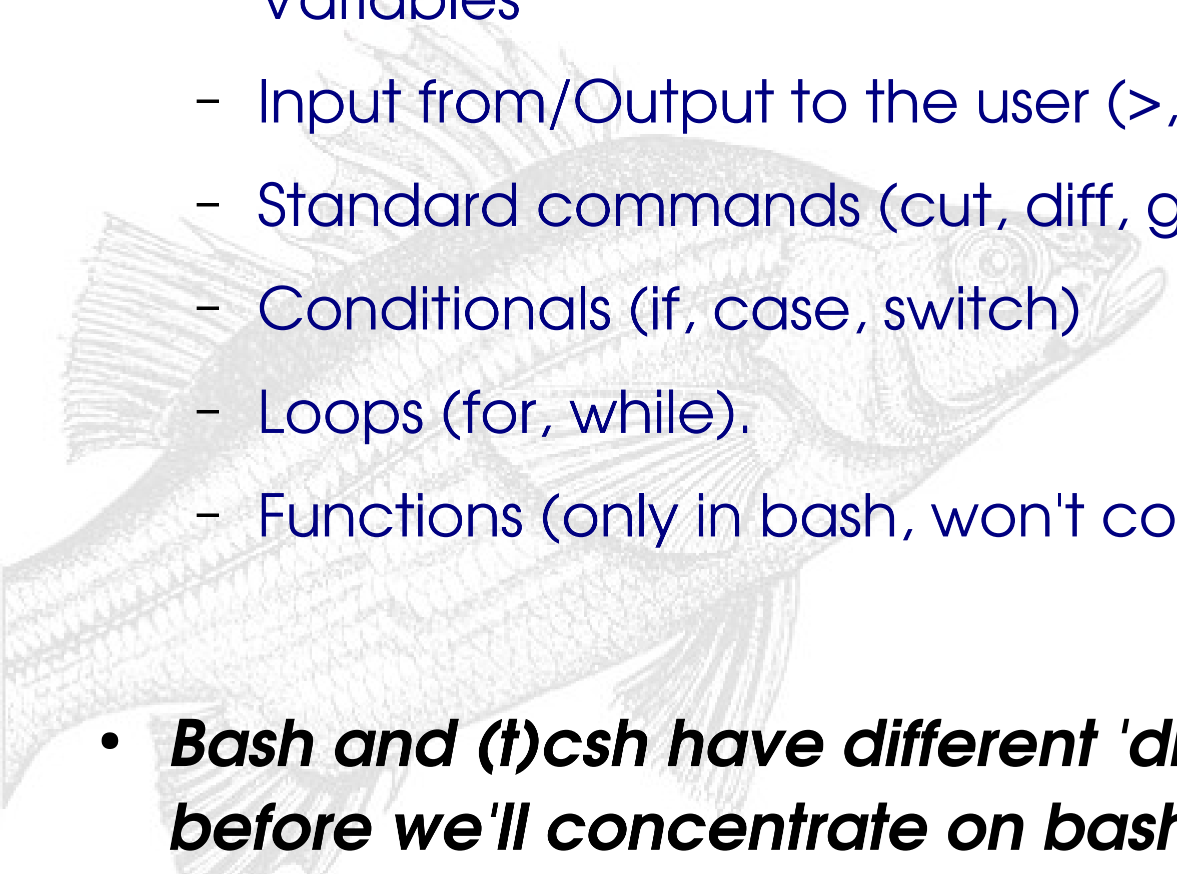
- Good example is to set up aliases for commands

```
[me@here ~]$ alias ssh-cern=”ssh -v myusername@lxplus.cern.ch”
[me@here ~]$ alias ssh-cern ssh -v myusername@lxplus.cern.ch
```

- These are 'shell shorthand'.

Script Breakdown

- 'Shell Scripting' might be better titled 'Shell Programming'
- We have all (well, almost all) the functionality of structured programming:
  - Variables
  - Input from/Output to the user (>, < etc)
  - Standard commands (cut, diff, grep, sed, and so on)
  - Conditionals (if, case, switch)
  - Loops (for, while).
  - Functions (only in bash, won't consider these here).



- ***Bash and (t)csh have different 'dialects' for these, so as before we'll concentrate on bash, but highlight the differences.***

Bash Variables

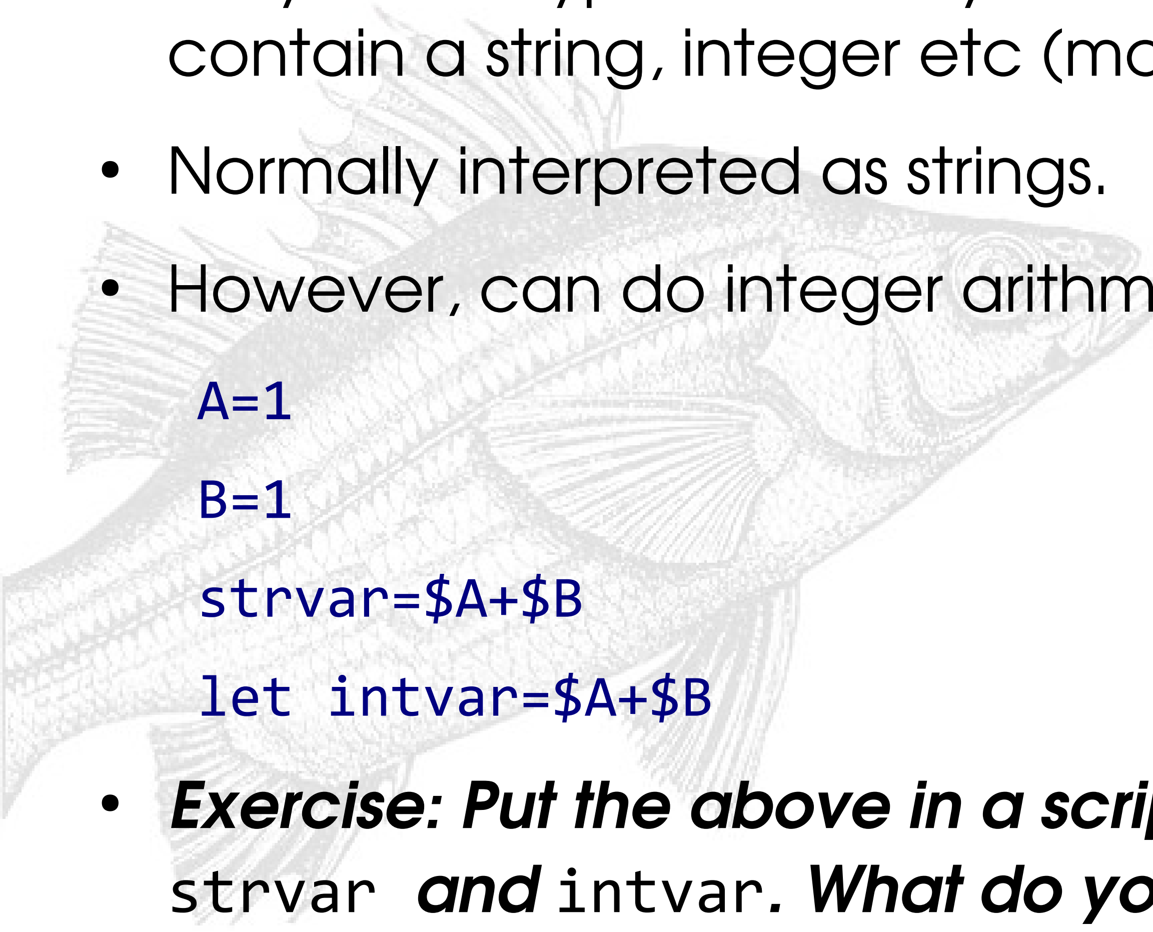
- Variables in bash are just the shell variables we saw last time

```
myVar=”hello”
anotherVar=$myVar
```

Note use of $ to obtain value.

- They are untyped, i.e. they don't specify whether they contain a string, integer etc (more on 'type' in C++ later).
- Normally interpreted as strings.
- However, can do integer arithmetic using the `let` keyword

```
A=1
B=1
strvar=$A+$B
let intvar=$A+$B
```

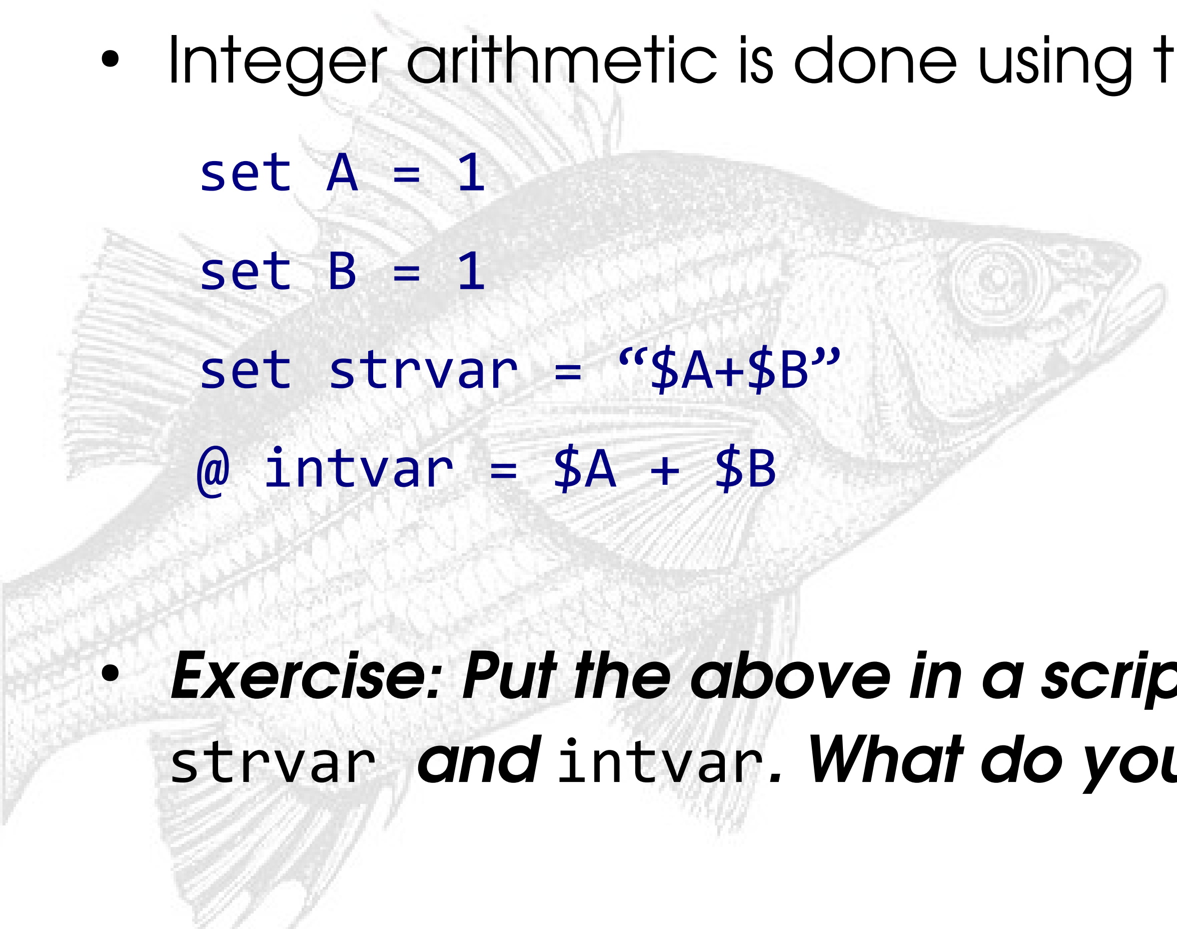


- ***Exercise: Put the above in a script and output the values of*** `strvar` ***and*** `intvar`. ***What do you notice?***

(t)csh Variables

- Variables in (t)csh are also the same as the (t)csh shell variables we saw before.
- Like bash variables, they are untyped and usually interpreted as strings.
- Integer arithmetic is done using the `@` prefix

```
set A = 1
set B = 1
set strvar = “$A+$B”
@ intvar = $A + $B
```



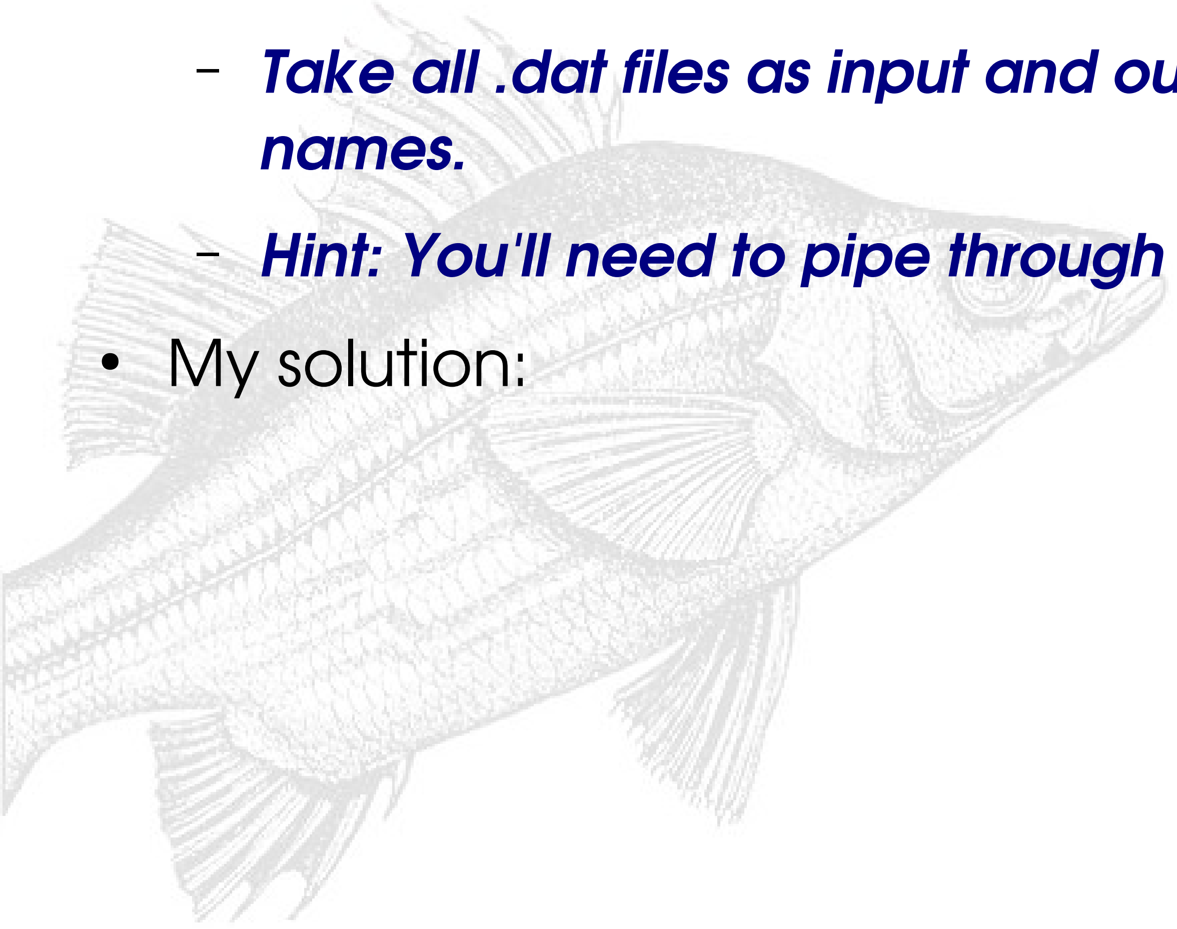
- ***Exercise: Put the above in a script and output the values of*** `strvar` ***and*** `intvar`***. What do you notice?***

Command Substitution

- We can assign the result of a command to a variable using backticks, like so

```
myFiles=`ls *.dat`
```

- ***Exercise:***
  - ***Take all .dat files as input and output a list of the unique particle names.***
  - ***Hint: You'll need to pipe through*** `sort` ***and then*** `uniq`***...***
- My solution:

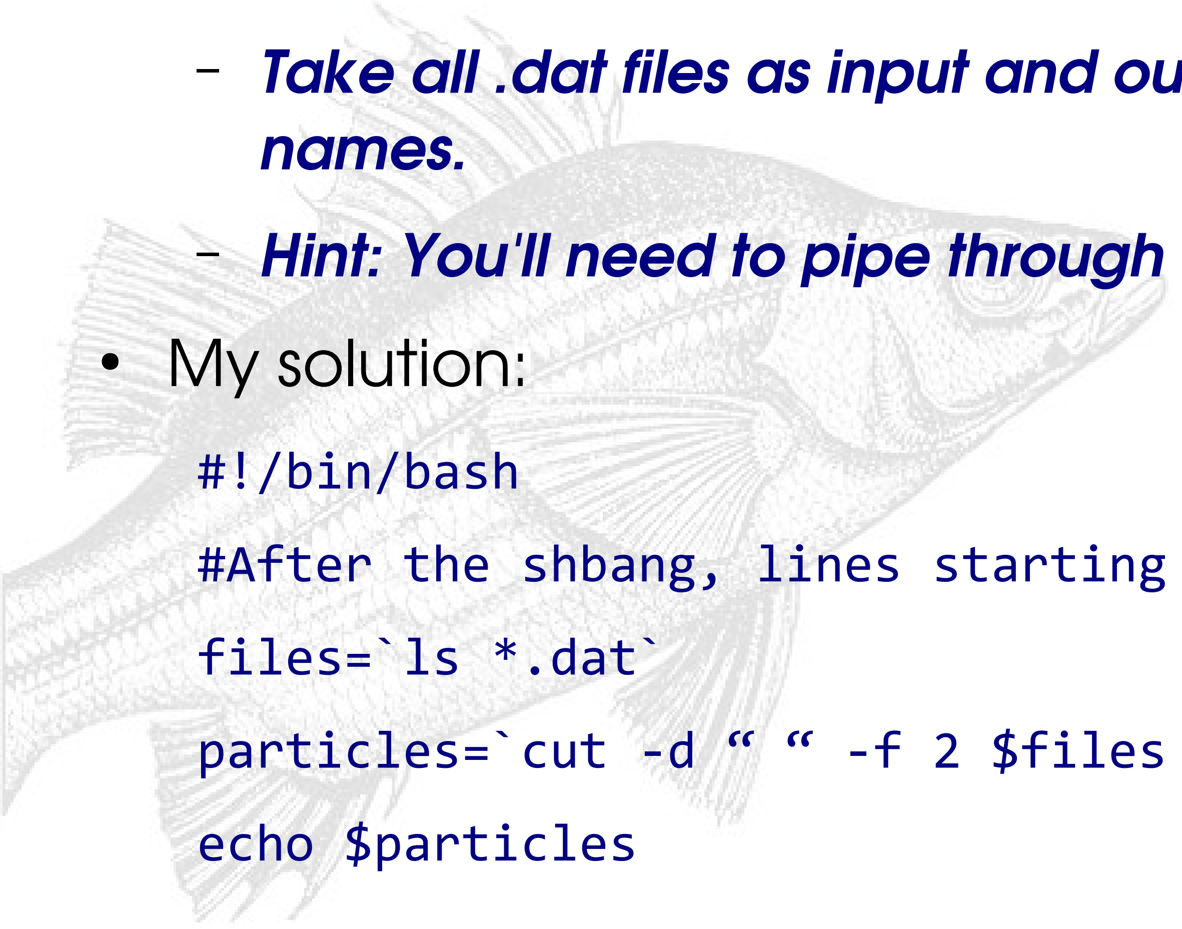


Command Substitution

- We can assign the result of a command to a variable using backticks, like so

```
myFiles=`ls *.dat`
```

- ***Exercise:***
  - ***Take all .dat files as input and output a list of the unique particle names.***
  - ***Hint: You'll need to pipe through*** `sort` ***and then*** `uniq`***...***
- My solution:



```
#!/bin/bash
#After the shbang, lines starting with a hash are comments
files=`ls *.dat`
particles=`cut -d “ “ -f 2 $files | sort | uniq`
echo $particles
```

Input to Scripts

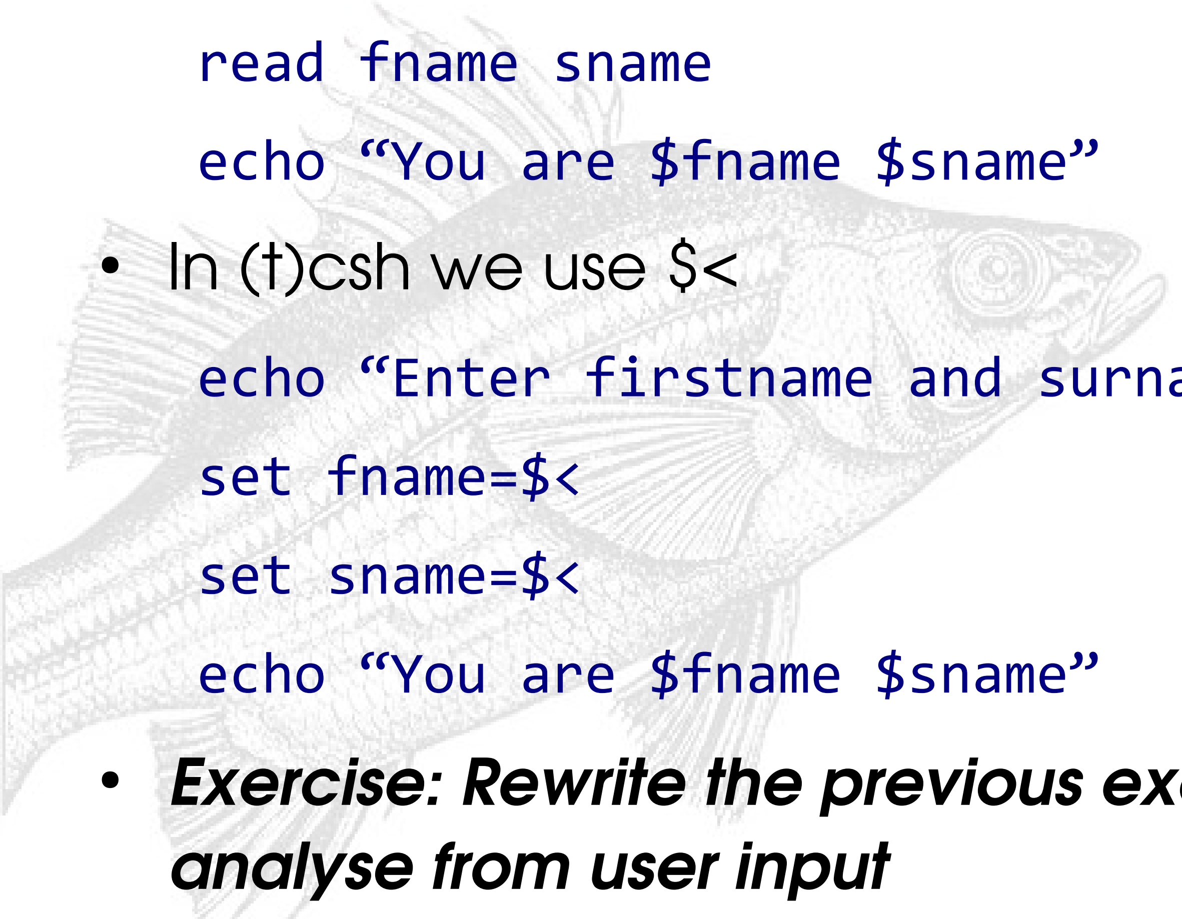
- We have two ways to supply input directly to the script
- Firstly, the script can prompt for input and parse this using the read builtin (bash only):

```
echo “Enter firstname and surname”
read fname sname
echo “You are $fname $sname”
```

- In (t)csh we use $<

```
echo “Enter firstname and surname:”
set fname=$<
set sname=$<
echo “You are $fname $sname”
```

- ***Exercise: Rewrite the previous exercise script to take the file to analyse from user input***

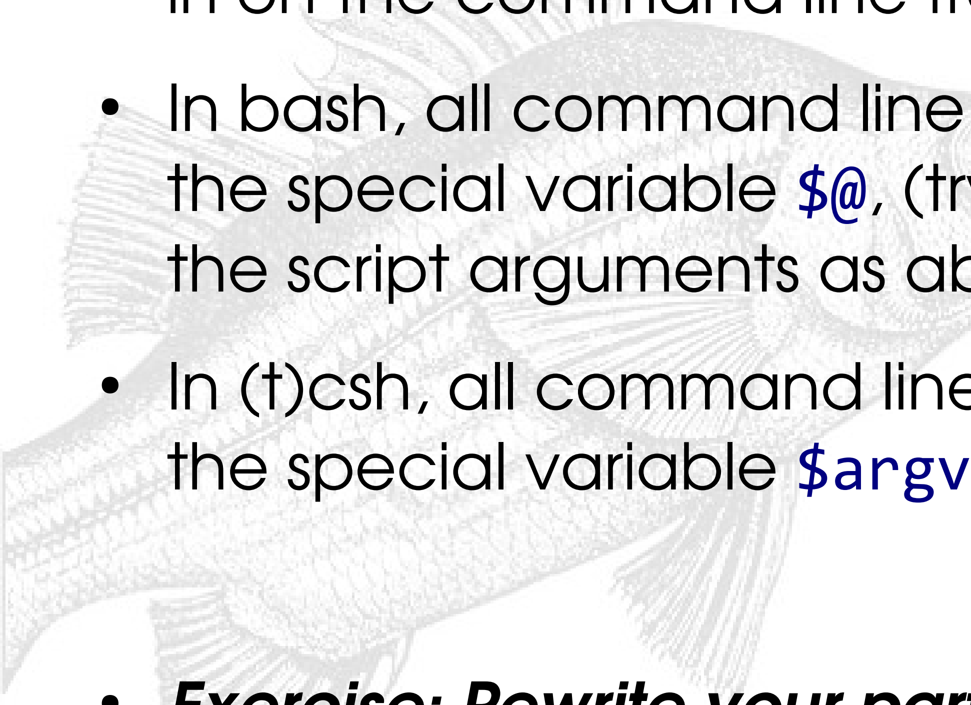


Input to Scripts

- The second way we can supply input to the script is through command line arguments, i.e.

```
[me@here ~]$ script arg1 arg2 arg3
```

- These are usable in the script as the special variables `$1,$2,$3,...,$N` where the integer represents they appeared in on the command line from left to right.
- In bash, all command line arguments are available through the special variable `$@`, (try echoing this in a script and passing the script arguments as above!).
- In (t)csh, all command line arguments are available through the special variable `$argv`.



- ***Exercise: Rewrite your particle name sorting script to take the file to be analysed as a command line argument.***

Conditionals: if

- if allows branching based on the result of a series of tests.
- In bash, the basic syntax is as follows

```
if [ FIRSTEXPRESSION ]
then
    echo “FIRSTEXPRESSION evaluated to true”
elif [ SECONDEXPRESSION ]
then
    echo “SECONDEXPRESSION is true”
else
    echo “Neither test passed”
fi
```

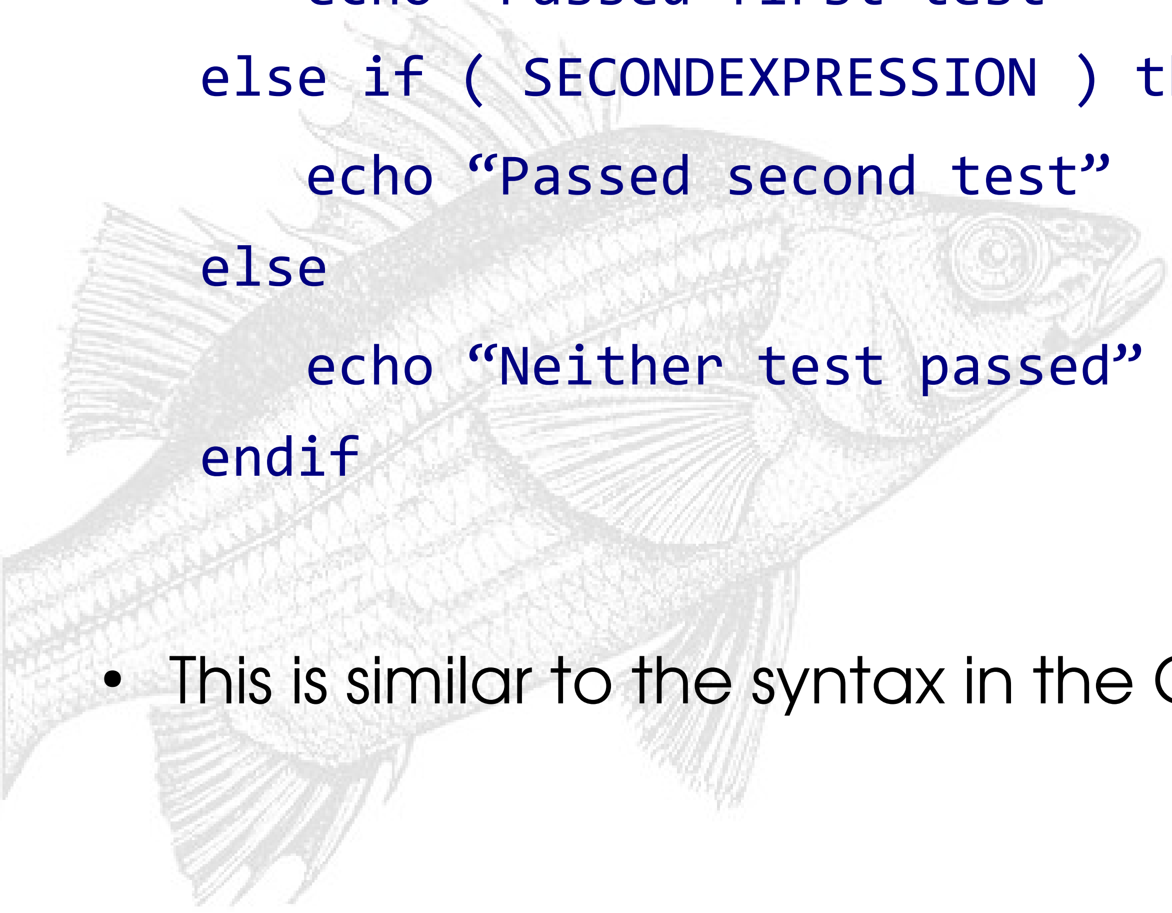
*Aside:
We can add as many elif statements as required.*

- The expressions must evaluate to TRUE or FALSE.
- Note that for Unices, TRUE is 0 and FALSE is 1.

Conditionals: if

- if in (t)csh has a slightly different syntax.

```
if ( FIRSTEXPRESSION ) then
    echo “Passed first test”
else if ( SECONDEXPRESSION ) then
    echo “Passed second test”
else
    echo “Neither test passed”
endif
```



- This is similar to the syntax in the C/C++ language.

Expressions in Bash

- The expression that `if` evaluates

```
if [ FIRSTEXPRESSION ]
```

takes unary (one arg) and binary (two args) forms.

- File tests:

  `[ -e FILE ]` TRUE if FILE exists

  `[ FILE1 -nt FILE2 ]` TRUE if FILE1 newer (by time) than FILE2

- String comparison

  `[ STRING1 == STRING2 ]` TRUE if strings are equal

  `[ STRING1 < STRING2 ]` TRUE if STRING1 sorts before STRING2

- Arithmetic comparison

  `[ NUM1 OP NUM2 ]` OP is one of `-eq`, `-ne`, `-lt`, `-le`, `-gt`, `-ge`.

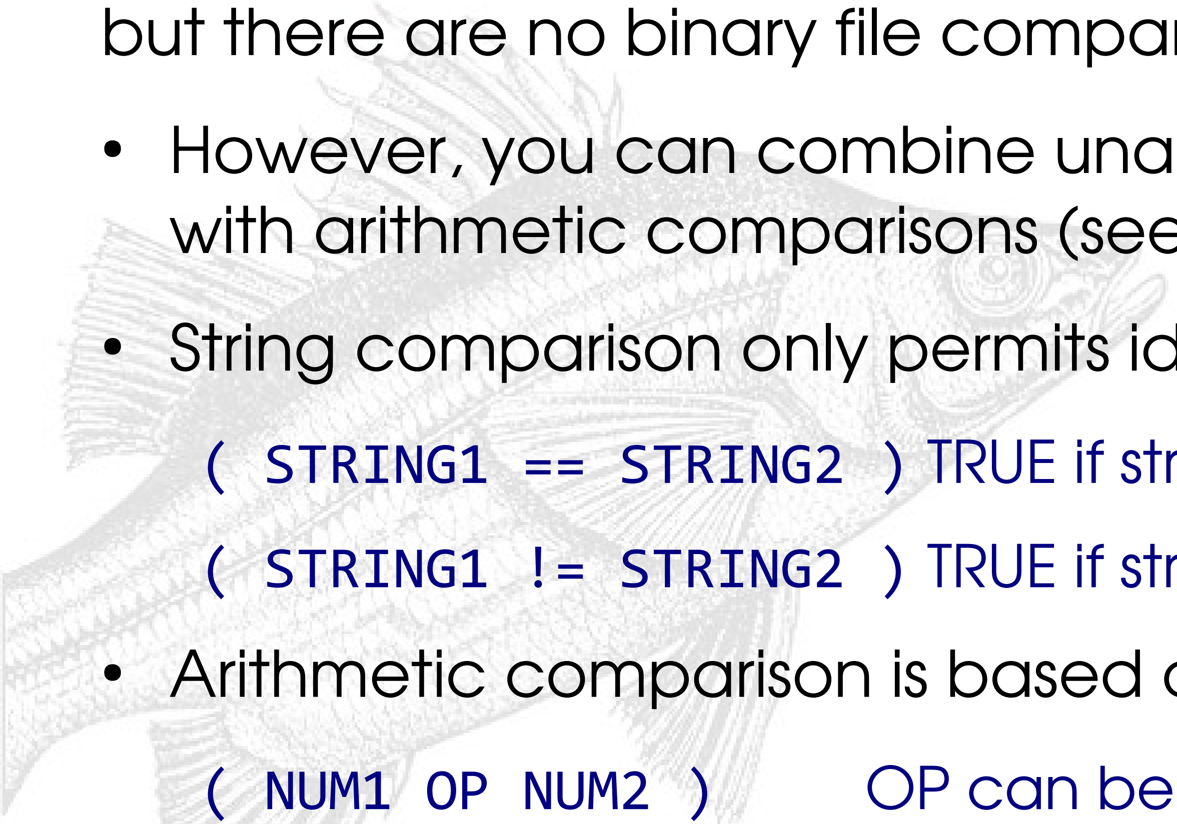
- See the bash manual pages for more information.

Expressions in (T)Csh

- Expressions in (t)csh are fairly similar to bash
- File tests are possible, e.g.

  `( -e FILE )` TRUE if FILE exists

but there are no binary file comparison operators.



- However, you can combine unary operators (e.g. -A, -Z) with arithmetic comparisons (see below).
- String comparison only permits identity tests

  `( STRING1 == STRING2 )` TRUE if strings are equal

  `( STRING1 != STRING2 )` TRUE if strings are not equal

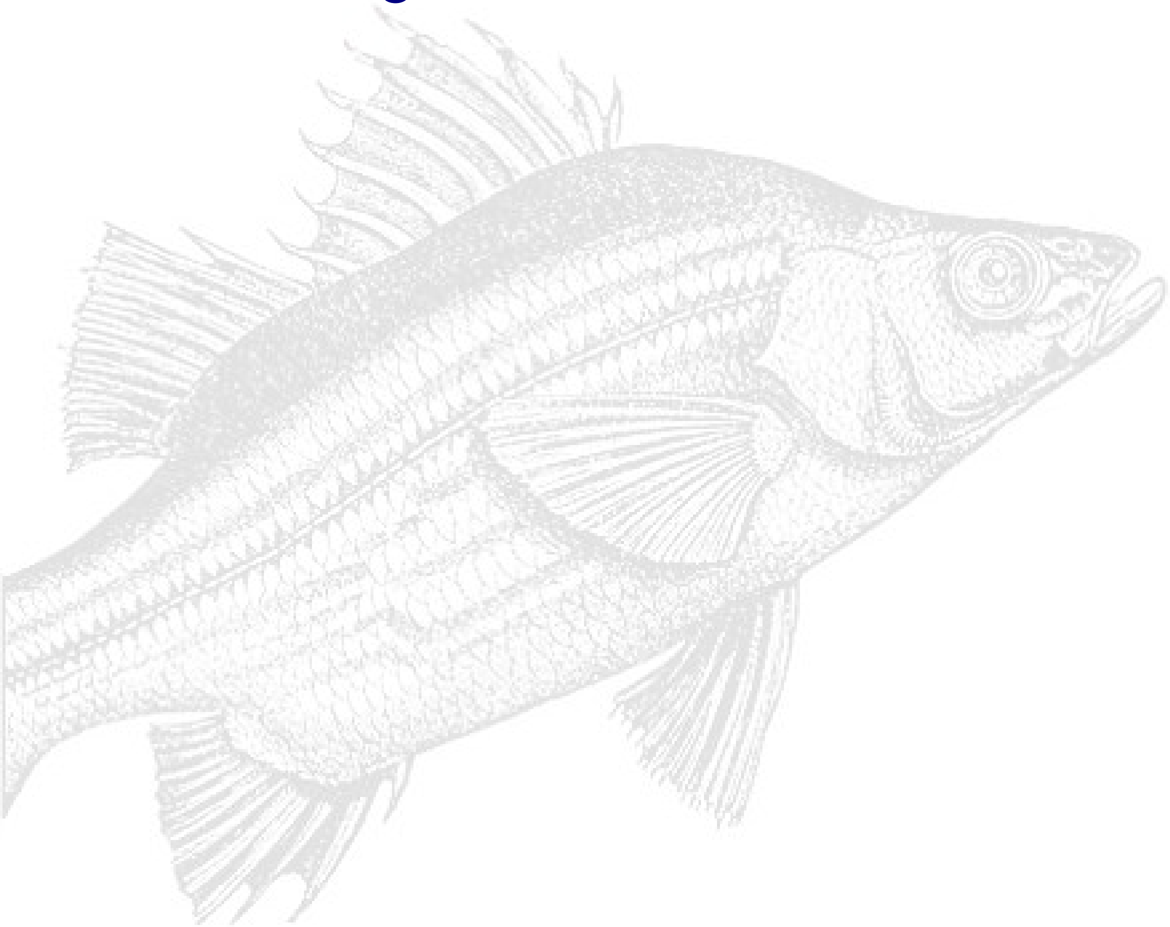
- Arithmetic comparison is based on C-style operators

  `( NUM1 OP NUM2 )` OP can be `== != <= >= < >`

- See the (t)csh manual pages for more information.

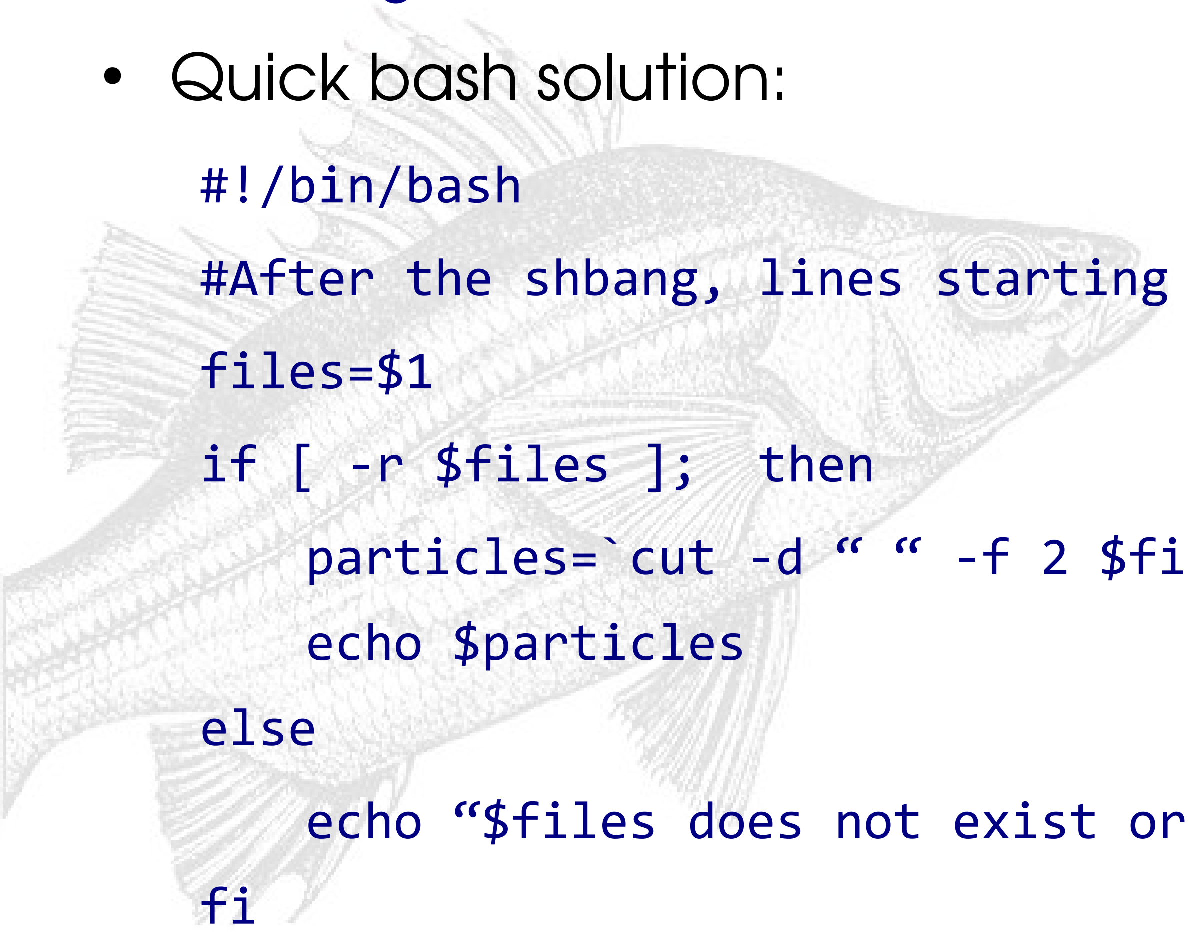
An Exercise using if

- Using the script you've already written to output a list of unique particle names in our data files, add a check on the existence of the file(s).
  - Design issue: is it better to check for existence or readability?



An Exercise using if

- Using the script you've already written to output a list of unique particle names in our data files, add a check on the existence of the file(s).
  - Design issue: is it better to check for existence or readability?
- Quick bash solution:



```
#!/bin/bash
#After the shbang, lines starting with a hash are comments
files=$1
if [ -r $files ];  then
    particles=`cut -d “ “ -f 2 $files | sort | uniq`
    echo $particles
else
    echo “$files does not exist or is not readable”
fi
```

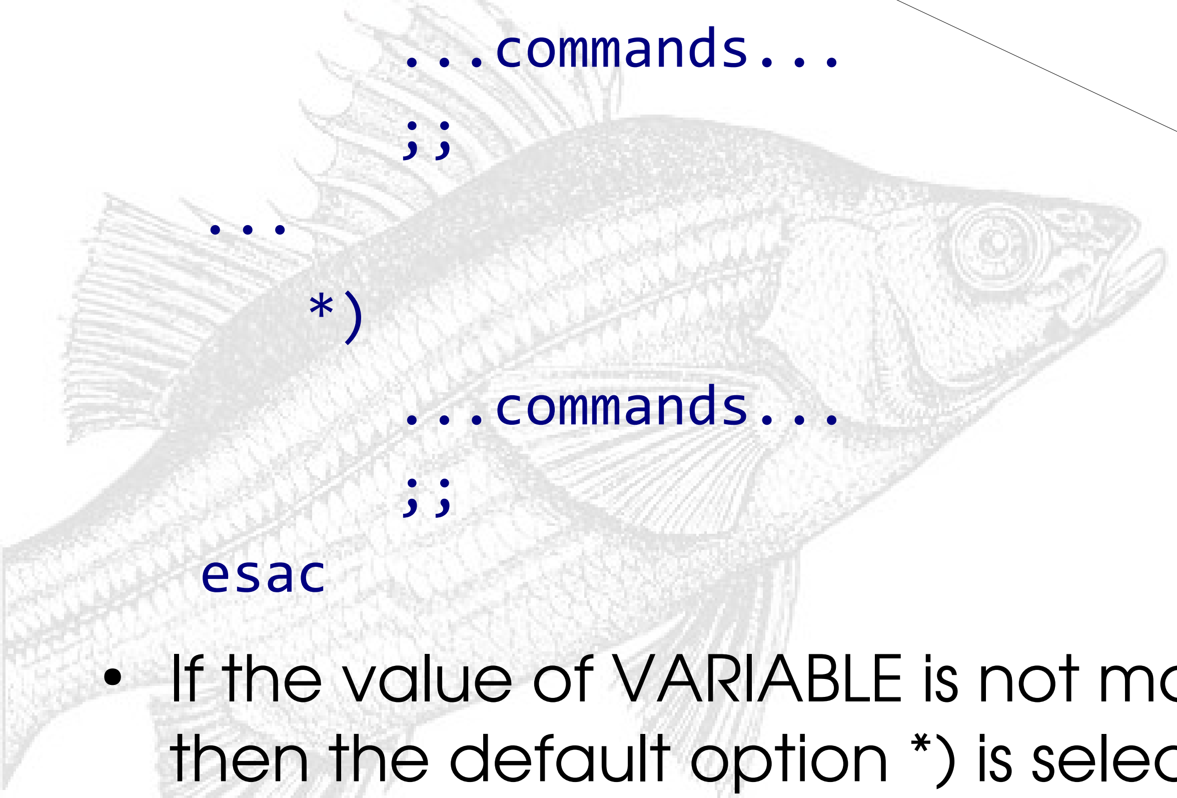
Conditionals: case

- case conditional allows complex conditional choices to be made. Basic structure in bash is:

```
case VARIABLE in
   OPTION1)
      ...commands...
      ;;
...
   *)
      ...commands...
      ;;
esac
```

Aside:
Options may be simple strings/ints or more complex regular expressions, e.g.
[aA]rg)
would match 'arg' AND 'Arg'

- If the value of VARIABLE is not matched by any OPTION, then the default option *) is selected.
- Typical 'use case' is processing command line options.

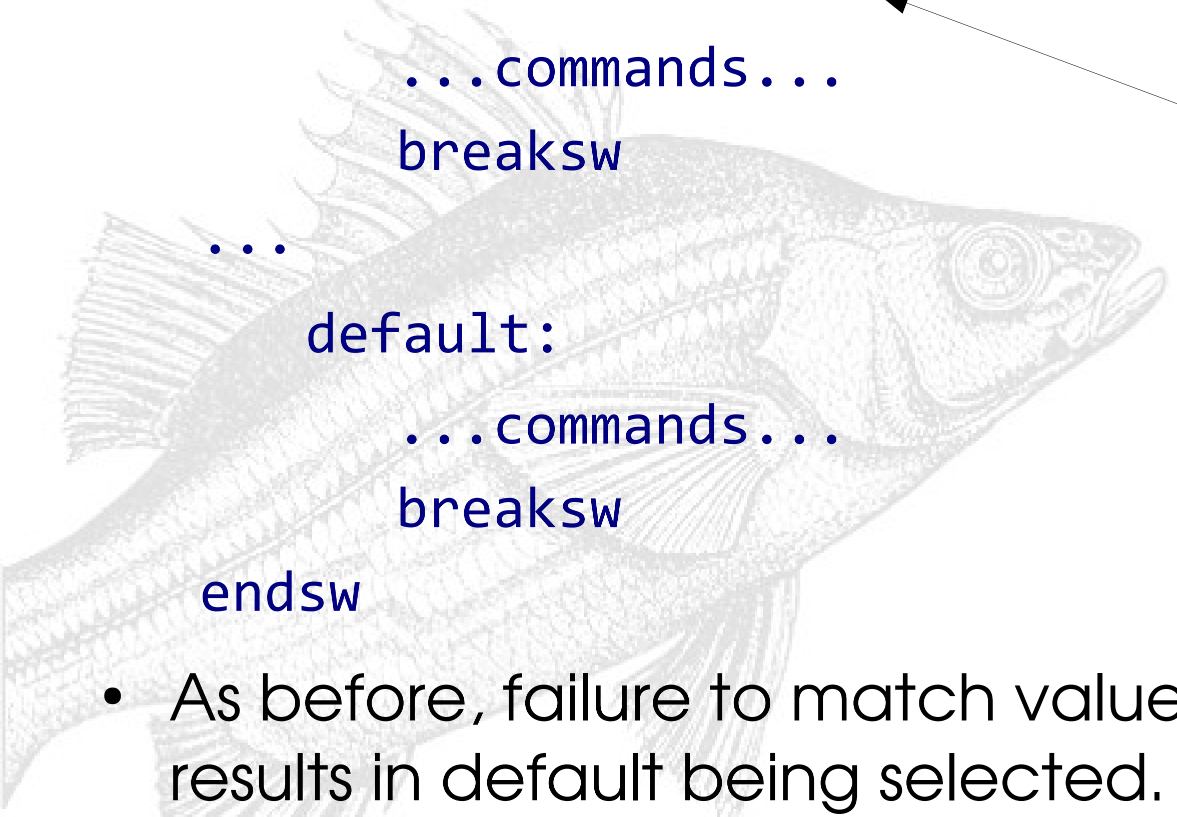


Conditionals: switch

- In (t)csh, `case` is replaced with the very C-like `switch` statement that is written as

```
 switch (VARIABLE)
     case OPTION1:
         ...commands...
         breaksw
 ...
     default:
         ...commands...
         breaksw
 endsw
```

Aside:
Options may be simple strings/ints or more complex regular expressions, e.g.
*   [aA]rg:*
would match 'arg' AND 'Arg'



- As before, failure to match value of VARIABLE to any OPTION results in default being selected.
- Typical 'use case' is again processing command line options.

Loops: for

- Loops enable a sequence of commands to be repeated a defined number of times, potentially with different input.
- In bash, a loop can be performed using for:

```
for VARIABLE in LIST
do
    ...commands...
done
```

- i.e. commands are repeated for every value in the LIST, e.g.

```
for num in `seq 1 10`
do
    let sqr=$num*$num
    echo $sqr
done
```

Loops: foreach

- (t)csh is quite un C-like here, as instead of `for` it uses `foreach` written as

```
foreach VARIABLE (LIST)
    ...commands...
end
```

- Just like bash, commands are repeated for every value in the LIST, e.g.

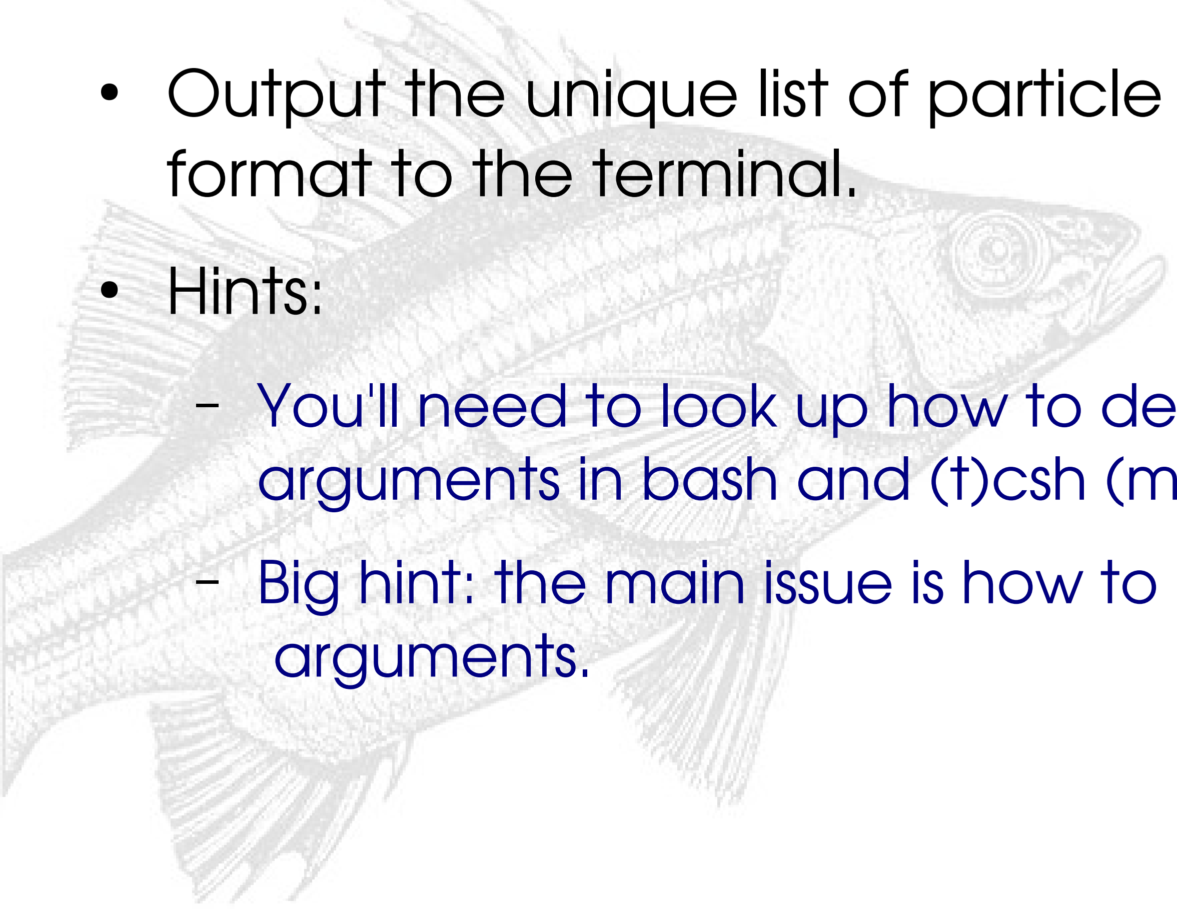
```
foreach num (`seq 1 10`)
    @ sqr = $num * $num
    echo $sqr
end
```

Exercise using for/foreach

- Take your script for extracting particle names from files and modify it to accept n filenames as command line inputs, i.e.

```
[me@here ~]$ ./myscript file1 file2 file3 (and so on)
```

- Make sure to check that each file is readable.
- Output the unique list of particle files for each file in a nice format to the terminal.
- Hints:
  - You'll need to look up how to deal with command line arguments in bash and (t)csh (man and Google).
  - Big hint: the main issue is how to get a list of the command line arguments.



Loops: while

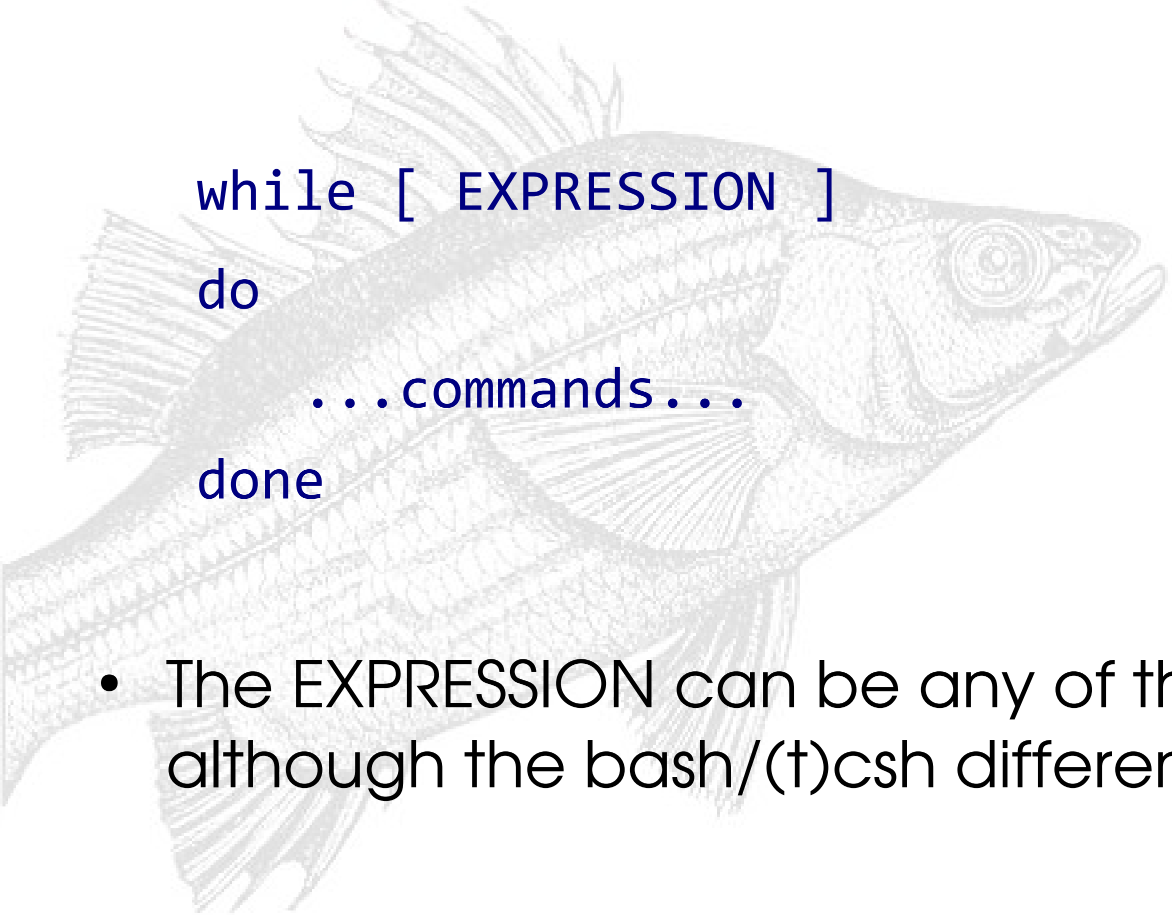
- for loops only repeat a fixed number of times, whereas while enables repeated execution until a conditional expression evaluates to FALSE.

- Basic syntax in bash is

```
while [ EXPRESSION ]
do
    ...commands...
done
```

- Whilst in (t)csh we write

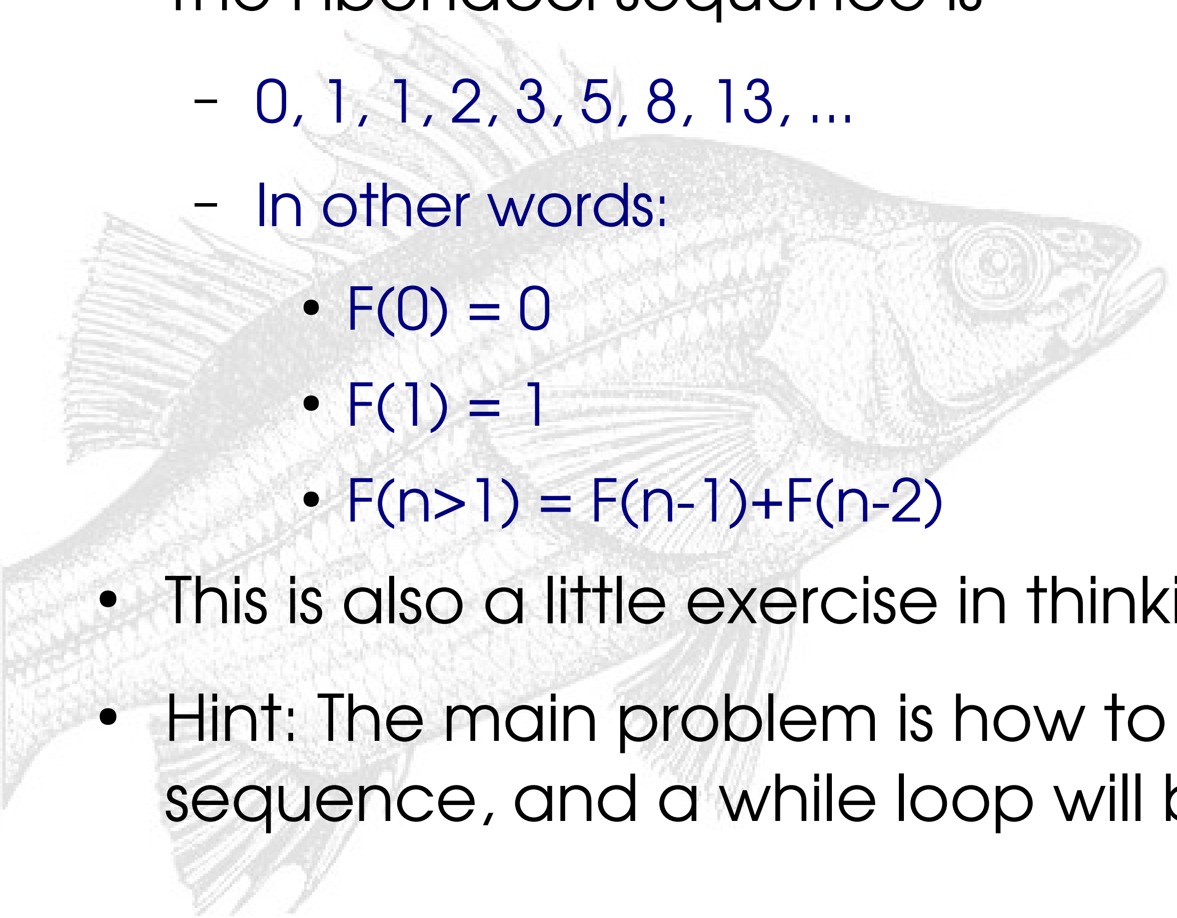
```
while ( EXPRESSION )
    ...commands...
end
```



- The EXPRESSION can be any of those we saw earlier, although the bash/(t)csh differences must be considered.

Fibonacci Sequence

- Write a script in bash or (t)csh to print out the first n numbers in the Fibonacci sequence.
  - The user should be able to specify n.
- The Fibonacci sequence is
  - 0, 1, 1, 2, 3, 5, 8, 13, ...
  - In other words:
    - F(0) = 0
    - F(1) = 1
    - F(n>1) = F(n-1)+F(n-2)
- This is also a little exercise in thinking about programming!
- Hint: The main problem is how to treat the start of the sequence, and a while loop will be useful!

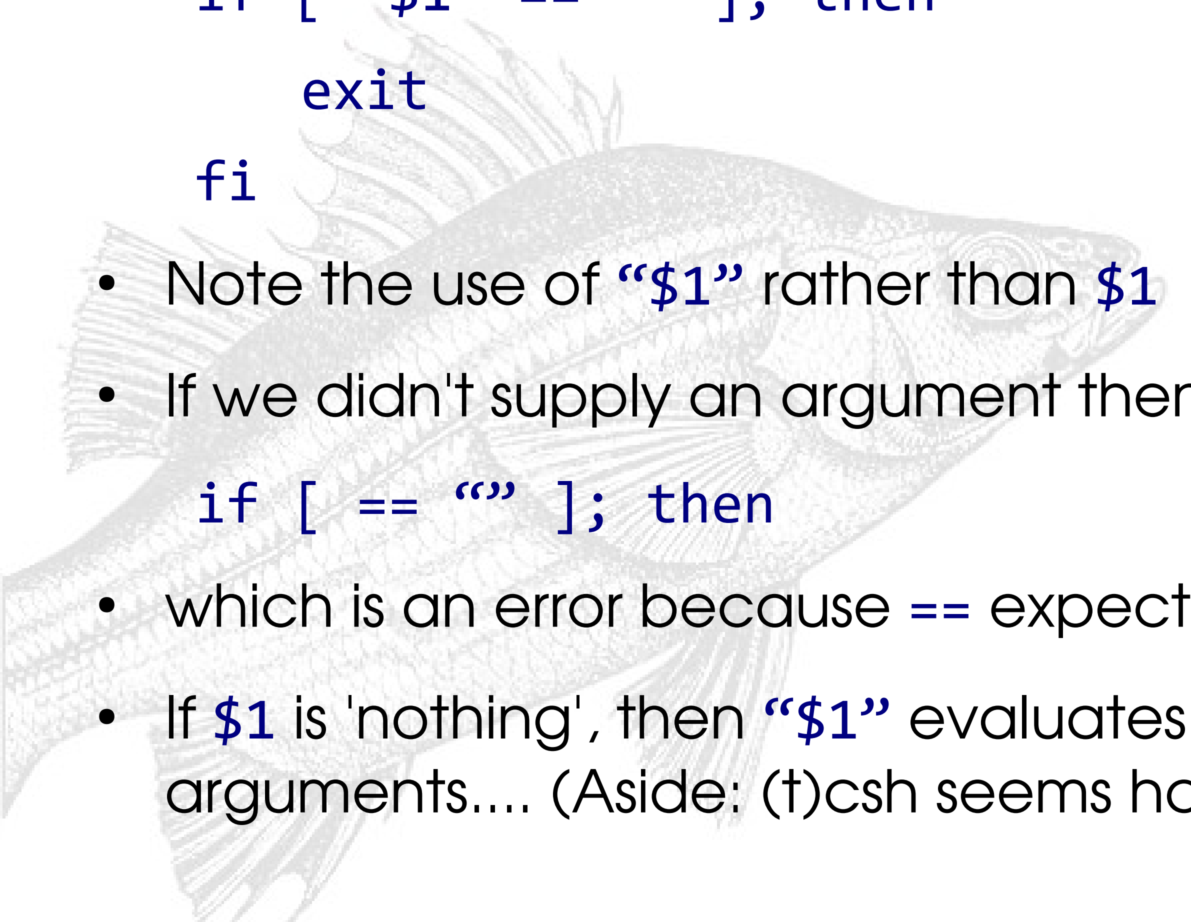


Fibonacci Solution

- Solution will be available on request, just send me an email
- Some further notes on this exercise:
- Verifying that we have the correct input is always good practice

```
if [ “$1” == “” ]; then
    exit
fi
```

Aside:
The expression
```
   x$var == x””
```
is another common check against 'nothing'



- Note the use of “$1” rather than $1
- If we didn't supply an argument then $1 is 'nothing' so bash sees

```
if [ == “” ]; then
```

- which is an error because == expects two arguments.
- If $1 is 'nothing', then “$1” evaluates to “”, so we do get two arguments.... (Aside: (t)csh seems happier with 'nothing')

Where to go next

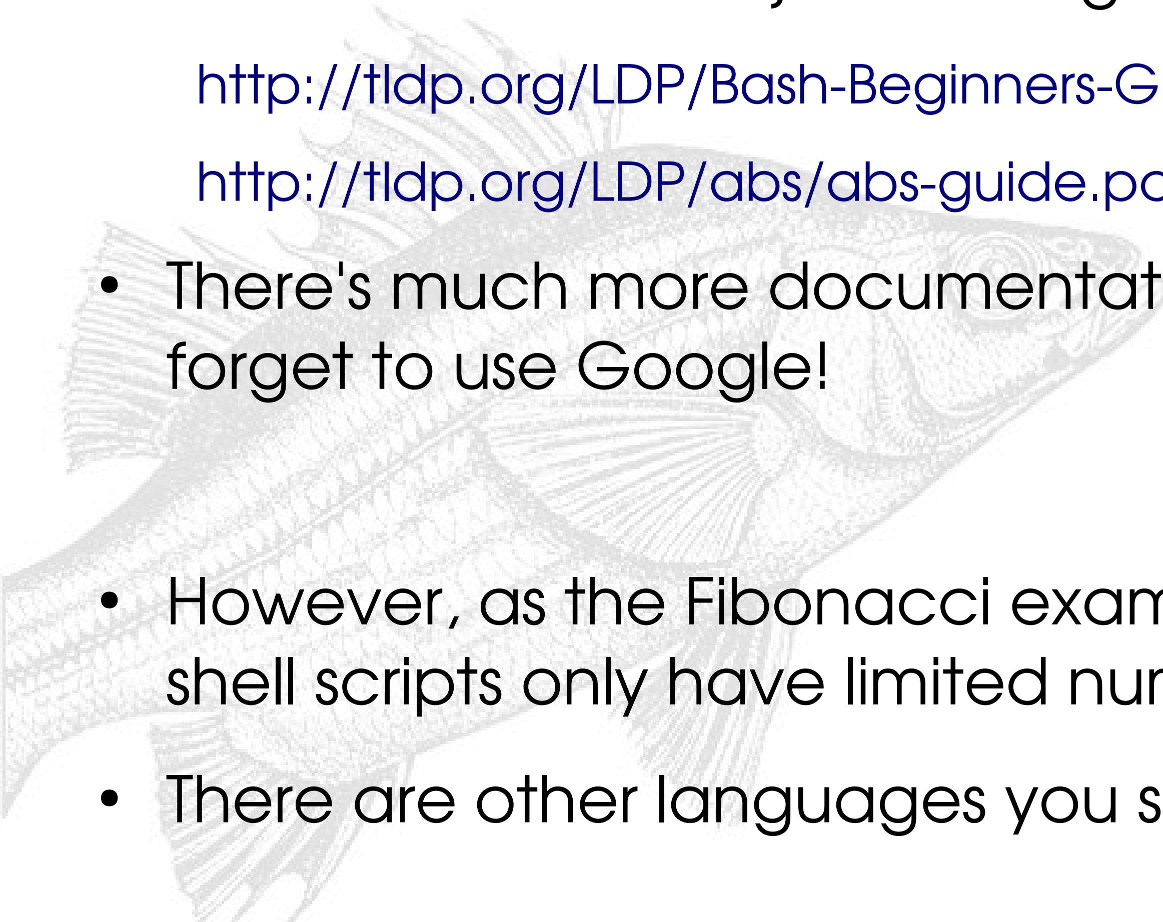
- Even in a booklet we've only had time to look at the basic features of shell scripting.
- You can find lots more helpful information in the Linux Documentation Project bash guides:

  http://tldp.org/LDP/Bash-Beginners-Guide/Bash-Beginners-Guide.pdf

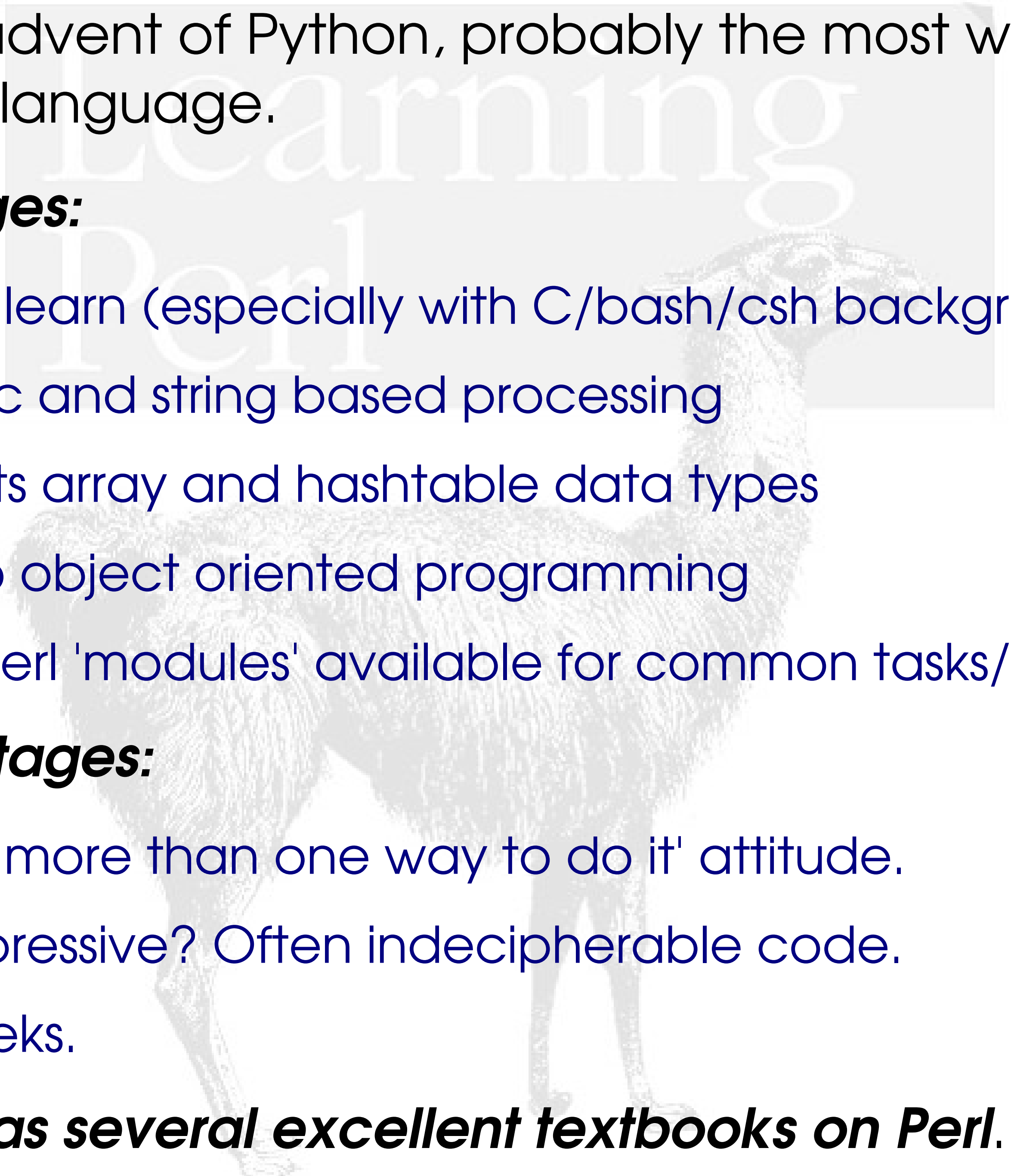
  http://tldp.org/LDP/abs/abs-guide.pdf

- There's much more documentation out there, so don't forget to use Google!

- However, as the Fibonacci example may have illustrated, shell scripts only have limited numerical power.
- There are other languages you should investigate.

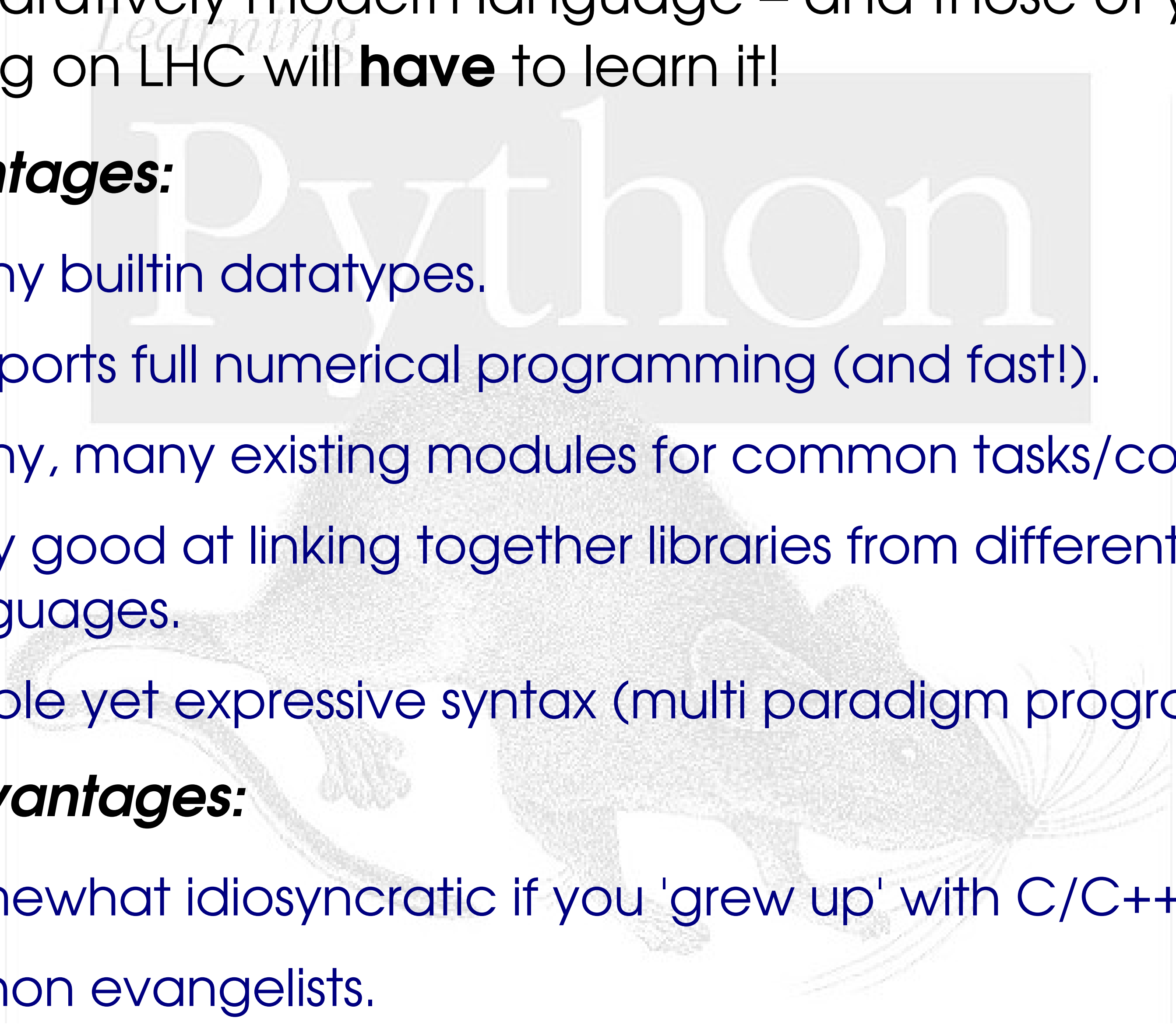


Perl



- Until the advent of Python, probably the most well known 'scripting' language.
- ***Advantages:***
  - Easy to learn (especially with C/bash/csh background)
  - Numeric and string based processing
  - Supports array and hashtable data types
  - Can do object oriented programming
  - Many Perl 'modules' available for common tasks/connectivity.
- ***Disadvantages:***
  - 'There's more than one way to do it' attitude.
  - Too expressive? Often indecipherable code.
  - Perl geeks.
- ***O'Reilly has several excellent textbooks on Perl***.

Python



- Comparatively modern language – and those of you working on LHC will **have** to learn it!
- ***Advantages:***
  - Many builtin datatypes.
  - Supports full numerical programming (and fast!).
  - Many, many existing modules for common tasks/connectivity.
  - Very good at linking together libraries from different languages.
  - Simple yet expressive syntax (multi paradigm programming!)
- ***Disadvantages:***
  - Somewhat idiosyncratic if you 'grew up' with C/C++/Java
  - Python evangelists.
- ***Once again, O'Reilly have many excellent textbooks.***

Further Resources

- As with almost anything, O'Reilly provide some excellent texts on Perl and Python.

- There are also the websites

  http://www.perl.com, http://www.perl.org

  http://www.python.org