Testing Times!?

Lots of material here,
mostly standard, simple and obvious
so mostly for reference
most of which, most of you
may know or rarely use…
… so don't worry, be happy!
Easy concepts & principles
details mostly non-examinable!

Testing conditions – last bash at bash (details)

- Program process data, whether state conditions as text or numbers
  - process decisions are data dependent, else why program?
  - Therefore decisions based on tests are required.
  - Last of the tricky detailed overview of inconsistencies
- There are lots more "gotcha's", but will move on to learning by doing
  - just be aware there are syntax inconsistencies which can trap,
- So when developing something new and tricky
  - Test everything on sample data file, or even interactively at CLI
  - And set trace on the bash shell for interactive mode using:- bash – x
- Following are a range of variations you may typically encounter.
  - But you only need to learn to program one simple & safe way
  - Not all are recommended, safe or secure, including specifications
- Try to follow
  - best practice if (re-?)writing a new script,
  - Consistent style if modifying an older one, to avoid chaos

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 logical operators **&&** and **||**

❑Success denoted by either true = 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, in some versions the only 1!
- NB Test for -ne 0, and not just testing for 1 of many fail codes!
  - i.e. use [$? –ne 0] rather than [$? –eq 1] as 2, 3, 4 … also fail

❑Command may also be a logical expression which is checked for truth by
- ***test*** *cmd* or equivalent single square bracket syntax *[command]*
  with a space between the parentheses and enclosed command,
  - […] are best avoided as has confusing constraints, and cryptic.
  - Original sh version, was a POSIX compliant command
- Other initially non-standard 'reserved words' or compound commands
  - [[…]] – new compound command test in bash
    – with extra features, but not POSIX compliant or portable
  - ((…)) – for arithmetic tests, really evaluation ~ slide 20 +/- 1

Decision Making

The Bourne shell has many ways to test :-

❑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

Testing conditions - Exit status of commands…

- Commands may be joined with **&&** for **AND** or **||** for **OR** ,

BUT relates to their execution : (con/dis)-junctive ops. NOT just logic
so unlike simple logic tests, each command can actually do something !!!

- **&&** – means both are executed, only if both are true (successful)
  - a sequence of all commands linked with &&
    will run until the first in the sequence fails.
- **||** – means only one true (successful) command need be executed
  - a sequence of all commands linked with ||
    will attempt to run until 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.
- Expressions may also be combined with **–a** for **AND** or **–o** for **OR**
- But POSIX & bash won't work with more than 4 arguments
  - And can't exceed 4 arguments, not conditions,…
  - tokens -n -a -o count as arguments!! so better using && and ||
- Usual language tradeoff : expressiveness - errors; concise - confuse.

test lets shell make TRUE or FALSE decisions

- returns zero exit status if condition evaluates to TRUE
- often used to test in an *if*, *while*, or *until* command
- POSIX does not require it to work with more than 4
- has two possible formats
  - *test condition*
  - [*condition*] - keep test space around the brackets
    - Memory aid : does it square up --> square brackets
    - Suggested to avoid as cryptic, & error prone, but
      - tradition, so need to know:
      - & short => so handy – easy to code & read!
  - where *condition* is an operator or a set of operators ANDed and/or ORed together

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

- S (upper case) checks if file is a Socket for interprocess communication.

TEST in Quotes in case of null 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 quoted) works even if $string is null,
  - The system realises it is dealing with a null string
- But unquoted
  - $string will usually not work correctly if $string is null
    - The system interprets it exactly as a null string name
      - Often humanly read as a null string variable value

Although the reported errors will differ across shells,
these errors are best avoided,
achieved simply by quoting the string variable
“$string”

String Operators – n is for non-zero, not null!

Operator	Returns ***TRUE*** *(zero exit status) if*
string	String exists... ie is not null, of nonzero length-> as next below
-n string	String is nonzero ... ie exists, not null -> as for case above
-z string	String is zero length ... null string
$string_1$ = $string_2$	Identical strings
$string_1$!= $string_2$	Different strings

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

Lengths:- -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:-

NB test "string", with no flag, defaults to –n => nonzero length,

```
cs1> test "string" ; echo $?   # "string" is not null, test succeeds
0
cs1> test "" ; echo $?         # "" is null, so default -n (nonzero) test fails
1
cs1> test -n "" ; echo $?      # as above, except -n is explicit
1
cs1> [ -n "" ] ; echo $?       # square bracket syntax...for test
1
cs1> [ -z "" ] ; echo $?       # "" is null, -z zero test succeeds, returns 0
0
cs1> test -z "" ; echo $?      # as above, but uses test syntax
0
```

NB need spaces between […] and internal test !!!

String-ing along

Note string concatenation on first 2 lines of output, $null vs $notnull
With identifying text immediately concatenated onto variables above
Not the most elegant code
But quick, short & handy

Memory aid:-
n/z – non/zero length

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

Outputs:-

```
null
abcnotnull
-n "$null?" exit status 1
-n "$notnull?" exit status 0
-z "$null?" exit status 0
-z "$notnull?" exit status 1
```


Lexical comparison… alphabetic sort

In bash, the < and > symbols used in comparison,
Must be escaped with \
to prevent their being interpreted as redirection operators;
Not to be confused with regex word boundary designators
(A bit like Alice in wonderland : means whatever I want it to!)

```
str1=abc
str2=def            # syntax autocolouring thinks reserved 'def' !
test "$str1" \< "$str2"
echo $?
0
test "$str1" \> "$str2"
echo $?
1
```

Integer Comparison Operators

- strings came first and used numeric 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

Like Negative Logic... since 0 indicates OK

A	B	AND	OR
1	1	1	1
0	1	1	0
1	0	1	0
0	0	0	0

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 as : ($expr_1$ -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

as an ordinary file (or symbolic link!)

(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

Strings or Numbers

Shell assumes text, unless told otherwise!

- 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
  juxtaposition is sufficient for concatenation in bash
  no need for '+' operator, it too will be added to string!
- At first sight, the shell appears useless for sums!

Numbers… !? LET it know! (new exts.)

Simple & basic

Let : it know, let it know, let it know… let c=a+b

- And no need for
  - Either $ to access values
  - Or spaces around operators:- [ ] (( ))
    otherwise it will just parse them as tokens
    examples on next slide...

- Also alternative numerics with
  - expr : this and all above are integer only
  - bc : basic calculator ; arbitrary precision

No need for $!!! ‘let’ just lets it know!

- Simple & basic

```
a=1
b=2
let c=a+b    NB no spaces around operators
             else it will parse them as tokens!
```

- Similar to (newer) C & derivatives (old BASIC)

```
let c++     equivalent to   c=c+1 autoincrement
let c--     equivalent to   c=c-1 autodecrement
let c+=5    equivalent to   c=c+5
let c-=5    equivalent to   c=c=-5
```


Brackets [ ] or (( )) for shell arithmetic

- $ are
  - either essential : on RHS of = and outside (()) [ ]
  - or optional : if inside [ ] test & (()) - arithmetic
- Spaces
  - Around ‘=’ assignment : forbidden
  - Between variables and operators: optional
  - Except when using [ ] as test… see slide 4
- Assume all below follow a=1; b=2
- Applies to both [ ] and (( )) (on bash 5.03)
  - c=$[a + b] c=$[a+b]
  - c=$((a + b)) c=$((a+b))
  - c=$[$a + 5] c=$[$a + 5]
  - c=$(($a + b)) c=$(($a+b))

Interpret $(…) as command substitution like backticks
But $((…)) as C-like arithmetic expression/expansion

Expr – integer arithmetic only!

- The *expr* command "evaluates" it's arguments and writes it's output on STDOUT (standard output_
- Note, since it is evaluating arguments, they **must** be separated by spaces (or $IFS) to be parsed properly and taken as separate arguments.
- Also, expr only works with integer arithmetic expressions => so values are truncated (decimal chopped & dropped)
  - **expr** $a + 2 note spaces around ‘+’
  - 3
  - **expr** 7 / 2 note spaces around ‘/’
  - 3 integer division
  - **expr** $a * 2 backslash to avoid * shell expansion
  - 2

Expr – can be combined with brackets

- Since expr is a command, can issue with backticks
  - c=`expr $a + 5` using backticks to issue cmd
- But can also issue using single round brackets like value of algebraic expression or entity in regex
  - c=$(expr $a + 5) and $ to retrieve result
- But cannot issue using
  - [ ] : Either single square brackets
  - (( )) : Or double round

  Need backticks within either above
  - c=$[`expr $a + 5`] Note spaces need only be
  - c=$((`expr $a + 5`)) around operators within expr
- Also, expr only works with integer arithmetic expressions => so values are truncated (decimal chopped & dropped)

basic calculator exists for ***REAL*** numbers - **bc**

- Supports arbitrary precision ***reals (floating-point)***
- In fact, it is a small math programming language, which first takes program files, before std input i.e. reading from terminal – only a glance at it here
- So expression is piped to bc, or read from a file
- For more info check man or info on bc
- Basic operations: + - * % / ^
  - NB / needs scale to specify significant decimal digits
  - echo "scale=5;1/3" | bc
  - .33333
  - a=1.2;b=2.3
  - echo "$a+$b" |bc
  - 3.5
- For more info check usual sources.

What's another test!?

Evaluations of evaluations (double brackes) as tests [[…]] ((…))

- **[[…]]**
  - upgraded ***test*** – introduced by ksh, now zsh & bash support
  - Unlike ***test*** it
    - is not a built-in command, but a compound command as a reserved word and part of shell grammar
    - Does not parse as a built in command between [[…]]
      - Parameters are expanded
      - But no word splitting or filename expansion
      - Works with spaces between tokens and without quotes (unlike test [ ])
    - Supports the same operators as ***test***
      - + some enhancements & additions
    - Is not POSIX or portable, ***test*** is
- NB **[…]** is a simpler form like test, without enhancements
- But either is shorter than if.. then.. else.. fi - for slides
  - compact for presentations – but use test for better code.

Non-POSIX standard enhancements to **[[…]] -** What does this all mean:-

- Does not parse as a built in command between [[…]]
  - Parameters are expanded
  - But no word splitting or filename expansion
    - [[$x == a*]] tests if x starts with a,
    - but [$x == a*] will expand a* to a filelist of all files starting with 'a' in the current directory
- Works with token spaces and without quotes (unlike test [ ])
- Works with spaces in filenames e.g. filename=my file
  - [[-f $filename]] ok,
    but not [-f $filename] which splits as 2 tokens with no quotes
    but [-f "$filename"] is ok, as " " inhibit shell expansion
    " " removes meaning of all enclosed chars EXCEPT $, \ and `
    whereas single qutoes ' ' removes all,
  - so need to use double " "

Which should you use…

- [ ] or test
  - POSIX for portability
  - May be easier, since it follows normal command conventions
  - But not guaranteed to work with more than 4 arguments, notice arguments, not conditions … eg. [-n $a -a -n $b] is 5!
- [[ ]]
  - If only using bash…
    - avoids some problems with spaces etc in filenames
      - which you should not have anyway…
      - Unless files imported from another…OS
- And there are many more issues and complications…
- Will Avoid as (a bit of a mess!)
  - Too much info, too detailed and confusing
  - most are implementation dependent
- But ALL tend to use what's handy and fast…! So lots of [[ ]] !

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 arithmetic evaluation **((…))** success Booleans

((arithmetic expression)) returns exit-status of false for 0 result, true otherwise!
NB Boolean process success true for normal exit status of 0, else false 1, non-zero exit
But here: if arithmetic expression ==0 success Boolean is false (1); else true (0)
Memory aid: false if nothing between the ears – like brackets! (( )) - or normal C!

- Sample runs with a =0,1

(($a-1)); echo	a-value $a	arithmetic (($a-1))	exit status $?
	a-value 0	arithmetic -1	exit status 0
	a-value 1	arithmetic 0	exit status 1

Double take on this logic… $ up front gives value, not truth – cynical or what?

test $**((** $a – 3**))** -ne 0
If $a – 3 = 0, then the arithmetic expression is 0,
but $(($a-3)) merely returns the value of the arithmetic = 0 in this case,
So 0 –ne 0 is false, which in bash boolean test value is 1

Sample runs with a =2,3

$ test $(($a-3)) -ne 0; echo a-value $a arithmetic $(($a-3)) exit status $?

a-value 2	arithmetic -1	exit status 0
a-value 3	arithmetic 0	exit status 1

- Again, using **if…fi** is cleaner & safer, unless absolutely sure of details.
- NB both sets of parenthesis must be used together – one operator, else wrong...
  - i.e. Don't think of as maths algebra splitting ((…)) into X = (…) so taking (X)

Bracket & test review

An if statement typically looks like

```
if commands1
then
    commands2
else
    commands3
fi
```

The then clause is executed if the exit code of commands1 is zero.
If the exit code is nonzero, then the else clause is executed.
commands1 can be simple or complex.
It can, for example, be a sequence of one or more pipelines
separated by one of the operators ;, &, &&, or ||.
NB exit code of whole statement sequence, is that of last executed!

Bracket & test review

All conditions below set an exit code (0 on success or true!), for the if decision

- if [condition]
  - traditional shell test command, available on all POSIX shells.
- if [[condition]]
  - new upgraded variation on test from *ksh* that *bash* and *zsh* also support. Extended features include test if a string matches a regular expression.
- if ((condition))
  - Another *ksh* extension that *bash* and *zsh* also support, does arithmetic. It returns an exit code of zero (true) if the result of the arithmetic calculation is nonzero. Like [[...]], this form is not POSIX and therefore not portable.
- if (command)
  - runs command in a subshell, typically to limit side-effects
    if command required variable assignments or other changes to the shell's environment, which cease after the subshell completes.
- if command
  - command is executed and the if statement acts according to its exit code.

Flow & Control

If … then … fi

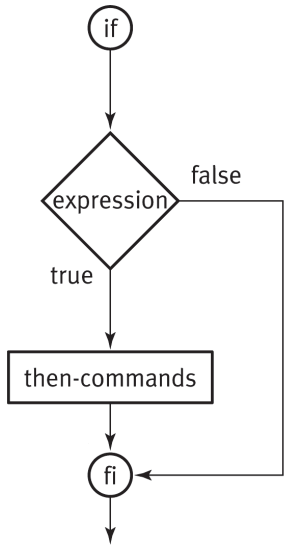


Figure 15.1 Semantics of the `if-then-fi` statement

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

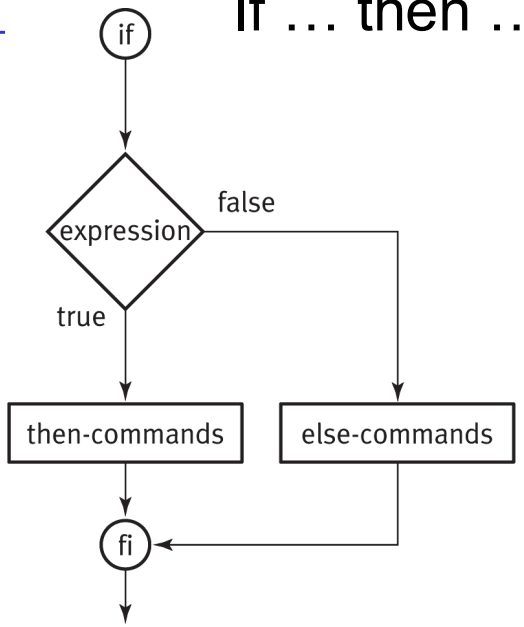


Figure 15.2 Semantics of the `if-then-else-fi` statement

Random samples… from former sysadmin!

```
if [[ `hostname` != "cs1" ]]
then
  echo -e "############################"
  echo -e "This script only runs on cs1"
  echo -e "need to be on d'right horse boy!"
  echo -e "offloading now!"
  echo -e "############################"
  exit 1
fi
```

Will run with
Either [[expr]]
Or [expr]
With if... then ... fi

But needs
[[expr]]
To run with && etc

```
[[ `hostname` != "cs1" ]] &&
{
  echo -e "############################"
  echo -e "This script only runs on cs1"
  echo -e "need to be on d'right horse boy!"
  echo -e "offloading now!"
  echo -e "############################"
  exit 1
}
```


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

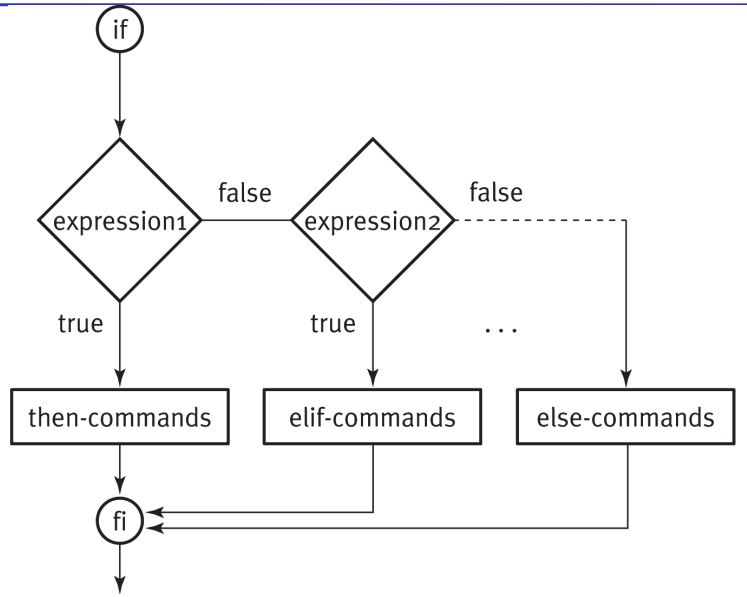


Figure 15.3 Semantics of the `if-then-elif-else-fi` statement

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
```


More Testing Times!?

Testing conditions - Exit status of commands…

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

BUT && and || relate to their execution success rather than logic
NOT JUST A COURT CASE, BUT AN EXECUTION, CHANGES THINGS!!

  - **&&** – 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 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**
  - Or in words, if $command_1$ succeeds then do $command_2$ also
- $command_1$ || $command_2$
  - $command_2$ executes only if $command_1$ returns a non-zero status (did not execute successfully)
  - Logically like … **If NOT**(*command*$_1$) **then** *command*$_2$ **fi** ..
  - Or in words, if $command_1$ fails then do $command_2$ instead.
- Examples

who | grep ”your_id" > /dev/null && echo ”You are logged on"

who | grep ”Noah" > /dev/null || \

echo ”Noah was supported by logs – but 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

- Or reverting to our double take double bracket arithmetic test

```
a=1
(($a-1)) && echo true || echo false
false
a=2
(($a-1)) && echo true || echo false
true
```


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 unnecessary 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 with computer insecurity and vulnerabilities, the impossible might just happen!? ;-))

Even if the test above were done correctly, it is still pointlessly repetitive

e.g. **test -f "$times" –a $test –eq 0**

test -x bin/file -o $test -gt 0

This will test

Test IF bin/file is executable OR IF an error occurs in the test…e.g. file missing

Simple input data validation – no second chance!

Read & check input, either from file or command arguments $[1-9]:

```
#!/bin/bash
echo "Please enter your name:-"
read name
if [ -z $name ]       # if $name is null - none entered
then
  echo "No name entered" >&2   # redirect error msg to stderr
  exit 1                 # exit with failed (non-zero) return code
fi                       # so next script can test before running
```

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

Using a loop without a limit counter might result in an infinite loop, if data input were from a faulty process...

Input validation – potential 'infinite loop' of chances...

request a re-entry until data OK, might limit to a few loops to avoid endless.

NB normal pretest loop practice is
to precede the while test with an attempted initiation,
otherwise the initial value for the variable 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.
(may risk an infinite loop… but you can deal with that, with limits on the number of tries, or time, or just interrupt or kill the process...

```
#!/bin/bash
while [ -z "$name" ]
do
  echo Please enter a name…
  read name
done
```

…and more obtuse (but typical use) still…

- To check for a directory and cd into it if it exists, use this, or [[ ]] equivalent:
  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"      # no need for backticks to issue cmd
Fi                  # $PWD is an environment variable...
```

  - Cryptic form: **[[-d "$dir"]] && cd "$dir" && echo "$PWD"**
    - Concise & compact – probably why it is used..

Compounded extensions of && and ||!

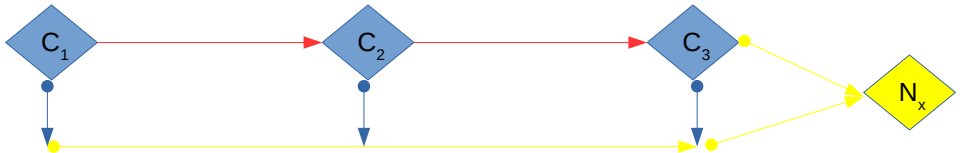
Consider each of the following:-

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

Graphical representations of && and ||!

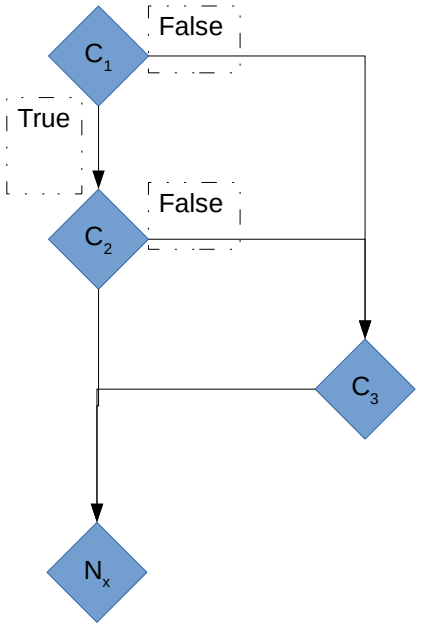


Yellow is for a sunny future!

1. cmd_1 && cmd_2 && cmd_3 ; nx_cmd # run all until first fails!
   - cmd_3 will only be run if the previous two are true/succeed
   - In the conjunctive (&&) case (like a sequence of 'then if's):-
     - the red line is for success or true,
     - the blue line is for fail or false

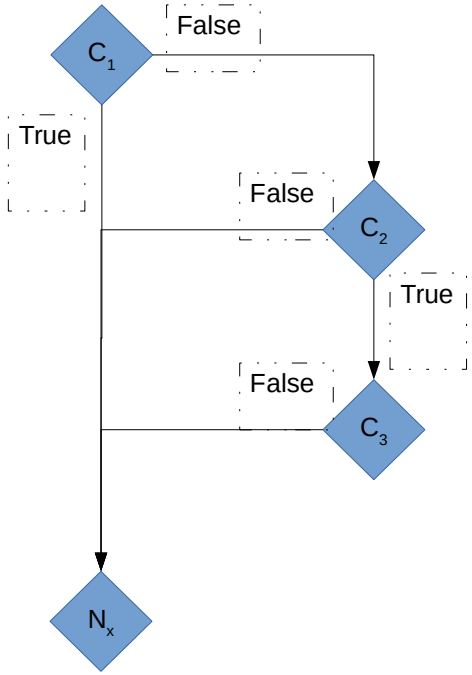
2. cmd_1 || cmd_2 || cmd_3 ; nx_cmd # attempt all, until first succeeds!
   - cmd_3 will only be run if the previous two are false/fail
   - In the disjunctive (||) case (like a sequence of 'elif / else if's) :-
     - the blue line is for success or true,
     - the red line is for fail or false

Graphical representations : cmd_1 && cmd_2 || cmd_3



cmd_3 will only be run if either of the previous two are false/fail

Graphical representations : cmd_1 || cmd_2 && cmd_3



cmd_3 will only be run
if cmd_1 is false
and cmd_2 is true

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 …

Clearer reading & coding of && & || !

- Combinations : c_1 && … c_n || d_1 && … d_n || e_1 && … e_n
- Can be read
  - Within a letter group (all c's or d's etc) conjoined with &&
    - try letter sequence until one fails,
    - If one fails, letter group fails, else group succeeds!
  - A whole letter group – groups of same letter conjoined with ||
    - Try each in sequence until one succeeds,
    - If one succeeds, full statement succeeds, else it fails!
- Reverse Combinations : c_1 ||… c_n && d_1 || … d_n && e_1 || … e_n
- need braces : { c_1 ||… c_n } && { d_1 || … d_n } && { e_1 || … e_n }

Otherwise, for unbraced :

- if all c's fail until the last c
- Then if last c succeeds, then it will attempt first d & proceed
- Else if last c fails, then it will skip first d & proceed… wrong!!!

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 without thinking, you might extend it like this
    c_1 && c_2 ; c_x

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

if c_1
  then c_2
fi
c_x

… if this was the intention...

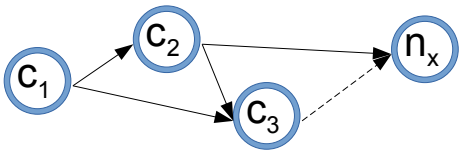
if c_1
  then {c_2 ; c_x }
fi

… achievable simply by …

c_1 && { c_2 ; c_x }

Gotchas! … even more complex, confusing & common!

- Careful about lists, even those not enclosed within { ; }
  If the original valid correct expression is ..c_1 && c_2 || c_3 then :
  IF c_1 THEN c_2 ELSE c_3 FI IS NOT A CORRECT RE-EXPRESSION
  NB the statement is executed with && || so state diags. help
- The correct expression in typical if...then...else..fi is on right,with a simple state diagram shown below.
- Outgoing solid edges show true|up OR false| down branch
- Dotted edges are for either



Pseudocode here
Not bash syntax

```
if c1
    then
        if c2
            Then break 2
            (out of both if's)
            else c3
        fi ( c2 )
    else c3
fi ( c1 )
nx
```

Or restructure:-
if NOT (c1 AND c2)
then c3
fi
nx (next stmnt.)

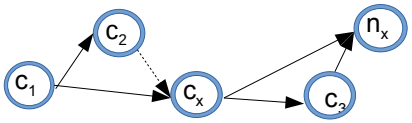
Gotchas! … even more complex, confusing & common!

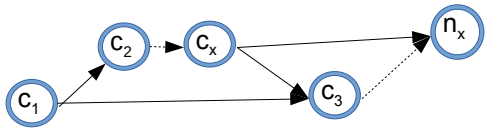
- Careful about lists, especially editing extensions…enclose within { ; }
  - e.g., if the original expression was ...c_1 && c_2 || c_3
  - And you wanted to include another command with c_2 , then again without thinking, you might extend it like this : - c_1 && c_2 ; c_x || c_3

c_1 && c_2 ; c_x || c_3
… which, logically, is a mess
…
if c_1
    then c_2
fi
if !C_x (but with C_x executed)
    then c_3
fi

… if the following was intended…
… achievable simply by …
c_1 && { c_2 ; c_x } || c_3

if c_1
    then {c_2 ; c_x }
    else c_3
fi





Case

much easier than convoluted tests!

Python caught up at last with match and case

Most languages have different syntax – egos don't agree!

Case – note : two semicolons terminate cases

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…
- Another instruction which supports menus is select but it only takes single word options and is non-standard/portable across shells,
- NB lists of commands within an option are separated by ;
  BUT terminated by ;;

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

Loopers

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 fails!
   *)          true ;;
esac
```


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

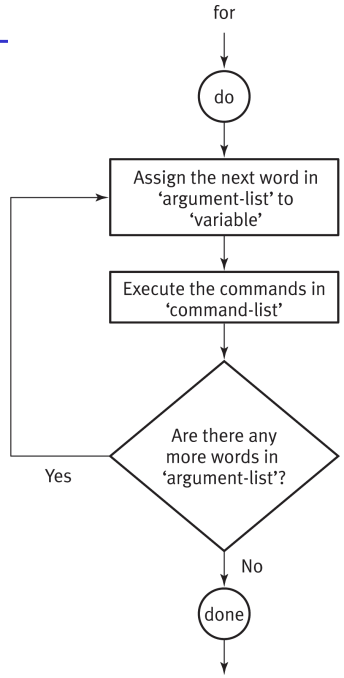


Figure 15.4 Semantics of the `for` statement

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
```

Arrays

```
a[1]=foo
echo ${a[1]}
foo

for ((expr; expr; expr))

# Initialize array
for i in {0..23}; do hours[i]=0; done
```

Remember from first bash slide deck

$* is a bit of a bags, usually better with "$@"

Arrays – quick flick!

Set value `a[1]=foo`
Access `echo ${a[1]}`
`foo`

Initialize array

```
for i in {0..23}; do hours[i]=0; done
```

Alternative

```
animals=("a dog" "a cat" "a fish")
for i in "${animals[@]}"; do echo $i; done
a dog
a cat
a fish
```

Typical issues with *, @ and quotes re arguments apply

Loops – typical application

```
for ((expr; expr; expr))
```

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

A simple search script:

```
#!/bin/sh
# first
# This file looks through all the files in the current directory for file in *
# 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...
-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.
-l list only filenames with matches, not every match,
Suppress all other output

But **grep -RI SEARCHSTRING** -would of course do it recursively!

Which is why all the string processing tools (tr, sed, grep, awk) done first!

while

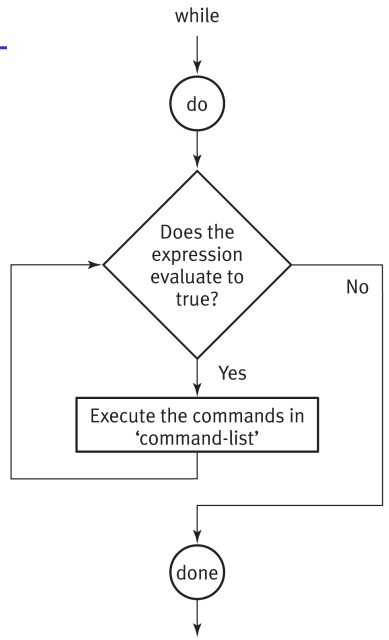


Figure 15.5 Semantics of the `while` statement

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 $command_t$

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!
```


until

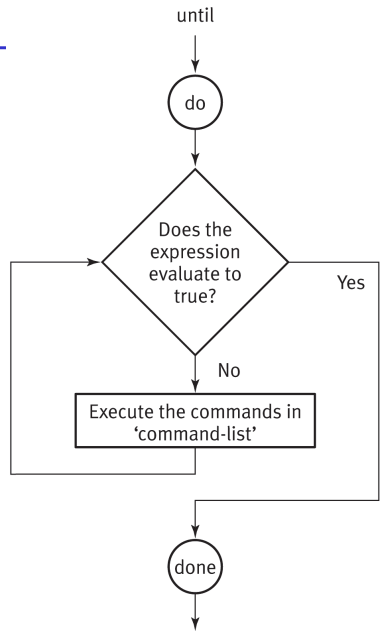


Figure 15.6 Semantics of the `until` statement

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 Boolean status may certainly be used instead

Break or continue

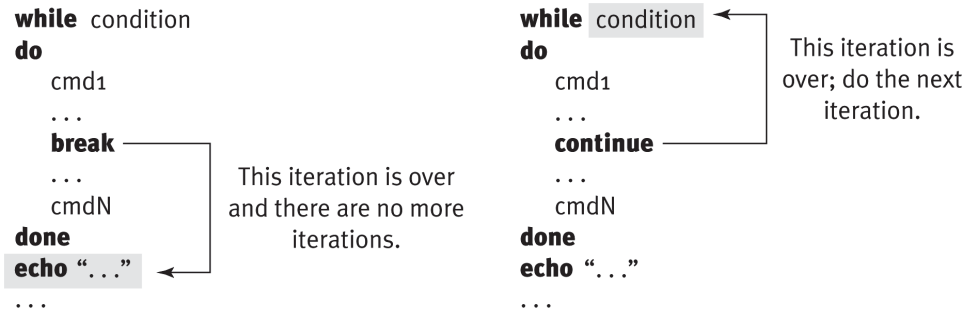


Figure 15.7 Semantics of the `break` and `continue` commands

Dealing with HABITs & addiction:-
- break ... .be done with it
- continue – after skipping this round!

I want to Break free!? or continue in drudgery!?

- Break
  - The habit for once and for all
  - free out of same-old same-old loops entirely
    - Conditions may arise to get out and stay out
    - *break* allows you to do this
- Continue
  - With the same old habit,
  - after a short reprieve - pass on the rest of this round
- But continue with the same-old loop restart at the next pass
  - Sometimes, in the middle of executing a pass through a loop the need arises to skip the rest of the commands in this pass, rather than breaking out of the loop forever,
  - *continue* will let you do that

Breaking out…. To any level...>!!!

Read flags
-n1 – no of chars to read = 1
-p – display prompt for input expected from terminal

```
#!/bin/bash
# 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 ! "
exit 0
```

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 logical 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
  - Useful when your script calls other scripts,
    - exit codes can incorporate discreet and discrete uniform communication of errors for handling in following scripts

Whats on the

menu

For real menus : a few approaches

DIY menus: but VALIDATE the input (issues arise in some assignments)

- will also see a handy 'select' statement but only has one-word options
- And getopt(s) as another way to validate arguments for a command.

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

Regardless it is wise to co-ordinate and validate input with program actions, so

Either have case statement which decides on actions, also validate data, within a single loop which, repeatedly requests input until valid, and/or executes actions until some exit condition / input occurs

Or precede the action cases with a separate input data validation loop, possibly also with a case statement which again repeatedly requests input until satisfactory, or some exit condition / input occurs

Both need consistent and co-ordinated logic between validation and action OR use a data structure e.g. list to effect equal input and action choices

Input menu validation options…

Validation: case in test loop

Validation loop

Depends on Case to validate

Needs consistent control flow logic between statements

Case

Valid options

Invalid option

Repeats validation loop by resetting loop control test flag

Validation Loop before case

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

Needs consistent data comparison beween statements

Case

Now assumes valid data

all options have been validated in loop.

Risks of costly loss of data runs
Above from program errors
Left from correct input unavailable!

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 – may be case dependent!
      - 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…

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; valid=true (or break 2 levels);;
        a valid option)   command; command; valid=true (or break 2 levels);;
        an exit option)   exit program section (usually a break out);;
        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)
```

Other implementation could be done with while invalid, as indicated in box.

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

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

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
```

Drawback is that valid input set in while must match valid case options,
Which is easy if code blocks are close
But may be missed if in data init. block

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…

Doing a do...while with a while … do

```
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!?"
                    actions; actions; and more actions ;;
        …
        # if it's still rubbish, then it's invalid, so reset accordingly
        rubbish )   echo "rubbish – tell the truth!" ; invalid=true ;;
    esac
done # while invalid
```


Doing a do… until with ... 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!?";
                actions; actions; and more actions ;;
        …
        # if it's still rubbish, then it's invalid, so reset accordingly
        rubbish ) echo "Rubbish – tell the truth!" ; valid=false;;
    esac
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 it makes sense with this syntax to have all 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 at a glance. Select is fine for simple single word menu option responses, otherwise… case

Script argument
(data passed at start of script)
validation
using getopt(s)
s indicates the bash version!

Argument processing via getops – just be aware it's there!

- getopt(s) - can
  - Parse arguments – returns $1, $2 etc. - alternative to $* & $@
  - Partially validate arguments and flags– alternative to case for validation
    - Often used with case
- As with much software, particularly open source, where there is freedom and divergence, best to check operation on your system
- Don't assume portability
- Don't even assume compatibility with specs.
- Don't assume anyone knows, as things change!
- Although GNU getopt is largely compatible with getopts and not modified since 2005, still fake news on many sites

Differences between the getopt(s)?

As usual – differences...

- There is a lot of baloney/confusion on the net about features and compatibility claims...
- Most main modern Linux distros and their derivatives (all you're likely to encounter!) ship with a getopt that supports long arguments and whitespace, if not forced into traditional mode
- Rather than getting more getopt(s) gotchas, bypass and look at simple use.

Recommendation…

- Use traditional $* & $@ for portability

getopts optstring name [arg]

```
#!/bin/bash
while getopts ":a:bc:" OPTION ;do   # OPTION here for clarity, flag in other slides
   echo "flag -$OPTION, Argument $OPTARG";
done
```

- while loop,
  - iterates through arguments that match given optstring, in this case, "a:bc:",
  - and stores the value of the flag in the variable OPTION -
    - If the flag has an associated argument, it is stored in OPTARG.

Works as follows:

1) Needs a new call to getopts to get the next argument
2) For every option letter,
   1) getopts stores the option in the variable 'OPTION'
   2) If followed by a colon, it expects an argument, stored in OPTARG.
3) If getopts expects an argument, but could not parse one, it prints an error.
4) If it was not expecting one, OPTARG will be initialized to ""(an empty string), and in Ubuntu 20:04 did not parse the string further, but baled and failed.
5) If the very first character of optstring was ":"(a colon), then no error message.

while getopts ":a:bc:" flag;
echo "flag -$flag, Argument $OPTARG";

Running as specified – really “?

$ ./getoptsDemo -a ant -b -c cat
flag -a, Argument ant
flag -b, Argument
flag -c, Argument cat

Running with an unrequested parameter – gotcha!

./getoptsDemo -a ant -b badger -c cat
flag -a, Argument ant
flag -b, Argument

Irrespective of error reporting
Not expecting badger for -b, wrong but no warning,
Even with error reporting on : gone from first char “ a:bc:”
Returns blank string for -b parm instead of badger
But does not recover to parse rest of string -c cat

while getopts ":a:bc:" flag;
echo "flag -$flag, Argument $OPTARG";

Running without a required parameter for a flag :- a:

$ ./getoptsDemo -a
flag -:, Argument a

Error reporting off :

returns ‘:’ for the flag, and a as the argument

Error reporting on :

./getoptsDemo: option requires an argument -- a
flag -?, Argument

Case partially made…

- The valid options and the invalid errors reported as : & ? can help script argument validation with a case statement
- often used with them for that very purpose.
- case statements then used to implement various options.

while getopts ":a:bc:" flag;
echo "flag -$flag, Argument $OPTARG";

Running without a listed flag – it’s OK – they’re optional!

$ ./getoptsDemo -a arga -b
flag -a, Argument arga
flag -b, Argument

Note: irrespective of error reporting
OPTARG for -b is blank, does not retain past values.
Also, -c absent, which is OK, as each flag is optional.

Running with an unlisted flag – now that’s not an option!

$ ./getoptsDemo -d
flag -?, Argument d

Not expecting a ‘d’ !
Error reporting off… returns ‘?’ for the flag, and d as the argument
Error reporting on…
./getoptsDemo: illegal option -- d
flag -?, Argument

Check the facts

- Can’t believe half the lies you hear!
- As an example of persistent prejudice, ignorance from narrow-mindedness
  - There are still misrepresentations of getopt(s) from both sides... after 15 yrs...
- No particular interest, what gets the job done
- Depends on convenience vs portability needs

getopt(s) - alternative to $* & $@

Alleged discrepancies of the getopt(s)?

Ignore, including getopts tutorial listed e.g.

https://wiki.bash-hackers.org/howto/getopts_tutorial

Features	Get opt	Get opts
Handle empty flags arguments, strings or whitespacen	n	y
Included in sh & bash, rather than needing installation (but as a bundled utility package not really part of (ba)?sh)	n	y
Allows long options ie. --long_option vs -l	y	n
Simpler syntax	n	y

Getopt - to help shell scripts parse command-line parameters.

This package contains a reimplementation of getopt(1).

HIGHLIGHTS

It can do anything that the GNU getopt(3) routines can do.

It can cope with spaces and shell metacharacters within arguments.

It can parse long parameters.

It can shuffle parameters, so you can mix options and other parameters on the command-line.

It can be easily identified as an enhanced getopt(1) from within shell scripts.

It can report parse errors as coming from the shell script.

It is fully compatible with other getopt(1) implementations.

Tradition... not like it was yesterday!

COPYING

This program comes under the GNU general public license version 2.

See the file COPYING included in this package.

Note that though you may freely copy it, it is copyright (c) 1997-2005 by Frodo Looijaard <frodo@frodo.looijaard.name>.

Files in the gnu directory are from glibc-2.0.4: copyright (C) 1987, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97 Free Software Foundation, Inc.

Changelog /usr/share/doc/util-linux

- 20051107: Bumped up version number to 1.1.4
- 20051107: Makefile: package target
- 20051107: Changed email and website to current ones
- 20051107: Fixed a few typos in the manpage
- 20030123: Bumped up version number to 1.1.3

... 5 years....

- 19980603: Bumped up version number to 1.0.1
- 19980603: Fixed sizeof() bug (should be strlen) in getopt.c, thanks to Bob
- 19980505: Changed date field in LSM to proper syntax
- 19980505: Released version 1.0

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!
- And each attempted fix, breaks something else!?

Congested if's

Dangling else's & unmatched if's

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

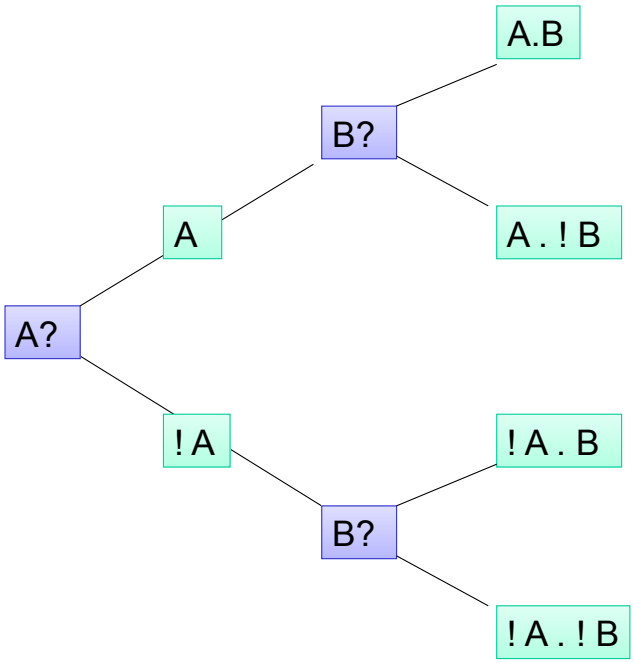
```
If A /*A true*/
        if B    /*B true - combined result: A true, B true  - (A.B) */
        else    /*B false - combined result: A true, B false  - (A.!B) */
Else    /*A false*/
        if B    /*B true - combined result: A false, B true - (!A.B) */
        else    /*B false - combined result: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

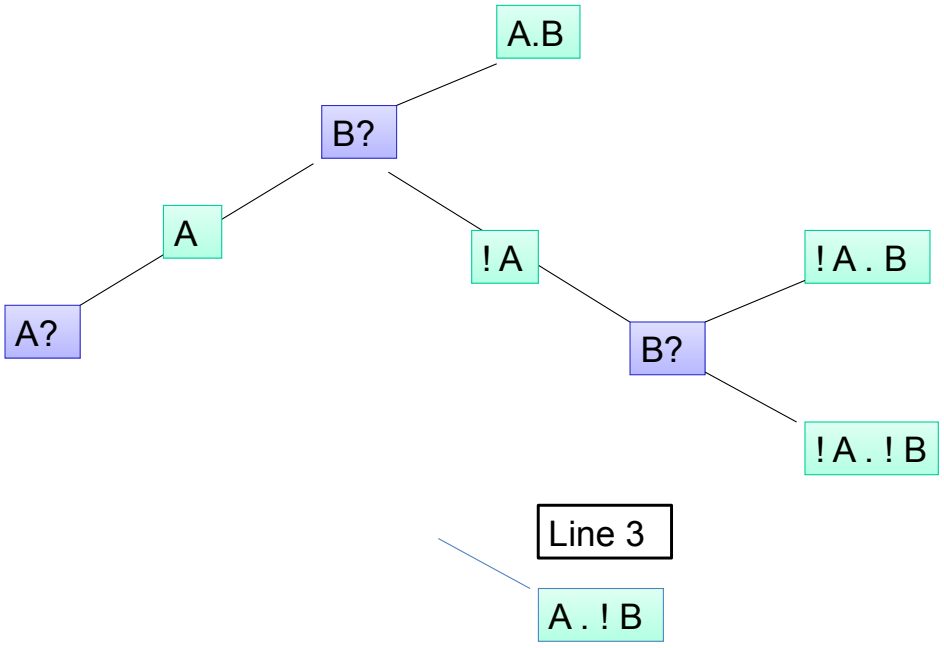
Basically, if line 3 were missing, AND line 2 was not 'BLOCKED OFF' – enclosed in braces etc., then the line 4 'else' would pair with the line 2 'if' resulting in..

- lines 5 & 6 would only be executed under B false implied from lines 2 & 4
  which of course means that line 5 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



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:-

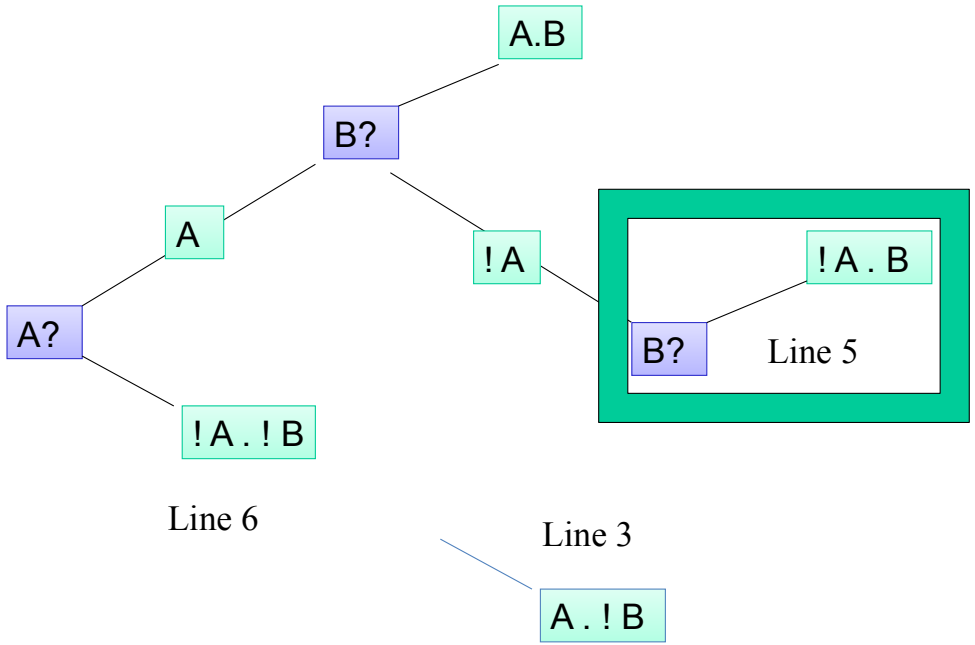
⇒ The statement in lines 5 would never be executed, since to get to 5 would require A true and B false*, from lines 1, 2 & 4, but the test in line 5 is for B true.
  * note also that this condition is quite the opposite of the original intention for 5

⇒ 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.

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 case/switch like 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