Testing conditions - Exit status of commands…

The success of a shell command can be tested directly with
- Either the Shell keywords ***while, until, if***
- Or the control operators **&&** and **||**

❑Success denoted by either true or 0; (Contrary to most Boolean norms!)

❑Failure denoted by false or any integer in range 1-255
(specific fail condition shown by number, & handled by returned routine)
- 1 being the usual response for failure
- Careful when testing for failure, that not just testing for 1!
  - i.e. use [$? –ne 0] rather than [$? –eq 1]

❑Command may also be a logical expression which is checked for truth by
- ***test*** command or equivalent [command]
  with a space between the parentheses and enclosed command.
  Probably best avoiding […] as it's cryptic, but is compact & handy
- Or one of two nonstandard shell reserved words
  - [[…]] – same as test + regular expressions
  - ((…)) – for arithmetic tests

Testing conditions - Exit status of commands…

- Logical Expressions may be combined with **–a** for **AND** or **–o** for **OR**
- Commands may be joined with **&&** for **AND** or **||** for **OR** ,

  BUT the meaning relates to their execution rather than logic
  - **&&** – means both are executed, only if both are true (successful)
    - In effect this means that commands will stop executing, after the first in the sequence fails.
  - **||** – means only one true (successful) command need be executed
    - In effect, this means that commands will stop executing, after the first in the sequence succeeds.
- Combinations of **&&** & **||** can effect compact, if not rather cryptic and confusing error prone, implementations of ***if… then… else …fi*** etc.
- Whose component logical tests can be combined with –a and –o for complete flexibility…and the risk of complex cryptic confusion.
- Usual language tradeoff : expressiveness errors; concise confuse.

Decision Making

The Bourne shell provides a variety of decision making tools for your use

❑test… or […] … with a space between condition and brackets!

```
test "$a" = "$b"          ## true (returns 0) if $a = $b
[ "$a" = "$b" ]           ## true (returns 0) if $a = $b
test –f ~/rubb/testfile   ## true if a normal file
test –h ~/rubb/linkfile   ## true if a symbolic link
[ -x ~/rubb/hello_world ] ## true if executable
```

❑&& and || :- conjunctive & disjunctive ops

❑if-else, case :- if & case to avoid iffy logic

❑for, while, until :- loopy constructions

*test*ing for Decision Making

- *test* gives the shell the ability to make TRUE or FALSE decisions, returning a zero exit status if the condition evaluates to TRUE
- *test* is often used to test conditions in an *if*, *while*, or *until* command
- The *test* command has two possible formats
  - *test condition*
  - [*condition*] – should avoid, it's cryptic, & error prone, but tradition, so need to know: & short => so handy!
  - where *condition* is an operator or a set of operators ANDed an/or ORed together
  - NOTE: when using the [*condition*] format, there must be a space separating the [and] symbols from the *condition*

File Operators

Operator	*Returns **TRUE** (zero exit status) if*
-d *file*	*file* is a directory
-f *file*	*file* is an ordinary file
-r *file*	*file* is readable by the process
-s *file*	*file* has non-zero length
-w *file*	*file* is writable by the process
-x *file*	*file* is executable

Testing strings..

- NB Always quote string variables
  when testing to avoid problems with null and void values
- The null string is ASCII 0 or “”
- “$string” (double quouted) works even if $string is null,
  - The system realises it is dealing with a null string
- But
  - $string will usually not work correctly if $string is null
    - The system thinks it is dealing with a variable whose name is null (or more precisely a null string name!)
    - Although the reported errors will differ across shells, there may still be errors which are best avoided, achieved simply by quoting the string variable
      “$string”

String Operators –z –n (zero (=null), nonzero)

Operator	*Returns **TRUE** (zero exit status) if*
string	*string* is not *null*
-n *string*	*string* is not *null* (and *string* must be seen by *test*)
-z *string*	string is *null* (and *string* must be seen by *test*)
$string_1$ = $string_2$	$string_1$ is identical to $string_2$
$string_1$!= $string_2$	$string_1$ is ***not*** identical to $string_2$

-z (zero (=null)) and –n (nonzero)

- *** can be used to determine if a variable is undefined ***
- remember $? Is the exit status of the last command;
- here are a few examples run on cs1:-

(note that " test string ", with no flag, defaults to –n; first case)

```
cs1> test "" ; echo $?
1
cs1> test -n "" ; echo $?
1
cs1> [ -n "" ] ; echo $?
1

cs1> [ -z "" ] ; echo $?
0
cs1> test -z "" ; echo $?
0
```


String-ing along

Not the most elegant
But quick, short & handy.

```
#!/bin/bash
null=; notnull=abc
echo "$null"null
echo "$notnull"notnull
test -n "$null" ; echo "-n null?" $?
test -n "$notnull" ; echo "-n notnull?" $?
test -z "$null" ; echo "-z null" $?
test -z "$notnull" ; echo "-z notnull" $?
exit 0
```

```
cs1> bash stringtest
null
abcnotnull
-n null? 1
-n notnull? 0
-z null 0
-z notnull 1
cs1>
```


Lexical comparison… alphabetic sort

In bash, the < and > symbols used in comparison,
Must be escaped with \
to prevent their being interpreted as redirection operators.

```
str1=abc
str2=def
test "$str1" \< "$str2"
echo $?
0
test "$str1" \> "$str2"
echo $?
1
```


Integer Comparison Operators

Operator	*Returns **TRUE** (zero exit status) if*
int_1 -eq int_2	int_1 is equal to int_2
int_1 -ge int_2	int_1 is greater than or equal to int_2
int_1 -gt int_2	int_1 is greater than int_2
int_1 -le int_2	int_1 is less than or equal to int_2
int_1 -lt int_2	int_1 is less than int_2
int_1 -ne int_2	int_1 is not equal to int_2

Boolean Truth Table

Var A	Var B	AND	OR
FALSE	FALSE	FALSE	FALSE
TRUE	FALSE	FALSE	TRUE
FALSE	TRUE	FALSE	TRUE
TRUE	TRUE	TRUE	TRUE

Boolean Operators

Operator	*Returns **TRUE** (zero exit status) if*
! *expr*	*expr* is FALSE; otherwise return TRUE
$expr_1$ -a $expr_2$	$expr_1$ is TRUE and $expr_2$ is TRUE
$expr_1$ -o $expr_2$	$expr_1$ is TRUE or $expr_2$ is TRUE

- Note: the -a operator has a higher precedence than - o. This means that:

  $expr_1$ -o $expr_2$ -a $expr_3$ is interpreted as:

  $expr_1$ -o ($expr_2$ -a $expr_3$) not ($expr_1$ -o $expr_2$) -a $expr_3$

testing times!? – using AND -a

```
times=~/rubbish/times

test ! -r "$times" ; echo $?
                        returns TRUE    if times is NOT readable by user

test ! -f "$times" ; echo $?
                        returns TRUE    if file does NOT exist
                                or is not an ordinary file
                                (if it's a directory, for example)

test -f "$times" -a -r "$times" ; echo $?

                        returns TRUE    if times exists
                                AND is an ordinary file
                                AND is readable by user
```


AND –a OR –o Account for our times!?

```
count=5 ; test "$count" -ge 0 -a "$count" -lt 20 ; echo $?

returns TRUE
                if count contains an integer value
                    greater than 0 AND less than 20


test "$count -lt 10 -o -f "$times" ; echo $?

returns TRUE
                if count contains a value
                    less than 10 OR times is an ordinary file that exists
```


Shell Arithmetic – needs expr

- The Bourne shell
  - Treats all variables as strings
  - So unless explicity overruled it has no idea how to do arithmetic
- For example

```
number=2
number=$number + 4
echo $number
2 + 4
```

- It just concatenates ' + 4' to the value for $number
- At first sight, the shell appears useless for sums!

Expr – integer arithmetic only!

- Fortunately, there is a Unix command that will allow us to perform arithmetic within a script
- The *expr* command "evaluates" it's arguments and writes it's output on STDOUT
- Example:

```
expr 1 + 2
3
expr 6 / 2 + 5
8
```

- Note, since it is evaluating arguments, they **must** be separated by spaces
- Also, expr only works with integer arithmetic expressions => so values are truncated (decimal chopped & dropped)

Double bracketed reserved test symbols **[[...]]**

- **[[...]]**
  - Like ***test***, it evaluates an expression,
  - Unlike ***test*** it
    - is not a built-in command, but part of shell grammar
    - Does not parse as a built in command between [[…]]
      - Parameters are expanded
      - But no word splitting or filename expansion
    - Supports the same operators as ***test***
      - + some enhancements & additions
    - Is non-standard, so use ***test*** in preference.
- NB **[...]** is a simpler form like test, without enhancements
- But **[[…]]** is shorter for showing & writing slides
  - So I like and use it for presentations – better use test

Non-standard enhancements to **[[...]]**

- If the argument to the right of = or != is unquoted, it's treated as a pattern and duplicates the functionality of a case option (Case statement covered within the next few (~10) slides)
- Can match extended regular expressions using =~ operator

```
$ string=whatever
$ [[ $string =~ h[aeiou] ]]     # 'w-ha-tever' matches 'ha'
$ echo $?
0
$ [[ $string =~ h[sdfghjkl] ]]  # 'w-ha-tever' does not match h[sdfghjkl]
$ echo $?
1
```

Remember $? Is the status of the last command not run in background?
So h[aeiou] succeeds with status 0, while h[sdfghjkl] fails with status 1.

Double bracketed reserved test symbol **((...))**

- ((arithmetic expression)) non-standard version returns
  - false if the arithmetic expression evaluates to zero ***
  - True otherwise.

```
test $(( $a – 2)) ne 0
```

If $a – 2 = 0, then the arithmetic expression is 0,
so (($a-2)) returns false***, which is 1 (or nonzero) in bash testing
So $(1) –ne 0 is true, which in bash testing is 0

- Again, using **if…fi** is cleaner & safer, unless absolutely sure of details.
- Avoid math confusion of splitting ((…)) into X = (…) so taking (X)
  - As possible if tired.
- The portable equivalent uses
  - ***test***
  - And the POSIX syntax for shell arithmetic

If … then … fi

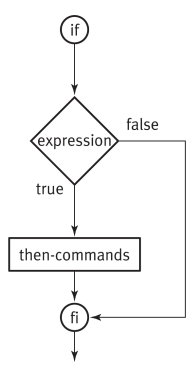


Figure 15.1 Semantics of the `if-then-fi` statement

If … then … fi

- if-then allows you to execute a series of commands if some test is TRUE, if not you can execute a different set of commands

```
if command_t
then
        command
        command
        ....
fi
```

- if *command*$_t$ returns a TRUE (zero status) then the following commands are executed

If … then … else … fi

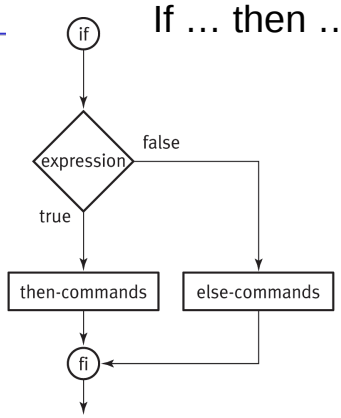


Figure 15.2 Semantics of the `if-then-else-fi` statement

Random samples…

```
# check we are on d' right horse
if [[ `hostname` != "cs1" ]]
  then
    echo -e "This script only runs on cs1"
    echo -e "##############################"
    echo -e "you need to be on d' right horse Boy!"
    echo -e "ABANDON now ..."
    exit 1
fi
```

```
if [[ ! $2 ]]
  then
    echo -e "Enter the Module code e.g. [cs1234]"
    echo -n "Module: "
    read MOD
  else
    MOD=$2
fi
```

If … then … else … fi

- Another optional form adds an *else* clause

if *command*$_t$
then
    command
    command
    ….
else
    command
    command
    ….
fi

If … then … elif … then … else … fi

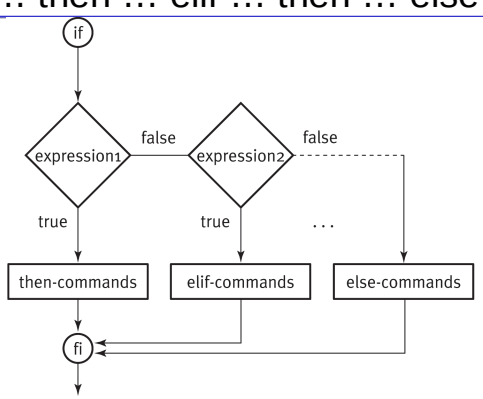


Figure 15.3 Semantics of the `if-then-elif-else-fi` statement

If … then … elif … then … else … fi

- One other form combines multiple ifs and elses

if *command*$_t$
then
    command
    ….
elif
then
    command
    …
else
    command
fi

Testing conditions - Exit status of commands…

- Logical Expressions may be combined with **–a** for **AND** or **–o** for **OR**
- Commands may be joined with **&&** for **AND** or **||** for **OR** ,

  BUT the meaning of && and || relates to their execution rather than logic

  - **&&** – means both are executed, only if both are true (successful)
    - In effect this means that commands will stop executing, at failure of the first command in the sequence.
  - **||** – means only one true (successful) command need be executed
    - In effect, this means that commands will stop executing, at success of the first command in the sequence.
- Combinations of **&&** & **||** can effect compact, if not rather cryptic and confusing error prone, implementations of ***if… then… else …fi*** etc.
- Whose component logical tests can be combined with –a and –o for complete flexibility…and the risk of complex cryptic confusion.
- Usual language tradeoff : expressiveness errors; concise confuse.

&& and || (can act as a (confusing) if ..)

- && and II allow you to conditionally execute commands
- *command*$_1$ && *command*$_2$
  - *command*$_2$ executes only if *command*$_1$ returns a zero status (executed successfully)
  - Logically like … **if** command$_1$ **then** command$_2$ **fi**
  - With first cmd done: **if** command$_1$ **then** { command$_1$; command$_2$ } **fi**
- *command*$_1$ || *command*$_2$
  - *command*$_2$ executes only if *command*$_1$ returns a non-zero status (did not execute successfully)
  - Logically like …
    - **If** *command*$_1$ **then** *command*$_1$ **else** *command*$_2$ **fi**
    - or **If NOT**(*command*$_1$) **then** *command*$_2$ **fi** ..
    - or **if** !(*command*$_1$) **then** *command*$_2$ **fi**
- Examples

  who | grep "your_id" > /dev/null && echo "You are logged on"
  who | grep "Noah" > /dev/null || echo "Noah is not logged on"

Possible examples

- && AND

test $debug -eq 1 && echo some_debug_output

If debug level is equal to 1,
then print some_debug_output

- || OR

test $debug -eq 1 || echo some_debug_output

If debug level is NOT equal to 1,
then print some_debug_output

And even more obtuse combinations…

times=~/rubbish/times
test -f "$times" –a $test –eq 0

This will test if $times is a valid file … AND if so…on to the next command
(***$test –eq 0*** is clearly an unecessary superfluous test for showing **$**test)

(Need to be careful about impossible contradictions in logic…
e.g. **test -f "$times" –a $test –eq 1**
if test(valid file) AND if (that test failed) …impossible contradiction !
(although an improperly recoverd system failure between tests …
… could conceivably validate the impossible!?)

Even if the test above were done correctly, it is still artificially repetitive
e.g. **test -f "$times" –a $test –eq 0**

test -x bin/file -o $test -gt 1

This will test

IF bin/file is executable
OR IF an error occurs…e.g. file missing

test normally used with if… or && or || operators.

To read and check input

```
read name
if [ -z "$name" ] then                  # if $name is null  - non entered.
echo "No name entered" >&2              # redirect to standard error file &2
  exit 1                                # Set a failed return code & exit
fi                                      # ( with exit 0 set elsewhere …
                                        #  … on normal termination)
```

Of course a better solution, would be to request a re-entry until data OK.

NB normal practice is to precede the while test with an attempted initiation, otherwise the name in the test will be undefined, and the loop may be skipped, but it is acceptable here, since the test is for an undefined value, which is input in the body of the loop, and repeated until non-zero.

```
while [ -z "$name" ]
do
 echo Please enter a name…
 read name
done
```


…and more obtuse (but compact) still…

- For example, to check for a directory and cd into it if it exists, use this:
  test -d "$directory" && cd "$directory"
- To change directory and exit with an error if cd fails, use this:
  cd "$HOME/bin" || exit 1
- The next command tries to create a directory and cd to it.
  If either mkdir or cd fails, it exits with an error:
  mkdir "$HOME/bin" && cd "$HOME/bin" || exit 1
- Conditional operators are often used with if.
  Here, the echo command is executed if both tests are successful:

```
if [ -d "$dir" ] && cd "$dir"  # read as : if both succeed
then
   echo "$PWD"
fi
```

  - Cryptic form: **[-d "$dir"] && cd "$dir" && echo "$PWD"**

Compounded extensions of && and ||!

Consider each of the following:-

1. cmd_1 && cmd_2 && cmd_3 ;
   - cmd_3 will only be run if the previous two are true
2. cmd_1 || cmd_2 || cmd_3 ;
   - cmd_3 will only be run if the previous two are false
3. cmd_1 && cmd_2 || cmd_3 ;
   - cmd_3 will only be run
     if either of the previous two are false
4. cmd_1 || cmd_2 && cmd_3 ;
   - cmd_3 will only be run
     if cmd_1 is false and cmd_2 is true.

For clarity and subsequent checking & debugging, if is easier..
But c_1 && c_2 && … c_n || $c_{otherwise}$ is compact & can be clear.

Clearer reading & coding of && & || !

- Basically these can be powerful, compact and clear,
  when used to choose options in success or failure.
- Conjuctions : c_1 && c_2 && … && c_n
  - can be read as: do c_1 and if that's ok then do c_2 etc
  - Alternatives : c_1 || c_2 || … || c_n
  - can be read as: try c_1 and if that fails then try c_2 etc
  - Combinations : c_1 && … c_n || d_1 && … d_n || e_1 && … e_n
  - Can be read as
  - Try to (do c_1 and if that's ok then do c_2 etc … c_n)
    – And if that fails (i.e. any one in the 'c' list fails)
  - Try to (do d_1 and if that's ok then do d_2 etc … d_n)
    – And if that fails (i.e. any one in the 'd' list fails)
  - Try to (do e_1 and if that's ok then do e_2 etc … e_n)
    – And if that fails …

Gotchas! …

- Careful about lists, especially editing extensions…
  - If the original intention was : IF c_1 THEN c_2
    then a valid expression is ..c_1 && c_2
  - But if you wanted to include another command with c_2 ,
    then in haste, you might extend it like this
    c_1 && c_2 ; c_x

But, logically, a mess …
with c_x executed in any case!

```
if c1
then c2
fi
cx
```

… instead of intended…

```
if c1
then {c2 ; cx }
fi
```

… achievable simply by …

c_1 && { c_2 ; c_x }

Gotchas! … even more complex, confusing & common!

- Careful about lists, especially editing extensions…enclose within { ; }
  - If the original intention was : IF c_1 THEN c_2 ELSE c_3 FI

  then a valid expression is ..c_1 && c_2 || c_3
  - But if you wanted to include another command with c_2 , then in haste, you might extend it like this : - c_1 && c_2 ; c_x || c_3

… which , logically, is a mess …

if c_1
then c_2
fi
if ! c_x
then c_3
fi

… instead of intended…

if c_1
then {c_2 ; c_x }
else c_3
fi

… achievable simply by …

c_1 && { c_2 ; c_x } || c_3

NB:- Case statement options terminate with double semicolons ;;
so that individual statements in the list terminate with single

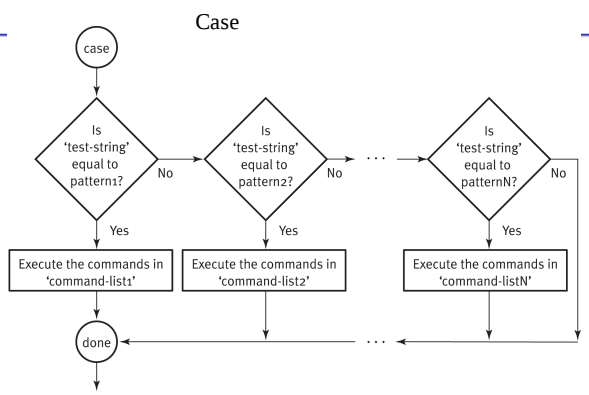


Figure 15.8 Semantics of the `case` statement

Case – note use of two separating semicolons

The case command permits comparing
a single value against one or more values
and execute one or more commands when a match is found

```
case value in
    pat1 )  command;
            .... ;
            command;;
    pat2 )  command;
            .... ;
            command;;
    * )     command;;
esac
```

```
case menuchoice in
    1 )  command;
         command;;
    2 )  command;
         command;;
    q)   break;;
    * )  command     # any other
         command;;   # invalid
esac                 # choice
```

- Handy for responding to a menu choice…
- Some have menu support, e.g. select but are non-standard/portable across shells, so other approaches covered first.
- NB lists of commands within an option are separated by ; & terminated ;;

Patterns in *case* Statements

- You can use the same special characters in case statement pattern specifiers as you do in shell file name substitutions
  - ? matches any single character
  - * matches zero or more occurrences of any character
    - This is different from normal regex where * means 0 or more occurrences of the previous character
  - […] matches any characters enclosed in the brackets

Special cases!?

- Case is often used to find if one string is in another,

Since grep spawns a new process, case is faster, and would be implemented as a shell function to avoid spawning one.

```
case $1 in
   *"$2"*) true ;;
   *) false ;;
esac
```

- Or to check if a number is valid…

```
case $1 in
   *[!0-9]*) false;;   ## a non-numeric character blows it!
   *) true ;;
esac
```


Hold on a Minute!

- Sometimes, you want your shell script to stop processing until either something happens or for some amount of time
- The *wait* and *sleep* commands provide these functions
- *wait* will cause the script to suspend execution until a specific process finishes
  - It's syntax is *wait n* where *n* is the PID of the process you want to wait for
  - Recall that $! can be used to get the PID of the last process sent to the background

- An example of *wait* might be

```
sort big_file > sorted_file &
pid=$!
command
command
….
wait $pid
plot sorted_file
```

- Here, we've started a large sort in the background. We want to do some other things while it is sorting but we can't plot the data until the sort has finished

Sweet Dreams

- Sleep suspends execution for a specified time.
- Syntax: *sleep n*
  - where *n* is the number of seconds to sleep
- It's used to execute a command after a certain amount of time as in:

```
(sleep 105; command)&
```

or to execute a command every so often, as in:

```
while true
do
    command
    sleep 37
done
```


I'm Outta' Here

- The *exit* command causes the current shell script to exit immediately and return the status specified by the exit command
- Syntax: *exit n*
  - where *n* is the desired exit status
  - if n is not given, exit returns the exit status of the last command that was executed by the script
- *exit* can be used to return diagnostic error codes when something goes wrong with your script
  - This is usually most interesting when your script calls other scripts, by using exit codes, you can incorporate error handling in you upper level scripts

Looping

- The shell has three built-in looping constructs
  - for
  - while
  - until
- These let you execute a set of commands either a specific number of times or until some condition is met

for

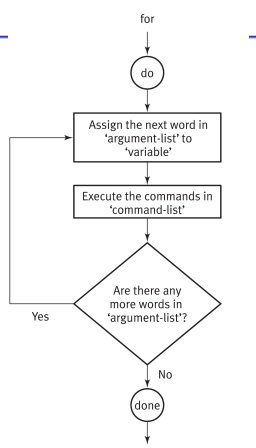


Figure 15.4 Semantics of the `for` statement

for

- General format of the *for* loop is:

```
for var in word1 word2 … wordn
do
    command
    command
    …
done
```

- When the loop is executed, first word1 is assigned to *var* and the body of the loop is executed
- Then *word2* is assigned to *var*, followed by *word3* until all the words have been processed

More *for*

```
for i in 1 2 3
do
    echo $i
done

for i in $*
do
    echo $i
done
```


while

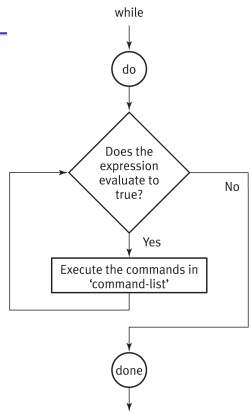


Figure 15.5 Semantics of the `while` statement

Special Form of *for*

- A special notation is recognized by the shell
- If you write

```
for var
do
    command
    command
    ….
Done
```

- The shell will automatically sequence through all the arguments typed on the command line

while

- while executes the commands between the do and done while the return status from $command_t$ is TRUE (zero)

```
while command_t
do
        command
        command
        …
done
```

- Infinite While Loop:

either	while true do… done
or	while : do … done

- *test* is often used in the while loop as $command_t$

A simple search script:

```
#!/bin/sh
# first
# This file looks through all the files in the current
# directory for the string SEARCHSTRING, and then prints the names of
# those files to the standard output.
for file in *
do
    if grep -q SEARCHSTRING $file
    then
        echo $file
    fi
done
exit 0
```

```
grep flags  - check manual for more chaos...
-l    list only filenames with matches, not every match,
      Suppress all other output
-q    quiet, don't print anything,
      On finding a match, exit immediately with status 0
      (in example code, echo then prints the filename
      on finding the first match.
```

But this could be easily done by…grep -l SEARCHSTRING * | more
Which is why we did all the string processing tools (tr, sed, grep, awk) first

the Wiles of whiles…

```
Cat wiles.scr
#!/bin/bash
while [ "$#" -ne 0 ]
do
    echo "$1"
    shift  # the arguments
Done


> wiles.scr 'a b' c
a b
C

> wiles.scr *
prints all the filenames in current directory
```

the Wiles of whiles…

```
while :                    # while true – infinite loop
do
  read x
  [ -z "$x" ] && break     # except for the break! Null input : done
done                       # when x is null and void!
```


Break or continue

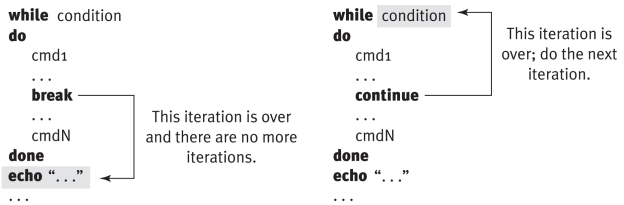


Figure 15.7 Semantics of the `break` and `continue` commands

until

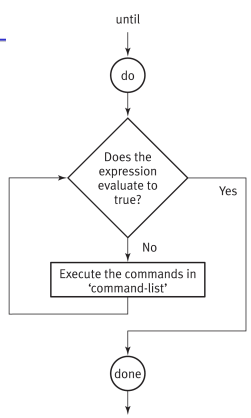


Figure 15.6 Semantics of the `until` statement

Break or continue

- Break-ing out of loops entirely
  - Sometimes you may want to exit immediately from a loop
  - *break* allows you to do this
- Continue – but pass on the rest of this pass
  - Skip the rest of this pass through the loop
  - But restart at the next pass
  - Sometimes you are in the middle of executing a pass through a loop and, rather than breaking out of the loop just want to skip the rest of the commands in this pass
  - *continue* will let you do that

until

- until executes the commands between the do and done until *command*$_t$ returns a TRUE status

  until *command*$_t$
  do
  command
  command
  ….
  done
- Again, the test *command* is often used for *command*$_t$ in until loops although other commands may certainly be used instead

Breaking out to any level…!

```
# for breaking out of a few levels...
for number in [0-9]
do
        echo -e " \n in loop number - $number"
        read -n1 -p "Press b to break out of this loop: " x
        [[ $x = "[bB]" ]] && echo -e " \nbreaking out early : \
                out of outer, before going into inner \n" ; break
        for letter in [a-z]
        do
                echo -e " \n in loop letter - $letter \n"
                read -n1 -p "Press 1 to break out of this inner letter loop \
                        Press 2 to break out of both loops entirely: " x
                [[ $x =  "1" ]] && echo -e " \n in inner, \
                        but breaking out of inner loop, to outer" ; break \
                [[ $x =  "2" ]] && echo -e " \n in inner, \
                        but breaking out of both loops" ; break 2
        done    # for letter
        echo -e " \n still in outer loop"
done # for number
echo -e  " \n Done Looping ! - g'die "
exit 0
```

For real menus : a few approaches

DIY: but VALIDATE

It's wise to validate menu input
- while processing, if it is a short run; can always re-run;
- or even before processing, to avoid wasting time already spent in a run rather than exiting with a fail, just because of possibly accidental input.

1. **Either** have case statement enclosed within a data input validation loop
   Which, again, repeatedly requests input
   until satisfactory,
   or some exit condition / input occurs
2. **Or** precede case with an input data validation loop
   which repeatedly requests input
   until satisfactory,
   or some exit condition / input occurs

Examples on following slides…

For real menus : 1

- **Either** have case statement enclosed within a data input validation loop
  - Which,
    - repeatedly requests input
      - until satisfactory,
      - or some exit condition / input occurs
  - Drawbacks : needs careful configuration & convoluted logic for:
    - valid data
      - Both appropriate handling by case statement
      - And breakout of input data validation loop
    - Invalid data
      - Another pass of validation loop

Examples are given in the following slides…

Input menu validation options…

Case validates within loop

Validation loop
Depends on Case to validate

Consistent control flow (Validation flags) between statements

Case
Valid options
Invalid option
Repeats validation loop by resetting loop control test flag

Loop validates before case

- Validation loop
  - Separate, complete and self-contained data validation set

Consistent data (valid dataset)beween statements

Case
Now assumes all options have been validated in loop.

For real menus : 2

- **Or** precede case with an input data validation loop which repeatedly requests input until satisfactory, or some exit condition / input occurs
  - Drawback…
    - need to co-ordinate valid input with loop and case statement,
      - changes to either must be reflected in the other
  - Solutions
    1. Use the same single array of strings to
       - Store options for menu
       - Provide options for case
    2. Use the bash select command, which
       - Does basically the same as solution 1 above
       - But is not portable across shells

Simple 'one-chance' menu input validation using Case – no loop!

Merely include a wildcard as the last option in a Case statement to account for all other presumed invalid input…
…necessitating all valid options precede this last option.

```
case
    a valid option) command; command;;
    a valid option) command; command;;
    a wildcard) exit program with fail code for invalid input
esac
```

Exiting with fail, would require running the script again, with possible loss, so it may be expedient to cause it to request user to re-enter input.

This entails some spaghetti coding… where logic and control are scattered. This is best avoided if language supports it, as it is a recipe for errors and maintenance & extension difficulties.
At times there are no convenient alternatives, such as in this case, where variables controlling the flow of control are not inside the control statement.

1 – Case statement within data-validation loop

The invalid option will force a repeated entry loop until correct data is input, so it may be wise to include a breakout / exit option, in case the user cannot get the correct data, or is fed up…

```
Valid=false;
Until valid
do
    Input data
    case data in
        a valid option)   command; command;;
        a valid option)   command; command;;
        an exit option)   exit program section (usually a break);;
        a wildcard)       valid=false; continue;
            #continue gives another chance for fat fingered typos
    esac
done
```

Common alternative, as in C-language 'while not(EOF)' idiom

Shown for completeness not confusion.

valid=false
While not(valid)

A similar implementation could be done with while true, as indicated in box.

2 - Precede case with separate (compatible) data validation loop

Option 1

precede case with an input data validation loop
which repeatedly requests input
until satisfactory,
or some exit condition / input occurs

while input is not in the valid input set or exit-code
do
request re-input
Done

Drawback is that valid input set in while must match valid case options

Case input
options in valid input set) command; command ;;
options in valid input set) command; command ;;
exit code option) exit with appropriate code & message
esac

For real menus : 2

- **Or** have case statement enclosed within a data input validation loop
  - Which,
    - repeatedly requests input
      - until satisfactory,
      - or some exit condition / input occurs
  - Drawbacks : needs careful configuration & convoluted logic for:
    - valid data
      - Both appropriate handling by case statement
      - And breakout of input data validation loop
    - Invalid data
      - Another pass of validation loop

Examples are given in the following two slides…

While on a break…!

```
Invalid=true # presetting loop test variable to ensure it will run once
while invalid
do
    read “give up the blather!” blather && invalid=false
    # 1-assuming read succeeds, then use && for #2 clause
    # 2- presume valid, until proved otherwise in case
    # else endless loop : invalid always true
    case blather in
        Patois)     echo “Blah! Blah! In the local patois!?” ;;
        …
        # if it’s still rubbish, then it’s invalid, so reset accordingly
        Rubbish )   echo “Rubbish – tell the truth!” ; invalid=true ;;
        esac
        [[ invalid = false ]] && break ; # but if ok, done by case, breakout
done # while invalid
```


Some things are better left : Until…done!

```
valid=false # presetting loop test variable to ensure it will run once
until valid
do
    read “give up the blather!” blather && valid=true
    # 1-assuming read succeeds, then use && for #2 clause
    # 2- presume valid, until proved otherwise in case
    # else endless loop : valid always false
    case blather in
        patois) echo “Blah! Blah! In the local patois!?” ;;
        …
        # if it’s still rubbish, then it’s invalid, so reset accordingly
        rubbish ) echo “Rubbish – tell the truth!” ; valid=false;;
        esac
        [[ valid = true ]] && break ; # but if ok, done by case, breakout
done # until valid
```


… or by being selective…the menu option!

```
#!/bin/bash
select item in one two three four five
do
    if [ ! -z “$item” ];
    then
        echo “You chose option number $REPLY which is \”$item\””
    else
        echo “$REPLY is not valid.”
    fi    # if [ ! -z “$item” ];
done # select item in one two three four five
# [[  -n “$item”  ]] && echo “You chose…” || echo “$REPLY …”  can replace if…
```

Various indentation approaches exist, but I prefer all the reserved keywords of a control statement indented at the same level, e.g. if…then…else…fi, with all statements internal to the control statement indented further, ensuring that the entire statement can be checked visually at once…

If the indentation block ‘span’ is very long, use comments, folding editors or break up the code using functions.

Dangling else’s & unmatched if’s

- This is less likely to happen when using a good modern block structured approach with:-
  - Indentation
  - Block delimiters
- And is less likely to happen in bash scripting since
  If is terminated with fi
  which is less likely to be overlooked than curly braces { }
- But to err is human, and really mess up is computing.
- Murphy’s law… if anything can go wrong… it uuill!

If … you're iffy about 'if'! …then cover your 'else'!

If <expression> statement-if-true;
else statement-if-false;

Easy enough mistake…
… but can be hard to uncover

'else' is optional but must be attached, or it might attach itself, in ways you might not want!

Nested if…shown below and can be nested more - but messy
- best concentrated / commented on & checked during design, or avoided

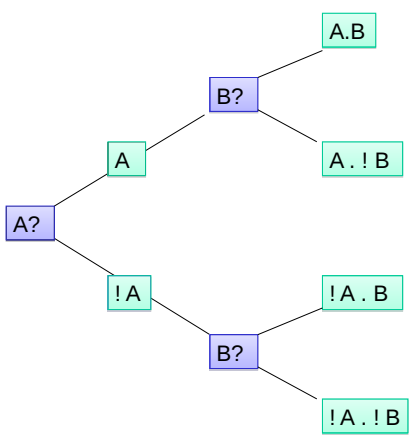
```
If A /*A true*/
        if B    /*B true - combined result: A true, B true  - (A.B) */
        else  /*B false - combined result: A true, B false  - (A.!B) */
Else    /*A false*/
        if B    /*B true - combined result: A false, B true - (!A.B) */
        else  /*B false - combined result: A false, B false - (!A.!B) */
```

Watch - 'the dangling else problem' – 'a loose random else!' – rogue random elses!

- **'else' will always relate to the immediately previous 'accessible & unattached' 'if'** which is **not enclosed in braces or not paired off with another else!**

The random else will match the previous free if which is not locked up or paired off!

Nested if's shown as a tree.



Pairing off: - 'Dangling Else' will grab the nearest previously available 'if'

An 'else' always relates to the immediately previous 'accessible and unattached' 'if'
***not enclosed in braces or not paired off with another else ***

```
                        /*original intentions*/          /*actual results for line 3 missing*/
1   If A                /*A true*/
2       if B            /*B true*/   /*A and B both true*/
3       else            /*B false*/  /*A true, B false*/
4   Else  /*A false… */                                  /*  B false, if line 3 missing  */
5       {if B  }        /*A false, B true*/              /* …and B true – so never executed!  */
6       else            /*A and B both false*/           // (!A.!B) this else now paired with line 1 'if' //
                            // right, but for the wrong reason.. Even harder to spot and debug! //
```

if line 3 were missing, and line 5 totally 'blocked off' enclosed in braces { } so
a) line 6 'else' could not pair with 'if' in line 5, but would be forced to pair with line 1 'if'
b) line would therefore be subject to else (not B) from line 2

so that:-
⇒ The statement in lines 5 would never be executed, since to get to 5 would require A true and B false*, from lines 1, 2 & 4, but the test in line 5 is for B true.
* note also that this condition is quite the opposite of the original intention for 5
⇒ Again, advanced compilers might spot the illogical test sequence - but not all.
⇒ And line 6 would now be executed just on A being false
⇒ To retain the conditions of lines 5-6 being executed as specified within /* */ following each line above - i.e. 5: /*A false,B true*/; 6: /*A and B both false*/ with line 3 missing, then line 2 would need to be entirely enclosed in {}, and those {} in 5 removed or extended to include line 6, so that lines 5 & 6 are treated together as an 'if-else' pair.

Pairing off: - 'Dangling Else' will grab the nearest previously available 'if'

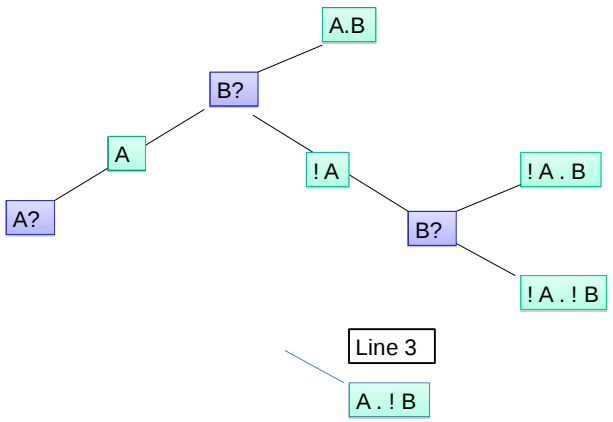
An 'else' always relates to the immediately previous 'accessible and unattached' 'if'
***not enclosed in braces or not paired off with another else *** …BENEFIT OF BLOCK STRUCTURE…!

```
1   If A            /*original intentions*/     /*actual results for line 3 missing*/
2       if B        /*A true, B true*/
3       else        /*A true, B false*/
4   Else                                        // for 3 missing, this else is now paired with line 2 'if' .
                                                // so what follows assumes A true and B false, not A false
5       if B        /*A false, B true*/         /* A.(!B.B) – impossible – **** never executed *****/
6       else         /*A false, B false*/       /* A.(!B.!B) – always executed if line 2 (A.!B) true */
```

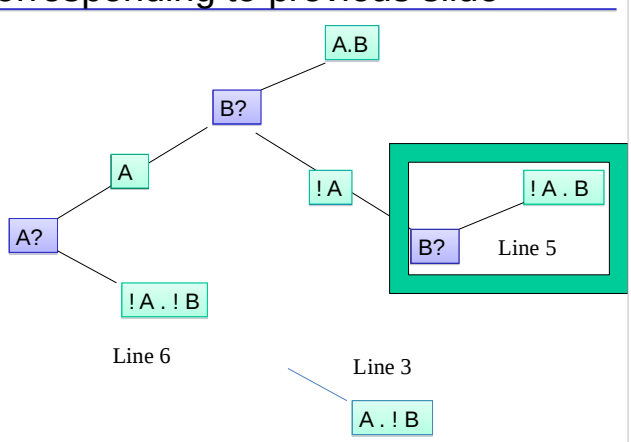
Basically, if line 3 were missing, AND line 2 was not 'BLOCKED OFF' – enclosed in braces etc., then the line 4 'else' would pair with the line 2 'if' resulting in..
⇒ lines 5 & 6 would only be executed under B false implied from lines 2 & 4
⇒ which of course means that line 5 statement would never be done, since line 5's own precondition is B true, so B cannot be false and true at the same place and time
⇒ advanced compilers might spot the illogical test sequence - but not all.
⇒ And line 6 would now be executed just (A.!B) (A true, B false) rather than (!A.!B) (A false also)
⇒ To ensure lines 5-6 will be executed as specified within following /* */ in each line above, if line 3 were missing- i.e.
  ⇒ 5: /*A false,B true*/;
  ⇒ 6: /*A and B both false*/,
then line 2 would need to be entirely enclosed in {},
to avoid the 'else' in line 4 pairing with it,
so forcing the 'else' in line 4 to pair with the 'if' in line 1 as originally intended.

(Easy to make a logical convolution, both of code (and it's explanation) !?)

Corresponding to previous slide



Corresponding to previous slide



If … only I hadn't used nested if statements…

An 'else' always relates to the immediately previous 'accessible ' & 'unattached' 'if'
Accessible = not enclosed in braces
Unattached = not paired off with another else

```
1   If A
2         if B    /*A true, B true*/
3         else  /*A true, B false*/
4   Else
5         if B    /*A false, B true*/
6         else  /*A false, B false*/
```

if line 4 were missing, AND lines 1-3 'BLOCKED OFF' in braces etc, then
⇒ then line 1 'if'statement would terminate at 3
⇒ lines 5 & 6 would be seen as an entirely separate and distinct if-else pair
⇒ Any other single line missing would cause a syntax error from 2 elses etc
Alternatively if lines 1-3 were not 'BLOCKED OFF' in braces etc, then lines 5-6 would be governed by the else in line 3…

And as for all the other mix-ups possible - there's more than
- I'd care to cover!
- And you'd care to study!

The good news is that block structured methods { … }, if … fi, BEGIN … END, tends to overcome these issues, but are still possible, if sufficient omission confuses.

If … only I hadn't used nested if statements…
…except in one fairly standard format…

```
If ( ) { };
Else if ( ) { };
Else if ( ) { };
```

The first true one is taken…
…and all the others aren't even considered…
… now there's a good simple pairing-off strategy!

- Used when you only want to select one option of many
  … or just use switch / case instead for ints, chars & enum types

- But there's still a catch…!!!
  Clearly the conditions must all be mutually exclusive,
  or subsequent tests are illogical and unreachable: e.g.

```
if (n>0) { }
else if (n > 1) {..illogical & unreachable..}
```

since logically (n>1) is already included in (n>0)
and is therefore unreachable as an alternative to (n>0)

If … you can't see the wood for the trees … then …

- get a decision tree!
  - Express it as a binary tree, and you have the structure of your nested loops
  - But be sure to force the correct pairing-off with {}
  - Use indentation as a visual check for your decision tree…

- Simplify the logical conditions
  - Use Boolean algebra if you cover it anywhere - to simplify the tests
  - Or K-maps (Karnaugh maps) a pictorial way of doing Boolean algebra;
    - these are merely rectangular Venn diagrams (remember sets in maths),
    using Grey coding (where adjacent numbers differ only in one bit being true or false)
    for states, so that if states represented by adjacent areas are true,
    then the variable is not needed A+!A =1.
  - Or a logic reduction software program

- Or spell each test condition out fully to avoid any confusion…
  Make out a table of conditions and options
  - May result in slower code, since it requires more testing,
    - but better slow and sure, than quick and tricked (or fast and daft)?
    - Not worth the effort to optimise for speed and minimal tests unless within frequently executed loops as in HPC or data/disk