Corso facilitato di bash scripting

By CasertaGLUG per informazioni contattare l'autore casertaglug-owner@autistici.org

ESEMPI ESSENZIALI DI SCRIPTING

Sezione Sommario

Bibliografia

A cura di Peter Denning, *Computers Under Attack: Intruders, Worms, and Viruses*, ACM Press, 1990, 0-201-53067-8.

Questo compendio contiene due articoli su virus in forma di script di shell.

*

Ken Burtch, *Linux Shell Scripting with Bash*, 1st edition, Sams Publishing (Pearson), 2004, 0672326426.

Tratta molti degli stessi argomenti di questa guida. Tuttavia, è una pubblicazione di una certa utilità.

*

Dale Dougherty e Arnold Robbins, *Sed and Awk*, 2nd edition, O'Reilly and Associates, 1997, 1-156592-225-5.

Per poter dispiegare pienamente la potenza dello scripting di shell bisogna avere almeno una certa familiarità, seppur superficiale, con **sed** e **awk**. Questa è la guida standard. Comprende un'eccellente spiegazione delle "espressioni regolari". È un libro da leggere.

*

Jeffrey Friedl, *Mastering Regular Expressions*, O'Reilly and Associates, 2002, 0-596-00289-0.

La migliore e più completa guida di riferimento sulle Espressioni Regolari.

*

Aeleen Frisch, *Essential System Administration*, 3rd edition, O'Reilly and Associates, 2002, 0-596-00343-9.

Questo eccellente manuale per l'amministrazione di sistema contiene una parte, piuttosto buona, dedicata alle basi dello scripting di shell che interessano gli amministratori di sistema e svolge un buon lavoro nella spiegazione degli script di installazione e d'avvio. Dopo una lunga attesa, finalmente è stata pubblicata la terza edizione di questo classico.

*

Stephen Kochan e Patrick Woods, *Unix Shell Programming*, Hayden, 1990, 067248448X.

La guida di riferimento standard sebbene, attualmente, un po' datata.

*

Neil Matthew e Richard Stones, *Beginning Linux Programming*, Wrox Press, 1996, 1874416680.

Ottima e approfondita trattazione dei diversi linguaggi di programmazione per Linux, contenente un capitolo veramente consistente sullo scripting di shell.

*

Herbert Mayer, *Advanced C Programming on the IBM PC*, Windcrest Books, 1989, 0830693637.

Eccellente analisi di algoritmi e regole generali di programmazione.

*

David Medinets, *Unix Shell Programming Tools*, McGraw-Hill, 1999, 0070397333.

Ottime informazioni sullo scripting di shell, con esempi ed una breve introduzione a Tcl e Perl.

*

Cameron Newham e Bill Rosenblatt, *Learning the Bash Shell*, 2nd edition, O'Reilly and Associates, 1998, 1-56592-347-2.

Discreto manuale che rappresenta un valido sforzo per l'introduzione alla shell, ma difetta nell'esposizione di argomenti attinenti alla programmazione, nonché di un sufficiente numero di esempi.

*

Anatole Olczak, *Bourne Shell Quick Reference Guide*, ASP, Inc., 1991, 093573922X.

Utilissima guida di riferimento tascabile, sebbene tralasci di trattare le funzionalità specifiche di Bash.

*

Jerry Peek, Tim O'Reilly, e Mike Loukides, *Unix Power Tools*, 2nd edition, O'Reilly and Associates, Random House, 1997, 1-56592-260-3.

Contiene un paio di dettagliate sezioni, con articoli approfonditi, sulla programmazione di shell, ma insufficiente come manuale. Sulle espressioni regolari, riporta molto del succitato libro di Dougherty e Robbins.

*

Clifford Pickover, *Computers, Pattern, Chaos, and Beauty*, St. Martin's Press, 1990, 0-312-04123-3.

Un tesoro ritrovato di idee e formule per esplorare, con il computer, molte curiosità matematiche.

*

George Polya, *How To Solve It*, Princeton University Press, 1973, 0-691-02356-5.

Il classico manuale di metodi per la soluzione di problemi (leggi: algoritmi).

*

Chet Ramey e Brian Fox, *The GNU Bash Reference Manual*, Network Theory Ltd, 2003, 0-9541617-7-7.

Questo manuale è la guida di riferimento finale per Bash GNU. Gli autori, Chet Ramey e Brian Fox, sono gli sviluppatori di Bash GNU. Per ogni copia venduta l'editore devolve un dollaro alla Free Software Foundation.

Arnold Robbins, *Bash Reference Card*, SSC, 1998, 1-58731-010-5.

Un'eccellente guida di riferimento tascabile per Bash (da non dimenticare mai a casa). Un affare a $ 4.95, ma che è anche possibile scaricare on-line in formato pdf.

*

Arnold Robbins, *Effective Awk Programming*, Free Software Foundation / O'Reilly and Associates, 2000, 1-882114-26-4.

In assoluto il miglior manuale e guida di riferimento su **awk**. La versione elettronica, libera, di questo libro fa parte della documentazione di **awk**, mentre quella stampata è disponibile presso O'Reilly and Associates.

Questo libro è servito d'ispirazione all'autore del presente documento.

*

Bill Rosenblatt, *Learning the Korn Shell*, O'Reilly and Associates, 1993, 1-56592-054-6.

Questo libro, ben scritto, contiene ottimi suggerimenti sullo scripting di shell.

*

Paul Sheer, *LINUX: Rute User's Tutorial and Exposition*, 1st edition, , 2002, 0-13-033351-4.

Testo introduttivo molto dettagliato e di piacevole lettura sull'amministrazione del sistema Linux.

Il libro è disponibile in forma stampata o on-line.

*

Ellen Siever e lo staff della O'Reilly and Associates, *Linux in a Nutshell*, 2nd edition, O'Reilly and Associates, 1999, 1-56592-585-8.

La migliore, e più completa, guida di riferimento ai comandi Linux, con una sezione dedicata a Bash.

*

The UNIX CD Bookshelf, 3rd edition, O'Reilly and Associates, 2003, 0-596-00392-7.

Raccolta, su CD ROM, di sette libri su UNIX, tra cui *UNIX Power Tools*, *Sed and Awk* e *Learning the Korn Shell*. Una serie completa di tutti i manuali e guide di riferimento UNIX, di cui dovreste aver bisogno, a circa 130 dollari. Acquistatela, anche se questo significa far debiti e non pagare l'affitto.

*

I libri O'Reilly su Perl (attualmente, tutti i libri O'Reilly).

Fioretti Marco, "Scripting for X Productivity," LINUX JOURNAL, numero 113, settembre, 2003, pp. 86-9.

I begli articoli di Ben Okopnik *introductory Bash scripting* nei numeri 53, 54, 55, 57 e 59 di Linux Gazette e la sua spiegazione su "The Deep, Dark Secrets of Bash" nel numero 56.

Chet Ramey's *bash - The GNU Shell*, serie in due parti pubblicata nei numeri 3 e 4 di Linux Journal, Luglio-Agosto 1994.

Bash-Programming-Intro HOWTO di Mike G.

Unix Scripting Universe di Richard.

Bash F.A.Q. di Chet Ramey.

Shell Corner di Ed Schaefer in Unix Review.

Gli script di shell d'esempio presso Lucc's Shell Scripts .

Gli script di shell d'esempio presso SHELLdorado.

Gli script di shell d'esempio al sito di Noah Friedman.

Shell Programming Stuff di Steve Parker.

Gli script di shell d'esempio presso SourceForge Snippet Library - shell scrips.

Bash-Prompt HOWTO di Giles Orr.

I bellissimi manuali su **sed**, **awk** e le espressioni regolari presso The UNIX Grymoire.

La Sed resources page, di Eric Pement.

Il manuale di riferimento di **gawk** GNU (**gawk** è la versione GNU estesa di **awk** disponibile sui sistemi Linux e BSD).

Il Groff tutorial, di Trent Fisher.

Printing-Usage HOWTO di Mark Komarinski.

The Linux USB subsystem (utile per gli script che devono occuparsi delle periferiche USB).

Vi è dell'ottimo materiale sulla redirezione I/O nel capitolo 10 della documentazione di textutils sul sito della University of Alberta.

Rick Hohensee ha scritto osimpa, assembler i386 implementato interamente con script Bash.

Aurelio Marinho Jargas ha scritto Regular expression wizard. Ha inoltre scritto un libro molto istruttivo sulle Espressioni Regolari, in portoghese.

Ben Tomkins ha creato Bash Navigator, strumento di gestione delle directory.

William Park sta lavorando ad un progetto per inserire in Bash alcune funzionalità di Awk e Python. Tra queste un'interfaccia *gdbm*. Ha rilasciato bashdiff su Freshmeat.net. Suo l'articolo, nel numero del novembre 2004 della Linux Gazette, che tratta della'aggiunta di funzioni stringa in Bash.

Rocky Bernstein sta procedendo nello sviluppo di un "maturo e collaudato" debugger per Bash.

L'eccellente "Bash Reference Manual" di Chet Ramey e Brian Fox, distribuito come parte del pacchetto "bash-2-doc"(disponibile in formato rpm). Si vedano in particolare gli istruttivi script d'esempio presenti nel pacchetto.

Il newsgroup comp.os.unix.shell.

comp.os.unix.shell FAQ e il suo mirror.

Diverse comp.os.unix FAQ.

Le pagine di manuale di **bash** e **bash2**, **date**, **expect**, **expr**, **find**, **grep**, **gzip**, **ln**, **patch**, **tar**, **tr**, **bc**, **xargs**. La documentazione texinfo di **bash**, **dd**, **m4**, **gawk** e **sed**.

Appendice A. Script aggiuntivi

Questi script, sebbene non siano stati inseriti nel testo del documento, illustrano alcune interessanti tecniche di programmazione di shell. Sono anche utili. Ci si diverta ad analizzarli e a eseguirli.

Esempio A-1. manview: visualizzare eleganti pagine di manuale

```
 1 #!/bin/bash
 2 # manview.sh: Formats the source of a man page for viewing.
 3
 4 #  This script is useful when writing man page source.
 5 #  It lets you look at the intermediate results on the fly
 6 #+ while working on it.
 7
 8 E_WRONGARGS=65
 9
10 if [ -z "$1" ]
11 then
12   echo "Usage: `basename $0` filename"
13   exit $E_WRONGARGS
14 fi
15
16 # ---------------------------
17 groff -Tascii -man $1 | less
18 # From the man page for groff.
19 # ---------------------------
20
21 #  If the man page includes tables and/or equations,
22 #+ then the above code will barf.
23 #  The following line can handle such cases.
24 #
25 #   gtbl < "$1" | geqn -Tlatin1 | groff -Tlatin1 -mtty-char -man
26 #
27 #   Thanks, S.C.
28
29 exit 0
```

Esempio A-2. mailformat: impaginare un messaggio e-mail

```
 1 #!/bin/bash
 2 # mail-format.sh: Format e-mail messages.
 3
```

```
 4 # Gets rid of carets, tabs, also fold excessively long lines.
 5
 6 # ================================================================
 7 #                 Standard Check for Script Argument(s)
 8 ARGS=1
 9 E_BADARGS=65
10 E_NOFILE=66
11
12 if [ $# -ne $ARGS ]  # Correct number of arguments passed to script?
13 then
14   echo "Usage: `basename $0` filename"
15   exit $E_BADARGS
16 fi
17
18 if [ -f "$1" ]       # Check if file exists.
19 then
20     file_name=$1
21 else
22     echo "File \"$1\" does not exist."
23     exit $E_NOFILE
24 fi
25 # ================================================================
26
27 MAXWIDTH=70          # Width to fold long lines to.
28
29 #  Delete carets and tabs at beginning of lines,
30 #+ then fold lines to $MAXWIDTH characters.
31 sed '
32 s/^>//
33 s/^  *>//
34 s/^  *//
35 s/          *//
36 ' $1 | fold -s --width=$MAXWIDTH
37           # -s option to "fold" breaks lines at whitespace, if possible.
38
39 #  This script was inspired by an article in a well-known trade journal
40 #+ extolling a 164K Windows utility with similar functionality.
41 #
42 #  An nice set of text processing utilities and an efficient
43 #+ scripting language provide an alternative to bloated executables.
44
45 exit 0
```

Esempio A-3. rn: una semplice utility per rinominare un file

Questo script è una modifica di Esempio 12-18.

```
 1 #! /bin/bash
 2 #
 3 # Very simpleminded filename "rename" utility (based on "lowercase.sh").
 4 #
 5 #  The "ren" utility, by Vladimir Lanin (lanin@csd2.nyu.edu),
 6 #+ does a much better job of this.
 7
 8
 9 ARGS=2
10 E_BADARGS=65
11 ONE=1                      # For getting singular/plural right (see below).
12
13 if [ $# -ne "$ARGS" ]
14 then
```

```
 15   echo "Usage: `basename $0` old-pattern new-pattern"
 16   # As in "rn gif jpg", which renames all gif files in working directory to
jpg.
 17   exit $E_BADARGS
 18 fi
 19
 20 number=0                 # Keeps track of how many files actually renamed.
 21
 22
 23 for filename in *$1*      #Traverse all matching files in directory.
 24 do
 25    if [ -f "$filename" ]  # If finds match...
 26    then
 27      fname=`basename $filename`            # Strip off path.
 28      n=`echo $fname | sed -e "s/$1/$2/"`   # Substitute new for old in
filename.
 29      mv $fname $n                          # Rename.
 30      let "number += 1"
 31    fi
 32 done
 33
 34 if [ "$number" -eq "$ONE" ]               # For correct grammar.
 35 then
 36  echo "$number file renamed."
 37 else
 38  echo "$number files renamed."
 39 fi
 40
 41 exit 0
 42
 43
 44 # Exercises:
 45 # ---------
 46 # What type of files will this not work on?
 47 # How can this be fixed?
 48 #
 49 #  Rewrite this script to process all the files in a directory
 50 #+ containing spaces in their names, and to rename them,
 51 #+ substituting an underscore for each space.
```

Esempio A-4. blank-rename: rinomina file i cui nomi contengono spazi

È una versione ancor più semplice dello script precedente

```
  1 #! /bin/bash
  2 # blank-rename.sh
  3 #
  4 # Substitutes underscores for blanks in all the filenames in a directory.
  5
  6 ONE=1                    # For getting singular/plural right (see below).
  7 number=0                 # Keeps track of how many files actually renamed.
  8 FOUND=0                  # Successful return value.
  9
 10 for filename in *         #Traverse all files in directory.
 11 do
 12      echo "$filename" | grep -q " "         #  Check whether filename
 13      if [ $? -eq $FOUND ]                   #+ contains space(s).
 14      then
 15        fname=$filename                      # Strip off path.
 16        n=`echo $fname | sed -e "s/ /_/g"`   # Substitute underscore for
blank.
```

```
17        mv "$fname" "$n"                    # Do the actual renaming.
18        let "number += 1"
19      fi
20 done
21
22 if [ "$number" -eq "$ONE" ]                # For correct grammar.
23 then
24  echo "$number file renamed."
25 else
26  echo "$number files renamed."
27 fi
28
29 exit 0
```

Esempio A-5. encryptedpw: upload a un sito ftp utilizzando una password criptata localmente

```
 1 #!/bin/bash
 2
 3 # Example "ex72.sh" modified to use encrypted password.
 4
 5 #  Note that this is still rather insecure,
 6 #+ since the decrypted password is sent in the clear.
 7 #  Use something like "ssh" if this is a concern.
 8
 9 E_BADARGS=65
10
11 if [ -z "$1" ]
12 then
13   echo "Usage: `basename $0` filename"
14   exit $E_BADARGS
15 fi
16
17 Username=bozo           # Change to suit.
18 pword=/home/bozo/secret/password_encrypted.file
19 # File containing encrypted password.
20
21 Filename=`basename $1`  # Strips pathname out of file name
22
23 Server="XXX"
24 Directory="YYY"         # Change above to actual server name & directory.
25
26
27 Password=`cruft <$pword`          # Decrypt password.
28 #  Uses the author's own "cruft" file encryption package,
29 #+ based on the classic "onetime pad" algorithm,
30 #+ and obtainable from:
31 #+ Primary-site:   ftp://ibiblio.org/pub/Linux/utils/file
32 #+                 cruft-0.2.tar.gz [16k]
33
34
35 ftp -n $Server <<End-Of-Session
36 user $Username $Password
37 binary
38 bell
39 cd $Directory
40 put $Filename
41 bye
42 End-Of-Session
43 # -n option to "ftp" disables auto-logon.
44 # Note that "bell" rings 'bell' after each file transfer.
45
```

```
 46 exit 0
```

Esempio A-6. copy-cd: copiare un CD di dati

```
  1 #!/bin/bash
  2 # copy-cd.sh: copying a data CD
  3
  4 CDROM=/dev/cdrom                            # CD ROM device
  5 OF=/home/bozo/projects/cdimage.iso          # output file
  6 #       /xxxx/xxxxxxx/                      Change to suit your system.
  7 BLOCKSIZE=2048
  8 SPEED=2                                     # May use higher speed if
supported.
  9 DEVICE=cdrom
 10 # DEVICE="0,0"    on older versions of cdrecord.
 11
 12 echo; echo "Insert source CD, but do *not* mount it."
 13 echo "Press ENTER when ready. "
 14 read ready                                  # Wait for input, $ready not
used.
 15
 16 echo; echo "Copying the source CD to $OF."
 17 echo "This may take a while. Please be patient."
 18
 19 dd if=$CDROM of=$OF bs=$BLOCKSIZE           # Raw device copy.
 20
 21
 22 echo; echo "Remove data CD."
 23 echo "Insert blank CDR."
 24 echo "Press ENTER when ready. "
 25 read ready                                  # Wait for input, $ready not
used.
 26
 27 echo "Copying $OF to CDR."
 28
 29 cdrecord -v -isosize speed=$SPEED dev=$DEVICE $OF
 30 # Uses Joerg Schilling's "cdrecord" package (see its docs).
 31 # http://www.fokus.gmd.de/nthp/employees/schilling/cdrecord.html
 32
 33
 34 echo; echo "Done copying $OF to CDR on device $CDROM."
 35
 36 echo "Do you want to erase the image file (y/n)? "  # Probably a huge file.
 37 read answer
 38
 39 case "$answer" in
 40 [yY]) rm -f $OF
 41       echo "$OF erased."
 42       ;;
 43 *)    echo "$OF not erased.";;
 44 esac
 45
 46 echo
 47
 48 # Exercise:
 49 # Change the above "case" statement to also accept "yes" and "Yes" as input.
 50
 51 exit 0
```

Esempio A-7. Serie di Collatz

```
 1 #!/bin/bash
 2 # collatz.sh
 3
 4 #  The notorious "hailstone" or Collatz series.
 5 #  ----------------------------------------
 6 #  1) Get the integer "seed" from the command line.
 7 #  2) NUMBER <--- seed
 8 #  3) Print NUMBER.
 9 #  4)  If NUMBER is even, divide by 2, or
10 #  5)+ if odd, multiply by 3 and add 1.
11 #  6) NUMBER <--- result
12 #  7) Loop back to step 3 (for specified number of iterations).
13 #
14 #  The theory is that every sequence,
15 #+ no matter how large the initial value,
16 #+ eventually settles down to repeating "4,2,1..." cycles,
17 #+ even after fluctuating through a wide range of values.
18 #
19 #  This is an instance of an "iterate",
20 #+ an operation that feeds its output back into the input.
21 #  Sometimes the result is a "chaotic" series.
22
23
24 MAX_ITERATIONS=200
25 # For large seed numbers (>32000), increase MAX_ITERATIONS.
26
27 h=${1:-$$}                      #  Seed
28                                 #  Use $PID as seed,
29                                 #+ if not specified as command-line arg.
30
31 echo
32 echo "C($h) --- $MAX_ITERATIONS Iterations"
33 echo
34
35 for ((i=1; i<=MAX_ITERATIONS; i++))
36 do
37
38 echo -n "$h"
39 #          ^^^^^
40 #           tab
41
42   let "remainder = h % 2"
43   if [ "$remainder" -eq 0 ]   # Even?
44   then
45     let "h /= 2"              # Divide by 2.
46   else
47     let "h = h*3 + 1"         # Multiply by 3 and add 1.
48   fi
49
50
51 COLUMNS=10                    # Output 10 values per line.
52 let "line_break = i % $COLUMNS"
53 if [ "$line_break" -eq 0 ]
54 then
55   echo
56 fi
57
58 done
59
60 echo
61
62 #  For more information on this mathematical function,
```

```
 63 #+ see "Computers, Pattern, Chaos, and Beauty", by Pickover, p. 185 ff.,
 64 #+ as listed in the bibliography.
 65
 66 exit 0
```

Esempio A-8. days-between: calcolo del numero di giorni intercorrenti tra due date

```
  1 #!/bin/bash
  2 # days-between.sh:    Number of days between two dates.
  3 # Usage: ./days-between.sh [M]M/[D]D/YYYY [M]M/[D]D/YYYY
  4 #
  5 # Note: Script modified to account for changes in Bash 2.05b
  6 #+      that closed the loophole permitting large negative
  7 #+      integer return values.
  8
  9 ARGS=2                # Two command line parameters expected.
 10 E_PARAM_ERR=65        # Param error.
 11
 12 REFYR=1600            # Reference year.
 13 CENTURY=100
 14 DIY=365
 15 ADJ_DIY=367           # Adjusted for leap year + fraction.
 16 MIY=12
 17 DIM=31
 18 LEAPCYCLE=4
 19
 20 MAXRETVAL=255         #  Largest permissable
 21                       #+ positive return value from a function.
 22
 23 diff=                 # Declare global variable for date difference.
 24 value=                # Declare global variable for absolute value.
 25 day=                  # Declare globals for day, month, year.
 26 month=
 27 year=
 28
 29
 30 Param_Error ()        # Command line parameters wrong.
 31 {
 32   echo "Usage: `basename $0` [M]M/[D]D/YYYY [M]M/[D]D/YYYY"
 33   echo "       (date must be after 1/3/1600)"
 34   exit $E_PARAM_ERR
 35 }
 36
 37
 38 Parse_Date ()                # Parse date from command line params.
 39 {
 40   month=${1%%/**}
 41   dm=${1%/**}                # Day and month.
 42   day=${dm#*/}
 43   let "year = `basename $1`"  # Not a filename, but works just the same.
 44 }
 45
 46
 47 check_date ()                # Checks for invalid date(s) passed.
 48 {
 49   [ "$day" -gt "$DIM" ] || [ "$month" -gt "$MIY" ] || [ "$year" -lt "$REFYR"
] && Param_Error
 50   # Exit script on bad value(s).
 51   # Uses "or-list / and-list".
 52   #
 53   # Exercise: Implement more rigorous date checking.
```

```
 54 }
 55
 56
 57 strip_leading_zero () #  Better to strip possible leading zero(s)
 58 {                     #+ from day and/or month
 59   return ${1#0}       #+ since otherwise Bash will interpret them
 60 }                     #+ as octal values (POSIX.2, sect 2.9.2.1).
 61
 62
 63 day_index ()          # Gauss' Formula:
 64 {                     # Days from Jan. 3, 1600 to date passed as param.
 65
 66   day=$1
 67   month=$2
 68   year=$3
 69
 70   let "month = $month - 2"
 71   if [ "$month" -le 0 ]
 72   then
 73     let "month += 12"
 74     let "year -= 1"
 75   fi
 76
 77   let "year -= $REFYR"
 78   let "indexyr = $year / $CENTURY"
 79
 80
 81   let "Days = $DIY*$year + $year/$LEAPCYCLE - $indexyr + $indexyr/$LEAPCYCLE
+ $ADJ_DIY*$month/$MIY + $day - $DIM"
 82   #  For an in-depth explanation of this algorithm, see
 83   #+ http://home.t-online.de/home/berndt.schwerdtfeger/cal.htm
 84
 85
 86   echo $Days
 87
 88 }
 89
 90
 91 calculate_difference ()           # Difference between to day indices.
 92 {
 93   let "diff = $1 - $2"            # Global variable.
 94 }
 95
 96
 97 abs ()                            #  Absolute value
 98 {                                 #  Uses global "value" variable.
 99   if [ "$1" -lt 0 ]               #  If negative
100   then                            #+ then
101     let "value = 0 - $1"          #+ change sign,
102   else                            #+ else
103     let "value = $1"              #+ leave it alone.
104   fi
105 }
106
107
108
109 if [ $# -ne "$ARGS" ]             # Require two command line params.
110 then
111   Param_Error
112 fi
113
114 Parse_Date $1
```

```
115 check_date $day $month $year       #  See if valid date.
116
117 strip_leading_zero $day            #  Remove any leading zeroes
118 day=$?                            #+ on day and/or month.
119 strip_leading_zero $month
120 month=$?
121
122 let "date1 = `day_index $day $month $year`"
123
124
125 Parse_Date $2
126 check_date $day $month $year
127
128 strip_leading_zero $day
129 day=$?
130 strip_leading_zero $month
131 month=$?
132
133 date2=$(day_index $day $month $year) # Command substitution.
134
135
136 calculate_difference $date1 $date2
137
138 abs $diff                           # Make sure it's positive.
139 diff=$value
140
141 echo $diff
142
143 exit 0
144 #  Compare this script with
145 #+ the implementation of Gauss' Formula in a C program at:
146 #+    http://buschencrew.hypermart.net/software/datedif
```

Esempio A-9. Creare un "dizionario"

```
  1 #!/bin/bash
  2 # makedict.sh  [make dictionary]
  3
  4 # Modification of /usr/sbin/mkdict script.
  5 # Original script copyright 1993, by Alec Muffett.
  6 #
  7 #  This modified script included in this document in a manner
  8 #+ consistent with the "LICENSE" document of the "Crack" package
  9 #+ that the original script is a part of.
 10
 11 #  This script processes text files to produce a sorted list
 12 #+ of words found in the files.
 13 #  This may be useful for compiling dictionaries
 14 #+ and for lexicographic research.
 15
 16
 17 E_BADARGS=65
 18
 19 if [ ! -r "$1" ]                   #  Need at least one
 20 then                               #+ valid file argument.
 21   echo "Usage: $0 files-to-process"
 22   exit $E_BADARGS
 23 fi
 24
 25
 26 # SORT="sort"                       #  No longer necessary to define
```

```
options
 27                                    #+ to sort. Changed from original
script.
 28
 29 cat $* |                           # Contents of specified files to
stdout.
 30         tr A-Z a-z |               # Convert to lowercase.
 31         tr ' ' '\012' |            # New: change spaces to newlines.
 32 #       tr -cd '\012[a-z][0-9]' |  #  Get rid of everything non-
alphanumeric
 33                                    #+ (original script).
 34         tr -c '\012a-z'  '\012' |  #  Rather than deleting
 35                                    #+ now change non-alpha to newlines.
 36         sort |                     # $SORT options unnecessary now.
 37         uniq |                     # Remove duplicates.
 38         grep -v '^#' |             # Delete lines beginning with a
hashmark.
 39         grep -v '^$'               # Delete blank lines.
 40
 41 exit 0
```

Esempio A-10. Conversione soundex

```
  1 #!/bin/bash
  2 # soundex.sh: Calculate "soundex" code for names
  3
  4 # ======================================================
  5 #        Soundex script
  6 #              by
  7 #         Mendel Cooper
  8 #     thegrendel@theriver.com
  9 #       23 January, 2002
 10 #
 11 #   Placed in the Public Domain.
 12 #
 13 # A slightly different version of this script appeared in
 14 #+ Ed Schaefer's July, 2002 "Shell Corner" column
 15 #+ in "Unix Review" on-line,
 16 #+ http://www.unixreview.com/documents/uni1026336632258/
 17 # ======================================================
 18
 19
 20 ARGCOUNT=1                      # Need name as argument.
 21 E_WRONGARGS=70
 22
 23 if [ $# -ne "$ARGCOUNT" ]
 24 then
 25   echo "Usage: `basename $0` name"
 26   exit $E_WRONGARGS
 27 fi
 28
 29
 30 assign_value ()                 #  Assigns numerical value
 31 {                               #+ to letters of name.
 32
 33   val1=bfpv                     # 'b,f,p,v' = 1
 34   val2=cgjkqsxz                 # 'c,g,j,k,q,s,x,z' = 2
 35   val3=dt                       #  etc.
 36   val4=l
 37   val5=mn
 38   val6=r
```

```
 39
 40 # Exceptionally clever use of 'tr' follows.
 41 # Try to figure out what is going on here.
 42
 43 value=$( echo "$1" \
 44 | tr -d wh \
 45 | tr $val1 1 | tr $val2 2 | tr $val3 3 \
 46 | tr $val4 4 | tr $val5 5 | tr $val6 6 \
 47 | tr -s 123456 \
 48 | tr -d aeiouy )
 49
 50 # Assign letter values.
 51 # Remove duplicate numbers, except when separated by vowels.
 52 # Ignore vowels, except as separators, so delete them last.
 53 # Ignore 'w' and 'h', even as separators, so delete them first.
 54 #
 55 # The above command substitution lays more pipe than a plumber <g>.
 56
 57 }
 58
 59
 60 input_name="$1"
 61 echo
 62 echo "Name = $input_name"
 63
 64
 65 # Change all characters of name input to lowercase.
 66 # ------------------------------------------------
 67 name=$( echo $input_name | tr A-Z a-z )
 68 # ------------------------------------------------
 69 # Just in case argument to script is mixed case.
 70
 71
 72 # Prefix of soundex code: first letter of name.
 73 # ----------------------------------------
 74
 75
 76 char_pos=0                     # Initialize character position.
 77 prefix0=${name:$char_pos:1}
 78 prefix=`echo $prefix0 | tr a-z A-Z`
 79                                # Uppercase 1st letter of soundex.
 80
 81 let "char_pos += 1"            # Bump character position to 2nd letter of
name.
 82 name1=${name:$char_pos}
 83
 84
 85 # ++++++++++++++++++++++++++ Exception Patch
++++++++++++++++++++++++++++++++
 86 #  Now, we run both the input name and the name shifted one char to the
right
 87 #+ through the value-assigning function.
 88 #  If we get the same value out, that means that the first two characters
 89 #+ of the name have the same value assigned, and that one should cancel.
 90 #  However, we also need to test whether the first letter of the name
 91 #+ is a vowel or 'w' or 'h', because otherwise this would bollix things up.
 92
 93 char1=`echo $prefix | tr A-Z a-z`    # First letter of name, lowercased.
 94
 95 assign_value $name
 96 s1=$value
 97 assign_value $name1
```

```
 98 s2=$value
 99 assign_value $char1
100 s3=$value
101 s3=9$s3                               #  If first letter of name is a vowel
102                                       #+ or 'w' or 'h',
103                                       #+ then its "value" will be null
(unset).
104                                  #+ Therefore, set it to 9, an otherwise
105                                  #+ unused value, which can be tested for.
106
107
108 if [[ "$s1" -ne "$s2" || "$s3" -eq 9 ]]
109 then
110   suffix=$s2
111 else
112   suffix=${s2:$char_pos}
113 fi
114 # ++++++++++++++++++++++ end Exception Patch
++++++++++++++++++++++++++++++++
115
116
117 padding=000                     # Use at most 3 zeroes to pad.
118
119
120 soun=$prefix$suffix$padding     # Pad with zeroes.
121
122 MAXLEN=4                        # Truncate to maximum of 4 chars.
123 soundex=${soun:0:$MAXLEN}
124
125 echo "Soundex = $soundex"
126
127 echo
128
129 #  The soundex code is a method of indexing and classifying names
130 #+ by grouping together the ones that sound alike.
131 #  The soundex code for a given name is the first letter of the name,
132 #+ followed by a calculated three-number code.
133 #  Similar sounding names should have almost the same soundex codes.
134
135 #   Examples:
136 #   Smith and Smythe both have a "S-530" soundex.
137 #   Harrison = H-625
138 #   Hargison = H-622
139 #   Harriman = H-655
140
141 #  This works out fairly well in practice, but there are numerous anomalies.
142 #
143 #
144 #  The U.S. Census and certain other governmental agencies use soundex,
145 #  as do genealogical researchers.
146 #
147 #  For more information,
148 #+ see the "National Archives and Records Administration home page",
149 #+ http://www.nara.gov/genealogy/soundex/soundex.html
150
151
152
153 # Exercise:
154 # --------
155 # Simplify the "Exception Patch" section of this script.
156
157 exit 0
```

Esempio A-11. "Game of Life"

```
#!/bin/bash
# life.sh: "Life in the Slow Lane"
# Version 2: Patched by Daniel Albers
#+           to allow non-square grids as input.

# ########################################################################## #
# This is the Bash script version of John Conway's "Game of Life".      #
# "Life" is a simple implementation of cellular automata.               #
# --------------------------------------------------------------------- #
# On a rectangular grid, let each "cell" be either "living" or "dead".  #
# Designate a living cell with a dot, and a dead one with a blank space.#
#  Begin with an arbitrarily drawn dot-and-blank grid,                  #
#+ and let this be the starting generation, "generation 0".            #
# Determine each successive generation by the following rules:          #
# 1) Each cell has 8 neighbors, the adjoining cells                     #
#+   left, right, top, bottom, and the 4 diagonals.                     #
#                       123                                             #
#                       4*5                                             #
#                       678                                             #
#                                                                       #
# 2) A living cell with either 2 or 3 living neighbors remains alive.   #
# 3) A dead cell with 3 living neighbors becomes alive (a "birth").     #
SURVIVE=2                                                               #
BIRTH=3                                                                 #
# 4) All other cases result in a dead cell for the next generation.     #
# ########################################################################## #


startfile=gen0   # Read the starting generation from the file "gen0".
                 # Default, if no other file specified when invoking script.
                 #
if [ -n "$1" ]   # Specify another "generation 0" file.
then
  if [ -e "$1" ] # Check for existence.
  then
    startfile="$1"
  fi
fi


ALIVE1=.
DEAD1=_
                 # Represent living and "dead" cells in the start-up file.

#  ------------------------------------------------------- #
#  This script uses a 10 x 10 grid (may be increased,
#+ but a large grid will will cause very slow execution).
ROWS=10
COLS=10
#  Change above two variables to match grid size, if necessary.
#  ------------------------------------------------------- #

GENERATIONS=10          #  How many generations to cycle through.
                        #  Adjust this upwards,
                        #+ if you have time on your hands.

NONE_ALIVE=80           #  Exit status on premature bailout,
                        #+ if no cells left alive.
TRUE=0
FALSE=1
```

```
 61 ALIVE=0
 62 DEAD=1
 63
 64 avar=                   #  Global; holds current generation.
 65 generation=0            # Initialize generation count.
 66
 67 # ==================================================================
 68
 69
 70 let "cells = $ROWS * $COLS"
 71                          # How many cells.
 72
 73 declare -a initial      # Arrays containing "cells".
 74 declare -a current
 75
 76 display ()
 77 {
 78
 79 alive=0                 # How many cells "alive" at any given time.
 80                          # Initially zero.
 81
 82 declare -a arr
 83 arr=( `echo "$1"` )     # Convert passed arg to array.
 84
 85 element_count=${#arr[*]}
 86
 87 local i
 88 local rowcheck
 89
 90 for ((i=0; i<$element_count; i++))
 91 do
 92
 93   # Insert newline at end of each row.
 94   let "rowcheck = $i % COLS"
 95   if [ "$rowcheck" -eq 0 ]
 96   then
 97     echo                 # Newline.
 98     echo -n "      "     # Indent.
 99   fi
100
101   cell=${arr[i]}
102
103   if [ "$cell" = . ]
104   then
105     let "alive += 1"
106   fi
107
108   echo -n "$cell" | sed -e 's/_/ /g'
109   # Print out array and change underscores to spaces.
110 done
111
112 return
113
114 }
115
116 IsValid ()                           # Test whether cell coordinate valid.
117 {
118
119   if [ -z "$1"  -o -z "$2" ]         # Mandatory arguments missing?
120   then
121     return $FALSE
122   fi
```

```
123
124 local row
125 local lower_limit=0                   # Disallow negative coordinate.
126 local upper_limit
127 local left
128 local right
129
130 let "upper_limit = $ROWS * $COLS - 1" # Total number of cells.
131
132
133 if [ "$1" -lt "$lower_limit" -o "$1" -gt "$upper_limit" ]
134 then
135   return $FALSE                       # Out of array bounds.
136 fi
137
138 row=$2
139 let "left = $row * $COLS"             # Left limit.
140 let "right = $left + $COLS - 1"       # Right limit.
141
142 if [ "$1" -lt "$left" -o "$1" -gt "$right" ]
143 then
144   return $FALSE                       # Beyond row boundary.
145 fi
146
147 return $TRUE                          # Valid coordinate.
148
149 }
150
151
152 IsAlive ()              # Test whether cell is alive.
153                         # Takes array, cell number, state of cell as
arguments.
154 {
155   GetCount "$1" $2      # Get alive cell count in neighborhood.
156   local nhbd=$?
157
158
159   if [ "$nhbd" -eq "$BIRTH" ]  # Alive in any case.
160   then
161     return $ALIVE
162   fi
163
164   if [ "$3" = "." -a "$nhbd" -eq "$SURVIVE" ]
165   then                  # Alive only if previously alive.
166     return $ALIVE
167   fi
168
169   return $DEAD          # Default.
170
171 }
172
173
174 GetCount ()             # Count live cells in passed cell's neighborhood.
175                         # Two arguments needed:
176                   # $1) variable holding array
177                   # $2) cell number
178 {
179   local cell_number=$2
180   local array
181   local top
182   local center
183   local bottom
```

```
184   local r
185   local row
186   local i
187   local t_top
188   local t_cen
189   local t_bot
190   local count=0
191   local ROW_NHBD=3
192
193   array=( `echo "$1"` )
194
195   let "top = $cell_number - $COLS - 1"    # Set up cell neighborhood.
196   let "center = $cell_number - 1"
197   let "bottom = $cell_number + $COLS - 1"
198   let "r = $cell_number / $COLS"
199
200   for ((i=0; i<$ROW_NHBD; i++))           # Traverse from left to right.
201   do
202     let "t_top = $top + $i"
203     let "t_cen = $center + $i"
204     let "t_bot = $bottom + $i"
205
206
207     let "row = $r"                        # Count center row of
neighborhood.
208     IsValid $t_cen $row                   # Valid cell position?
209     if [ $? -eq "$TRUE" ]
210     then
211       if [ ${array[$t_cen]} = "$ALIVE1" ] # Is it alive?
212       then                                # Yes?
213         let "count += 1"                  # Increment count.
214       fi
215     fi
216
217     let "row = $r - 1"                    # Count top row.
218     IsValid $t_top $row
219     if [ $? -eq "$TRUE" ]
220     then
221       if [ ${array[$t_top]} = "$ALIVE1" ]
222       then
223         let "count += 1"
224       fi
225     fi
226
227     let "row = $r + 1"                    # Count bottom row.
228     IsValid $t_bot $row
229     if [ $? -eq "$TRUE" ]
230     then
231       if [ ${array[$t_bot]} = "$ALIVE1" ]
232       then
233         let "count += 1"
234       fi
235     fi
236
237   done
238
239
240   if [ ${array[$cell_number]} = "$ALIVE1" ]
241   then
242     let "count -= 1"        #  Make sure value of tested cell itself
243   fi                        #+ is not counted.
244
```

```
245
246   return $count
247
248 }
249
250 next_gen ()               # Update generation array.
251 {
252
253 local array
254 local i=0
255
256 array=( `echo "$1"` )     # Convert passed arg to array.
257
258 while [ "$i" -lt "$cells" ]
259 do
260   IsAlive "$1" $i ${array[$i]}   # Is cell alive?
261   if [ $? -eq "$ALIVE" ]
262   then                           #  If alive, then
263     array[$i]=.                  #+ represent the cell as a period.
264   else
265     array[$i]="_"                #  Otherwise underscore
266    fi                            #+ (which will later be converted to
space).
267   let "i += 1"
268 done
269
270
271 # let "generation += 1"   # Increment generation count.
272 # Why was the above line commented out?
273
274
275 # Set variable to pass as parameter to "display" function.
276 avar=`echo ${array[@]}`   # Convert array back to string variable.
277 display "$avar"           # Display it.
278 echo; echo
279 echo "Generation $generation -- $alive alive"
280
281 if [ "$alive" -eq 0 ]
282 then
283   echo
284   echo "Premature exit: no more cells alive!"
285   exit $NONE_ALIVE        #  No point in continuing
286 fi                        #+ if no live cells.
287
288 }
289
290
291 # ==============================================================
292
293 # main ()
294
295 # Load initial array with contents of startup file.
296 initial=( `cat "$startfile" | sed -e '/#/d' | tr -d '\n' |\
297 sed -e 's/\./\. /g' -e 's/_/_ /g'` )
298 # Delete lines containing '#' comment character.
299 # Remove linefeeds and insert space between elements.
300
301 clear          # Clear screen.
302
303 echo #         Title
304 echo "======================"
305 echo "    $GENERATIONS generations"
```

```
306 echo "           of"
307 echo "\"Life in the Slow Lane\""
308 echo "======================"
309
310
311 # -------- Display first generation. --------
312 Gen0=`echo ${initial[@]}`
313 display "$Gen0"           # Display only.
314 echo; echo
315 echo "Generation $generation -- $alive alive"
316 # ---------------------------------------
317
318
319 let "generation += 1"     # Increment generation count.
320 echo
321
322 # ------- Display second generation. -------
323 Cur=`echo ${initial[@]}`
324 next_gen "$Cur"          # Update & display.
325 # ---------------------------------------
326
327 let "generation += 1"     # Increment generation count.
328
329 # ------ Main loop for displaying subsequent generations ------
330 while [ "$generation" -le "$GENERATIONS" ]
331 do
332   Cur="$avar"
333   next_gen "$Cur"
334   let "generation += 1"
335 done
336 # ==============================================================
337
338 echo
339
340 exit 0
341
342 # --------------------------------------------------------------
343
344 # The grid in this script has a "boundary problem."
345 # The the top, bottom, and sides border on a void of dead cells.
346 # Exercise: Change the script to have the grid wrap around,
347 # +         so that the left and right sides will "touch,"
348 # +         as will the top and bottom.
349 #
350 # Exercise: Create a new "gen0" file to seed this script.
351 #           Use a 12 x 16 grid, instead of the original 10 x 10 one.
352 #           Make the necessary changes to the script,
353 #+          so it will run with the altered file.
354 #
355 # Exercise: Modify this script so that it can determine the grid size
356 #+          from the "gen0" file, and set any variables necessary
357 #+          for the script to run.
358 #           This would make unnecessary any changes to variables
359 #+          in the script for an altered grid size.
```

Esempio A-12. File dati per "Game of Life"

```
  1 # This is an example "generation 0" start-up file for "life.sh".
  2 # ---------------------------------------------------------
  3 #  The "gen0" file is a 10 x 10 grid using a period (.) for live cells,
  4 #+ and an underscore (_) for dead ones. We cannot simply use spaces
```

```
  5 #+ for dead cells in this file because of a peculiarity in Bash arrays.
  6 #  [Exercise for the reader: explain this.]
  7 #
  8 # Lines beginning with a '#' are comments, and the script ignores them.
  9 __.__..___
 10 ___._._____
 11 _____.___..
 12 _._______.
 13 _____._____
 14 ..__...___
 15 _____._____
 16 ___...____
 17 __..._..___
 18 _..___..__
```

+++

I due script seguenti sono di Mark Moraes della University of Toronto. Si veda l'allegato file "Moraes-COPYRIGHT" per quanto riguarda i permessi e le restrizioni.

Esempio A-13. behead: togliere le intestazioni dai messaggi di e-mail e di news

```
  1 #! /bin/sh
  2 # Strips off the header from a mail/News message i.e. till the first
  3 # empty line
  4 # Mark Moraes, University of Toronto
  5
  6 # ==> These comments added by author of this document.
  7
  8 if [ $# -eq 0 ]; then
  9 # ==> If no command line args present, then works on file redirected to
stdin.
 10     sed -e '1,/^$/d' -e '/^[       ]*$/d'
 11     # --> Delete empty lines and all lines until
 12     # --> first one beginning with white space.
 13 else
 14 # ==> If command line args present, then work on files named.
 15     for i do
 16             sed -e '1,/^$/d' -e '/^[       ]*$/d' $i
 17             # --> Ditto, as above.
 18     done
 19 fi
 20
 21 # ==> Exercise: Add error checking and other options.
 22 # ==>
 23 # ==> Note that the small sed script repeats, except for the arg passed.
 24 # ==> Does it make sense to embed it in a function? Why or why not?
```

Esempio A-14. ftpget: scaricare file via ftp

```
  1 #! /bin/sh
  2 # $Id: ftpget.sh,v 1.1.1.1 2003/06/25 22:41:32 giacomo Exp $
  3 # Script to perform batch anonymous ftp. Essentially converts a list of
  4 # of command line arguments into input to ftp.
  5 # Simple, and quick - written as a companion to ftplist
  6 # -h specifies the remote host (default prep.ai.mit.edu)
  7 # -d specifies the remote directory to cd to - you can provide a sequence
  8 # of -d options - they will be cd'ed to in turn. If the paths are relative,
  9 # make sure you get the sequence right. Be careful with relative paths -
```

```
 10 # there are far too many symlinks nowadays.
 11 # (default is the ftp login directory)
 12 # -v turns on the verbose option of ftp, and shows all responses from the
 13 # ftp server.
 14 # -f remotefile[:localfile] gets the remote file into localfile
 15 # -m pattern does an mget with the specified pattern. Remember to quote
 16 # shell characters.
 17 # -c does a local cd to the specified directory
 18 # For example,
 19 #   ftpget -h expo.lcs.mit.edu -d contrib -f xplaces.shar:xplaces.sh \
 20 #          -d ../pub/R3/fixes -c ~/fixes -m 'fix*'
 21 # will get xplaces.shar from ~ftp/contrib on expo.lcs.mit.edu, and put it in
 22 # xplaces.sh in the current working directory, and get all fixes from
 23 # ~ftp/pub/R3/fixes and put them in the ~/fixes directory.
 24 # Obviously, the sequence of the options is important, since the equivalent
 25 # commands are executed by ftp in corresponding order
 26 #
 27 # Mark Moraes (moraes@csri.toronto.edu), Feb 1, 1989
 28 # ==> Angle brackets changed to parens, so Docbook won't get indigestion.
 29 #
 30
 31
 32 # ==> These comments added by author of this document.
 33
 34 # PATH=/local/bin:/usr/ucb:/usr/bin:/bin
 35 # export PATH
 36 # ==> Above 2 lines from original script probably superfluous.
 37
 38 TMPFILE=/tmp/ftp.$$
 39 # ==> Creates temp file, using process id of script ($$)
 40 # ==> to construct filename.
 41
 42 SITE=`domainname`.toronto.edu
 43 # ==> 'domainname' similar to 'hostname'
 44 # ==> May rewrite this to parameterize this for general use.
 45
 46 usage="Usage: $0 [-h remotehost] [-d remotedirectory]... [-f
remfile:localfile]... \
 47            [-c localdirectory] [-m filepattern] [-v]"
 48 ftpflags="-i -n"
 49 verbflag=
 50 set -f             # So we can use globbing in -m
 51 set x `getopt vh:d:c:m:f: $*`
 52 if [ $? != 0 ]; then
 53     echo $usage
 54     exit 65
 55 fi
 56 shift
 57 trap 'rm -f ${TMPFILE} ; exit' 0 1 2 3 15
 58 echo "user anonymous ${USER-gnu}@${SITE} > ${TMPFILE}"
 59 # ==> Added quotes (recommended in complex echoes).
 60 echo binary >> ${TMPFILE}
 61 for i in $*   # ==> Parse command line args.
 62 do
 63     case $i in
 64     -v) verbflag=-v; echo hash >> ${TMPFILE}; shift;;
 65     -h) remhost=$2; shift 2;;
 66     -d) echo cd $2 >> ${TMPFILE};
 67         if [ x${verbflag} != x ]; then
 68             echo pwd >> ${TMPFILE};
 69         fi;
 70         shift 2;;
```

```
 71     -c) echo lcd $2 >> ${TMPFILE}; shift 2;;
 72     -m) echo mget "$2" >> ${TMPFILE}; shift 2;;
 73     -f) f1=`expr "$2" : "\([^:]*\).*"`; f2=`expr "$2" : "[^:]*:\(.*\)"`;
 74         echo get ${f1} ${f2} >> ${TMPFILE}; shift 2;;
 75     --) shift; break;;
 76     esac
 77 done
 78 if [ $# -ne 0 ]; then
 79     echo $usage
 80     exit 65   # ==> Changed from "exit 2" to conform with standard.
 81 fi
 82 if [ x${verbflag} != x ]; then
 83     ftpflags="${ftpflags} -v"
 84 fi
 85 if [ x${remhost} = x ]; then
 86     remhost=prep.ai.mit.edu
 87     # ==> Rewrite to match your favorite ftp site.
 88 fi
 89 echo quit >> ${TMPFILE}
 90 # ==> All commands saved in tempfile.
 91
 92 ftp ${ftpflags} ${remhost} < ${TMPFILE}
 93 # ==> Now, tempfile batch processed by ftp.
 94
 95 rm -f ${TMPFILE}
 96 # ==> Finally, tempfile deleted (you may wish to copy it to a logfile).
 97
 98
 99 # ==> Exercises:
100 # ==> ---------
101 # ==> 1) Add error checking.
102 # ==> 2) Add bells & whistles.
```

+

Antek Sawicki ha fornito lo script seguente che fa un uso molto intelligente degli operatori di sostituzione di parametro discussi in la Sezione 9.3.

Esempio A-15. password: generare password casuali di 8 caratteri

```
 1 #!/bin/bash
 2 # May need to be invoked with  #!/bin/bash2  on older machines.
 3 #
 4 # Random password generator for Bash 2.x by Antek Sawicki <tenox@tenox.tc>,
 5 # who generously gave permission to the document author to use it here.
 6 #
 7 # ==> Comments added by document author ==>
 8
 9
10 MATRIX="0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz"
11 # ==> Password will consist of alphanumeric characters.
12 LENGTH="8"
13 # ==> May change 'LENGTH' for longer password.
14
15
16 while [ "${n:=1}" -le "$LENGTH" ]
17 # ==> Recall that := is "default substitution" operator.
18 # ==> So, if 'n' has not been initialized, set it to 1.
19 do
20     PASS="$PASS${MATRIX:$(($RANDOM%${#MATRIX})):1}"
```

```
 21     # ==> Very clever, but tricky.
 22
 23     # ==> Starting from the innermost nesting...
 24     # ==> ${#MATRIX} returns length of array MATRIX.
 25
 26     # ==> $RANDOM%${#MATRIX} returns random number between 1
 27     # ==> and [length of MATRIX] - 1.
 28
 29     # ==> ${MATRIX:$(($RANDOM%${#MATRIX})):1}
 30     # ==> returns expansion of MATRIX at random position, by length 1.
 31     # ==> See {var:pos:len} parameter substitution in Chapter 9.
 32     # ==> and the associated examples.
 33
 34     # ==> PASS=... simply pastes this result onto previous PASS
(concatenation).
 35
 36     # ==> To visualize this more clearly, uncomment the following line
 37     #                 echo "$PASS"
 38     # ==> to see PASS being built up,
 39     # ==> one character at a time, each iteration of the loop.
 40
 41     let n+=1
 42     # ==> Increment 'n' for next pass.
 43 done
 44
 45 echo "$PASS"      # ==> Or, redirect to a file, as desired.
 46
 47 exit 0
```

+

James R. Van Zandt ha fornito questo script che fa uso delle named pipe ed è, parole sue, "vero esercizio di quoting ed escaping".

Esempio A-16. fifo: eseguire backup giornalieri utilizzando le named pipe

```
  1 #!/bin/bash
  2 # ==> Script by James R. Van Zandt, and used here with his permission.
  3
  4 # ==> Comments added by author of this document.
  5
  6
  7   HERE=`uname -n`    # ==> hostname
  8   THERE=bilbo
  9   echo "starting remote backup to $THERE at `date +%r`"
 10   # ==> `date +%r` returns time in 12-hour format, i.e. "08:08:34 PM".
 11
 12   # make sure /pipe really is a pipe and not a plain file
 13   rm -rf /pipe
 14   mkfifo /pipe       # ==> Create a "named pipe", named "/pipe".
 15
 16   # ==> 'su xyz' runs commands as user "xyz".
 17   # ==> 'ssh' invokes secure shell (remote login client).
 18   su xyz -c "ssh $THERE \"cat >/home/xyz/backup/${HERE}-daily.tar.gz\" <
/pipe"&
 19   cd /
 20   tar -czf - bin boot dev etc home info lib man root sbin share usr var
>/pipe
 21   # ==> Uses named pipe, /pipe, to communicate between processes:
 22   # ==> 'tar/gzip' writes to /pipe and 'ssh' reads from /pipe.
```

```
 23
 24   # ==> The end result is this backs up the main directories, from / on
down.
 25
 26   # ==> What are the advantages of a "named pipe" in this situation,
 27   # ==> as opposed to an "anonymous pipe", with |?
 28   # ==> Will an anonymous pipe even work here?
 29
 30
 31   exit 0
```

+

Questo script, inviato da Stephane Chazelas, dimostra che si possono generare numeri primi senza ricorrere agli array.

Esempio A-17. Generare numeri primi utilizzando l'operatore modulo

```
  1 #!/bin/bash
  2 # primes.sh: Generate prime numbers, without using arrays.
  3 # Script contributed by Stephane Chazelas.
  4
  5 #  This does *not* use the classic "Sieve of Eratosthenes" algorithm,
  6 #+ but instead uses the more intuitive method of testing each candidate
number
  7 #+ for factors (divisors), using the "%" modulo operator.
  8
  9
 10 LIMIT=1000                    # Primes 2 - 1000
 11
 12 Primes()
 13 {
 14  (( n = $1 + 1 ))             # Bump to next integer.
 15  shift                        # Next parameter in list.
 16 #  echo "_n=$n i=$i_"
 17
 18  if (( n == LIMIT ))
 19  then echo $*
 20  return
 21  fi
 22
 23  for i; do                    # "i" gets set to "@", previous values of $n.
 24 #   echo "-n=$n i=$i-"
 25    (( i * i > n )) && break   # Optimization.
 26    (( n % i )) && continue    # Sift out non-primes using modulo operator.
 27    Primes $n $@               # Recursion inside loop.
 28    return
 29    done
 30
 31    Primes $n $@ $n            # Recursion outside loop.
 32                               # Successively accumulate positional
parameters.
 33                               # "$@" is the accumulating list of primes.
 34 }
 35
 36 Primes 1
 37
 38 exit 0
 39
 40 #  Uncomment lines 16 and 24 to help figure out what is going on.
```

```
 41
 42 #  Compare the speed of this algorithm for generating primes
 43 #+ with the Sieve of Eratosthenes (ex68.sh).
 44
 45 #  Exercise: Rewrite this script without recursion, for faster execution.
```

+

Questa è la revisione di Rick Boivie dello script *tree* di Jordi Sanfeliu.

Esempio A-18. tree: visualizzare l'albero di una directory

```
  1 #!/bin/bash
  2 # tree.sh
  3
  4 #  Written by Rick Boivie.
  5 #  Used with permission.
  6 #  This is a revised and simplified version of a script
  7 #  by Jordi Sanfeliu (and patched by Ian Kjos).
  8 #  This script replaces the earlier version used in
  9 #+ previous releases of the Advanced Bash Scripting Guide.
 10
 11 # ==> Comments added by the author of this document.
 12
 13
 14 search () {
 15 for dir in `echo *`
 16 #  ==> `echo *` lists all the files in current working directory,
 17 #+ ==> without line breaks.
 18 #  ==> Similar effect to for dir in *
 19 #  ==> but "dir in `echo *`" will not handle filenames with blanks.
 20 do
 21   if [ -d "$dir" ] ; then # ==> If it is a directory (-d)...
 22   zz=0                    # ==> Temp variable, keeping track of directory
level.
 23   while [ $zz != $1 ]     # Keep track of inner nested loop.
 24     do
 25       echo -n "| "        # ==> Display vertical connector symbol,
 26                           # ==> with 2 spaces & no line feed in order to
indent.
 27       zz=`expr $zz + 1`   # ==> Increment zz.
 28     done
 29
 30     if [ -L "$dir" ] ; then # ==> If directory is a symbolic link...
 31       echo "+---$dir" `ls -l $dir | sed 's/^.*'$dir' //'`
 32       # ==> Display horiz. connector and list directory name, but...
 33       # ==> delete date/time part of long listing.
 34     else
 35       echo "+---$dir"       # ==> Display horizontal connector symbol...
 36       # ==> and print directory name.
 37       numdirs=`expr $numdirs + 1` # ==> Increment directory count.
 38       if cd "$dir" ; then # ==> If can move to subdirectory...
 39         search `expr $1 + 1` # with recursion ;-)
 40         # ==> Function calls itself.
 41         cd ..
 42       fi
 43     fi
 44   fi
 45 done
 46 }
```

```
47
48 if [ $# != 0 ] ; then
49   cd $1 # move to indicated directory.
50   #else # stay in current directory
51 fi
52
53 echo "Initial directory = `pwd`"
54 numdirs=0
55
56 search 0
57 echo "Total directories = $numdirs"
58
59 exit 0
```

Noah Friedman ha permesso l'uso del suo script *string function*, che riproduce, sostanzialmente, alcune delle funzioni per la manipolazione di stringhe della libreria C.

Esempio A-19. string functions: funzioni per stringhe simili a quelle del C

```
 1 #!/bin/bash
 2
 3 # string.bash --- bash emulation of string(3) library routines
 4 # Author: Noah Friedman <friedman@prep.ai.mit.edu>
 5 # ==>     Used with his kind permission in this document.
 6 # Created: 1992-07-01
 7 # Last modified: 1993-09-29
 8 # Public domain
 9
10 # Conversion to bash v2 syntax done by Chet Ramey
11
12 # Commentary:
13 # Code:
14
15 #:docstring strcat:
16 # Usage: strcat s1 s2
17 #
18 # Strcat appends the value of variable s2 to variable s1.
19 #
20 # Example:
21 #    a="foo"
22 #    b="bar"
23 #    strcat a b
24 #    echo $a
25 #    => foobar
26 #
27 #:end docstring:
28
29 ###;;;autoload   ==> Autoloading of function commented out.
30 function strcat ()
31 {
32     local s1_val s2_val
33
34     s1_val=${!1}                        # indirect variable expansion
35     s2_val=${!2}
36     eval "$1"=\'"${s1_val}${s2_val}"\'
37     # ==> eval $1='${s1_val}${s2_val}' avoids problems,
38     # ==> if one of the variables contains a single quote.
39 }
40
41 #:docstring strncat:
```

```
 42 # Usage: strncat s1 s2 $n
 43 #
 44 # Line strcat, but strncat appends a maximum of n characters from the value
 45 # of variable s2.  It copies fewer if the value of variabl s2 is shorter
 46 # than n characters.  Echoes result on stdout.
 47 #
 48 # Example:
 49 #    a=foo
 50 #    b=barbaz
 51 #    strncat a b 3
 52 #    echo $a
 53 #    => foobar
 54 #
 55 #:end docstring:
 56
 57 ###;;;autoload
 58 function strncat ()
 59 {
 60     local s1="$1"
 61     local s2="$2"
 62     local -i n="$3"
 63     local s1_val s2_val
 64
 65     s1_val=${!s1}                       # ==> indirect variable expansion
 66     s2_val=${!s2}
 67
 68     if [ ${#s2_val} -gt ${n} ]; then
 69        s2_val=${s2_val:0:$n}            # ==> substring extraction
 70     fi
 71
 72     eval "$s1"=\'"${s1_val}${s2_val}"\'
 73     # ==> eval $1='${s1_val}${s2_val}' avoids problems,
 74     # ==> if one of the variables contains a single quote.
 75 }
 76
 77 #:docstring strcmp:
 78 # Usage: strcmp $s1 $s2
 79 #
 80 # Strcmp compares its arguments and returns an integer less than, equal to,
 81 # or greater than zero, depending on whether string s1 is lexicographically
 82 # less than, equal to, or greater than string s2.
 83 #:end docstring:
 84
 85 ###;;;autoload
 86 function strcmp ()
 87 {
 88     [ "$1" = "$2" ] && return 0
 89
 90     [ "${1}" '<' "${2}" ] > /dev/null && return -1
 91
 92     return 1
 93 }
 94
 95 #:docstring strncmp:
 96 # Usage: strncmp $s1 $s2 $n
 97 #
 98 # Like strcmp, but makes the comparison by examining a maximum of n
 99 # characters (n less than or equal to zero yields equality).
100 #:end docstring:
101
102 ###;;;autoload
103 function strncmp ()
```

```
{
    if [ -z "${3}" -o "${3}" -le "0" ]; then
       return 0
    fi

    if [ ${3} -ge ${#1} -a ${3} -ge ${#2} ]; then
       strcmp "$1" "$2"
       return $?
    else
       s1=${1:0:$3}
       s2=${2:0:$3}
       strcmp $s1 $s2
       return $?
    fi
}

#:docstring strlen:
# Usage: strlen s
#
# Strlen returns the number of characters in string literal s.
#:end docstring:

###;;;autoload
function strlen ()
{
    eval echo "\${#${1}}"
    # ==> Returns the length of the value of the variable
    # ==> whose name is passed as an argument.
}

#:docstring strspn:
# Usage: strspn $s1 $s2
#
# Strspn returns the length of the maximum initial segment of string s1,
# which consists entirely of characters from string s2.
#:end docstring:

###;;;autoload
function strspn ()
{
    # Unsetting IFS allows whitespace to be handled as normal chars.
    local IFS=
    local result="${1%%[!${2}]*}"

    echo ${#result}
}

#:docstring strcspn:
# Usage: strcspn $s1 $s2
#
# Strcspn returns the length of the maximum initial segment of string s1,
# which consists entirely of characters not from string s2.
#:end docstring:

###;;;autoload
function strcspn ()
{
    # Unsetting IFS allows whitspace to be handled as normal chars.
    local IFS=
    local result="${1%%[${2}]*}"

    echo ${#result}
```

```
166 }
167
168 #:docstring strstr:
169 # Usage: strstr s1 s2
170 #
171 # Strstr echoes a substring starting at the first occurrence of string s2 in
172 # string s1, or nothing if s2 does not occur in the string.  If s2 points to
173 # a string of zero length, strstr echoes s1.
174 #:end docstring:
175
176 ###;;;autoload
177 function strstr ()
178 {
179     # if s2 points to a string of zero length, strstr echoes s1
180     [ ${#2} -eq 0 ] && { echo "$1" ; return 0; }
181
182     # strstr echoes nothing if s2 does not occur in s1
183     case "$1" in
184     *$2*) ;;
185     *) return 1;;
186     esac
187
188     # use the pattern matching code to strip off the match and everything
189     # following it
190     first=${1/$2*/}
191
192     # then strip off the first unmatched portion of the string
193     echo "${1##$first}"
194 }
195
196 #:docstring strtok:
197 # Usage: strtok s1 s2
198 #
199 # Strtok considers the string s1 to consist of a sequence of zero or more
200 # text tokens separated by spans of one or more characters from the
201 # separator string s2.  The first call (with a non-empty string s1
202 # specified) echoes a string consisting of the first token on stdout. The
203 # function keeps track of its position in the string s1 between separate
204 # calls, so that subsequent calls made with the first argument an empty
205 # string will work through the string immediately following that token.  In
206 # this way subsequent calls will work through the string s1 until no tokens
207 # remain.  The separator string s2 may be different from call to call.
208 # When no token remains in s1, an empty value is echoed on stdout.
209 #:end docstring:
210
211 ###;;;autoload
212 function strtok ()
213 {
214  :
215 }
216
217 #:docstring strtrunc:
218 # Usage: strtrunc $n $s1 {$s2} {$...}
219 #
220 # Used by many functions like strncmp to truncate arguments for comparison.
221 # Echoes the first n characters of each string s1 s2 ... on stdout.
222 #:end docstring:
223
224 ###;;;autoload
225 function strtrunc ()
226 {
227     n=$1 ; shift
```

```
228     for z; do
229         echo "${z:0:$n}"
230     done
231 }
232
233 # provide string
234
235 # string.bash ends here
236
237
238 # ===========================================================================
#
239 # ==> Everything below here added by the document author.
240
241 # ==> Suggested use of this script is to delete everything below here,
242 # ==> and "source" this file into your own scripts.
243
244 # strcat
245 string0=one
246 string1=two
247 echo
248 echo "Testing \"strcat\" function:"
249 echo "Original \"string0\" = $string0"
250 echo "\"string1\" = $string1"
251 strcat string0 string1
252 echo "New \"string0\" = $string0"
253 echo
254
255 # strlen
256 echo
257 echo "Testing \"strlen\" function:"
258 str=123456789
259 echo "\"str\" = $str"
260 echo -n "Length of \"str\" = "
261 strlen str
262 echo
263
264
265
266 # Exercise:
267 # --------
268 # Add code to test all the other string functions above.
269
270
271 exit 0
```

Esempio di array complesso, di Michael Zick, che utilizza il comando md5sum per codificare informazioni sulle directory.

Esempio A-20. Informazioni sulle directory

```
 1 #! /bin/bash
 2 # directory-info.sh
 3 # Parses and lists directory information.
 4
 5 # NOTE: Change lines 273 and 353 per "README" file.
 6
 7 # Michael Zick is the author of this script.
 8 # Used here with his permission.
 9
```

```
# Controls
# If overridden by command arguments, they must be in the order:
#   Arg1: "Descriptor Directory"
#   Arg2: "Exclude Paths"
#   Arg3: "Exclude Directories"
#
# Environment Settings override Defaults.
# Command arguments override Environment Settings.

# Default location for content addressed file descriptors.
MD5UCFS=${1:-${MD5UCFS:-'/tmpfs/ucfs'}}

# Directory paths never to list or enter
declare -a \
  EXCLUDE_PATHS=${2:-${EXCLUDE_PATHS:-'(/proc /dev /devfs /tmpfs)'}}

# Directories never to list or enter
declare -a \
  EXCLUDE_DIRS=${3:-${EXCLUDE_DIRS:-'(ucfs lost+found tmp wtmp)'}}

# Files never to list or enter
declare -a \
  EXCLUDE_FILES=${3:-${EXCLUDE_FILES:-'(core "Name with Spaces")'}}


# Here document used as a comment block.
: << LSfieldsDoc
# # # # # List Filesystem Directory Information # # # # #
#
#   ListDirectory "FileGlob" "Field-Array-Name"
# or
#   ListDirectory -of "FileGlob" "Field-Array-Filename"
#   '-of' meaning 'output to filename'
# # # # #

String format description based on: ls (GNU fileutils) version 4.0.36

Produces a line (or more) formatted:
inode permissions hard-links owner group ...
32736 -rw-------    1 mszick   mszick

size    day month date hh:mm:ss year path
2756608 Sun Apr 20 08:53:06 2003 /home/mszick/core

Unless it is formatted:
inode permissions hard-links owner group ...
266705 crw-rw----    1    root  uucp

major minor day month date hh:mm:ss year path
4,  68 Sun Apr 20 09:27:33 2003 /dev/ttyS4
NOTE: that pesky comma after the major number

NOTE: the 'path' may be multiple fields:
/home/mszick/core
/proc/982/fd/0 -> /dev/null
/proc/982/fd/1 -> /home/mszick/.xsession-errors
/proc/982/fd/13 -> /tmp/tmpfZVVOCs (deleted)
/proc/982/fd/7 -> /tmp/kde-mszick/ksycoca
/proc/982/fd/8 -> socket:[11586]
/proc/982/fd/9 -> pipe:[11588]

If that isn't enough to keep your parser guessing,
```

```
 72 either or both of the path components may be relative:
 73 ../Built-Shared -> Built-Static
 74 ../linux-2.4.20.tar.bz2 -> ../../../SRCS/linux-2.4.20.tar.bz2
 75
 76 The first character of the 11 (10?) character permissions field:
 77 's' Socket
 78 'd' Directory
 79 'b' Block device
 80 'c' Character device
 81 'l' Symbolic link
 82 NOTE: Hard links not marked - test for identical inode numbers
 83 on identical filesystems.
 84 All information about hard linked files are shared, except
 85 for the names and the name's location in the directory system.
 86 NOTE: A "Hard link" is known as a "File Alias" on some systems.
 87 '-' An undistingushed file
 88
 89 Followed by three groups of letters for: User, Group, Others
 90 Character 1: '-' Not readable; 'r' Readable
 91 Character 2: '-' Not writable; 'w' Writable
 92 Character 3, User and Group: Combined execute and special
 93 '-' Not Executable, Not Special
 94 'x' Executable, Not Special
 95 's' Executable, Special
 96 'S' Not Executable, Special
 97 Character 3, Others: Combined execute and sticky (tacky?)
 98 '-' Not Executable, Not Tacky
 99 'x' Executable, Not Tacky
100 't' Executable, Tacky
101 'T' Not Executable, Tacky
102
103 Followed by an access indicator
104 Haven't tested this one, it may be the eleventh character
105 or it may generate another field
106 ' ' No alternate access
107 '+' Alternate access
108 LSfieldsDoc
109
110
111 ListDirectory()
112 {
113     local -a T
114     local -i of=0          # Default return in variable
115 #   OLD_IFS=$IFS           # Using BASH default ' \t\n'
116
117     case "$#" in
118     3)      case "$1" in
119             -of)    of=1 ; shift ;;
120              * )    return 1 ;;
121             esac ;;
122     2)      : ;;           # Poor man's "continue"
123     *)      return 1 ;;
124     esac
125
126     # NOTE: the (ls) command is NOT quoted (")
127     T=( $(ls --inode --ignore-backups --almost-all --directory \
128     --full-time --color=none --time=status --sort=none \
129     --format=long $1) )
130
131     case $of in
132     # Assign T back to the array whose name was passed as $2
133             0) eval $2=\( \"\$\{T\[@\]\}\" \) ;;
```

```
134     # Write T into filename passed as $2
135             1) echo "${T[@]}" > "$2" ;;
136     esac
137     return 0
138    }
139
140 # # # # # Is that string a legal number? # # # # #
141 #
142 #   IsNumber "Var"
143 # # # # # There has to be a better way, sigh...
144
145 IsNumber()
146 {
147     local -i int
148     if [ $# -eq 0 ]
149     then
150             return 1
151     else
152             (let int=$1)  2>/dev/null
153             return $?      # Exit status of the let thread
154     fi
155 }
156
157 # # # # # Index Filesystem Directory Information # # # # #
158 #
159 #   IndexList "Field-Array-Name" "Index-Array-Name"
160 # or
161 #   IndexList -if Field-Array-Filename Index-Array-Name
162 #   IndexList -of Field-Array-Name Index-Array-Filename
163 #   IndexList -if -of Field-Array-Filename Index-Array-Filename
164 # # # # #
165
166 : << IndexListDoc
167 Walk an array of directory fields produced by ListDirectory
168
169 Having suppressed the line breaks in an otherwise line oriented
170 report, build an index to the array element which starts each line.
171
172 Each line gets two index entries, the first element of each line
173 (inode) and the element that holds the pathname of the file.
174
175 The first index entry pair (Line-Number==0) are informational:
176 Index-Array-Name[0] : Number of "Lines" indexed
177 Index-Array-Name[1] : "Current Line" pointer into Index-Array-Name
178
179 The following index pairs (if any) hold element indexes into
180 the Field-Array-Name per:
181 Index-Array-Name[Line-Number * 2] : The "inode" field element.
182 NOTE: This distance may be either +11 or +12 elements.
183 Index-Array-Name[(Line-Number * 2) + 1] : The "pathname" element.
184 NOTE: This distance may be a variable number of elements.
185 Next line index pair for Line-Number+1.
186 IndexListDoc
187
188
189
190 IndexList()
191 {
192     local -a LIST                  # Local of listname passed
193     local -a -i INDEX=( 0 0 )      # Local of index to return
194     local -i Lidx Lcnt
195     local -i if=0 of=0             # Default to variable names
```

```
196
197     case "$#" in                    # Simplistic option testing
198             0) return 1 ;;
199             1) return 1 ;;
200             2) : ;;                 # Poor man's continue
201             3) case "$1" in
202                     -if) if=1 ;;
203                     -of) of=1 ;;
204                      * ) return 1 ;;
205                esac ; shift ;;
206             4) if=1 ; of=1 ; shift ; shift ;;
207             *) return 1
208     esac
209
210     # Make local copy of list
211     case "$if" in
212             0) eval LIST=\( \"\$\{$1\[@\]\}\" \) ;;
213             1) LIST=( $(cat $1) ) ;;
214     esac
215
216     # Grok (grope?) the array
217     Lcnt=${#LIST[@]}
218     Lidx=0
219     until (( Lidx >= Lcnt ))
220     do
221     if IsNumber ${LIST[$Lidx]}
222     then
223             local -i inode name
224             local ft
225             inode=Lidx
226             local m=${LIST[$Lidx+2]}      # Hard Links field
227             ft=${LIST[$Lidx+1]:0:1}       # Fast-Stat
228             case $ft in
229             b)      ((Lidx+=12)) ;;       # Block device
230             c)      ((Lidx+=12)) ;;       # Character device
231             *)      ((Lidx+=11)) ;;       # Anything else
232             esac
233             name=Lidx
234             case $ft in
235             -)      ((Lidx+=1)) ;;        # The easy one
236             b)      ((Lidx+=1)) ;;        # Block device
237             c)      ((Lidx+=1)) ;;        # Character device
238             d)      ((Lidx+=1)) ;;        # The other easy one
239             l)      ((Lidx+=3)) ;;        # At LEAST two more fields
240 #  A little more elegance here would handle pipes,
241 #+ sockets, deleted files - later.
242             *)      until IsNumber ${LIST[$Lidx]} || ((Lidx >= Lcnt))
243                     do
244                             ((Lidx+=1))
245                     done
246                     ;;                    # Not required
247             esac
248             INDEX[${#INDEX[*]}]=$inode
249             INDEX[${#INDEX[*]}]=$name
250             INDEX[0]=${INDEX[0]}+1        # One more "line" found
251 # echo "Line: ${INDEX[0]} Type: $ft Links: $m Inode: \
252 # ${LIST[$inode]} Name: ${LIST[$name]}"
253
254     else
255             ((Lidx+=1))
256     fi
257     done
```

```
258     case "$of" in
259             0) eval $2=\( \"\$\{INDEX\[@\]\}\" \) ;;
260             1) echo "${INDEX[@]}" > "$2" ;;
261     esac
262     return 0                                # What could go wrong?
263 }
264
265 # # # # # Content Identify File # # # # #
266 #
267 #   DigestFile Input-Array-Name Digest-Array-Name
268 # or
269 #   DigestFile -if Input-FileName Digest-Array-Name
270 # # # # #
271
272 # Here document used as a comment block.
273 : <<DigestFilesDoc
274
275 The key (no pun intended) to a Unified Content File System (UCFS)
276 is to distinguish the files in the system based on their content.
277 Distinguishing files by their name is just, so, 20th Century.
278
279 The content is distinguished by computing a checksum of that content.
280 This version uses the md5sum program to generate a 128 bit checksum
281 representative of the file's contents.
282 There is a chance that two files having different content might
283 generate the same checksum using md5sum (or any checksum).  Should
284 that become a problem, then the use of md5sum can be replace by a
285 cyrptographic signature.  But until then...
286
287 The md5sum program is documented as outputting three fields (and it
288 does), but when read it appears as two fields (array elements).  This
289 is caused by the lack of whitespace between the second and third field.
290 So this function gropes the md5sum output and returns:
291     [0]     32 character checksum in hexidecimal (UCFS filename)
292     [1]     Single character: ' ' text file, '*' binary file
293     [2]     Filesystem (20th Century Style) name
294     Note: That name may be the character '-' indicating STDIN read.
295
296 DigestFilesDoc
297
298
299
300 DigestFile()
301 {
302     local if=0              # Default, variable name
303     local -a T1 T2
304
305     case "$#" in
306     3)      case "$1" in
307             -if)    if=1 ; shift ;;
308              * )    return 1 ;;
309             esac ;;
310     2)      : ;;            # Poor man's "continue"
311     *)      return 1 ;;
312     esac
313
314     case $if in
315     0) eval T1=\( \"\$\{$1\[@\]\}\" \)
316        T2=( $(echo ${T1[@]} | md5sum -) )
317        ;;
318     1) T2=( $(md5sum $1) )
319        ;;
```

```
320     esac
321
322     case ${#T2[@]} in
323     0) return 1 ;;
324     1) return 1 ;;
325     2) case ${T2[1]:0:1} in               # SanScrit-2.0.5
326        \*) T2[${#T2[@]}]=${T2[1]:1}
327            T2[1]=\*
328            ;;
329         *) T2[${#T2[@]}]=${T2[1]}
330            T2[1]=" "
331            ;;
332        esac
333        ;;
334     3) : ;; # Assume it worked
335     *) return 1 ;;
336     esac
337
338     local -i len=${#T2[0]}
339     if [ $len -ne 32 ] ; then return 1 ; fi
340     eval $2=\( \"\$\{T2\[@\]\}\" \)
341 }
342
343 # # # # # Locate File # # # # #
344 #
345 #   LocateFile [-l] FileName Location-Array-Name
346 # or
347 #   LocateFile [-l] -of FileName Location-Array-FileName
348 # # # # #
349
350 # A file location is Filesystem-id and inode-number
351
352 # Here document used as a comment block.
353 : <<StatFieldsDoc
354     Based on stat, version 2.2
355     stat -t and stat -lt fields
356     [0]     name
357     [1]     Total size
358             File - number of bytes
359             Symbolic link - string length of pathname
360     [2]     Number of (512 byte) blocks allocated
361     [3]     File type and Access rights (hex)
362     [4]     User ID of owner
363     [5]     Group ID of owner
364     [6]     Device number
365     [7]     Inode number
366     [8]     Number of hard links
367     [9]     Device type (if inode device) Major
368     [10]    Device type (if inode device) Minor
369     [11]    Time of last access
370             May be disabled in 'mount' with noatime
371             atime of files changed by exec, read, pipe, utime, mknod (mmap?)
372             atime of directories changed by addition/deletion of files
373     [12]    Time of last modification
374             mtime of files changed by write, truncate, utime, mknod
375             mtime of directories changed by addtition/deletion of files
376     [13]    Time of last change
377             ctime reflects time of changed inode information (owner, group
378             permissions, link count
379 -*-*- Per:
380     Return code: 0
381     Size of array: 14
```

```
382     Contents of array
383     Element 0: /home/mszick
384     Element 1: 4096
385     Element 2: 8
386     Element 3: 41e8
387     Element 4: 500
388     Element 5: 500
389     Element 6: 303
390     Element 7: 32385
391     Element 8: 22
392     Element 9: 0
393     Element 10: 0
394     Element 11: 1051221030
395     Element 12: 1051214068
396     Element 13: 1051214068
397
398     For a link in the form of linkname -> realname
399     stat -t  linkname returns the linkname (link) information
400     stat -lt linkname returns the realname information
401
402     stat -tf and stat -ltf fields
403     [0]     name
404     [1]     ID-0?          # Maybe someday, but Linux stat structure
405     [2]     ID-0?          # does not have either LABEL nor UUID
406                            # fields, currently information must come
407                            # from file-system specific utilities
408     These will be munged into:
409     [1]     UUID if possible
410     [2]     Volume Label if possible
411     Note: 'mount -l' does return the label and could return the UUID
412
413     [3]     Maximum length of filenames
414     [4]     Filesystem type
415     [5]     Total blocks in the filesystem
416     [6]     Free blocks
417     [7]     Free blocks for non-root user(s)
418     [8]     Block size of the filesystem
419     [9]     Total inodes
420     [10]    Free inodes
421
422 -*-*- Per:
423     Return code: 0
424     Size of array: 11
425     Contents of array
426     Element 0: /home/mszick
427     Element 1: 0
428     Element 2: 0
429     Element 3: 255
430     Element 4: ef53
431     Element 5: 2581445
432     Element 6: 2277180
433     Element 7: 2146050
434     Element 8: 4096
435     Element 9: 1311552
436     Element 10: 1276425
437
438 StatFieldsDoc
439
440
441 #   LocateFile [-l] FileName Location-Array-Name
442 #   LocateFile [-l] -of FileName Location-Array-FileName
443
```

```
444 LocateFile()
445 {
446     local -a LOC LOC1 LOC2
447     local lk="" of=0
448 
449     case "$#" in
450     0) return 1 ;;
451     1) return 1 ;;
452     2) : ;;
453     *) while (( "$#" > 2 ))
454        do
455           case "$1" in
456            -l) lk=-1 ;;
457           -of) of=1 ;;
458             *) return 1 ;;
459           esac
460        shift
461            done ;;
462     esac
463 
464 # More Sanscrit-2.0.5
465       # LOC1=( $(stat -t $lk $1) )
466       # LOC2=( $(stat -tf $lk $1) )
467       # Uncomment above two lines if system has "stat" command installed.
468     LOC=( ${LOC1[@]:0:1} ${LOC1[@]:3:11}
469           ${LOC2[@]:1:2} ${LOC2[@]:4:1} )
470 
471     case "$of" in
472            0) eval $2=\( \"\$\{LOC\[@\]\}\" \) ;;
473            1) echo "${LOC[@]}" > "$2" ;;
474     esac
475     return 0
476 # Which yields (if you are lucky, and have "stat" installed)
477 # -*-*- Location Discriptor -*-*-
478 #   Return code: 0
479 #   Size of array: 15
480 #   Contents of array
481 #   Element 0: /home/mszick               20th Century name
482 #   Element 1: 41e8               Type and Permissions
483 #   Element 2: 500                User
484 #   Element 3: 500                Group
485 #   Element 4: 303                Device
486 #   Element 5: 32385              inode
487 #   Element 6: 22                 Link count
488 #   Element 7: 0                  Device Major
489 #   Element 8: 0                  Device Minor
490 #   Element 9: 1051224608         Last Access
491 #   Element 10: 1051214068        Last Modify
492 #   Element 11: 1051214068        Last Status
493 #   Element 12: 0                 UUID (to be)
494 #   Element 13: 0                 Volume Label (to be)
495 #   Element 14: ef53              Filesystem type
496 }
497 
498 
499 
500 # And then there was some test code
501 
502 ListArray() # ListArray Name
503 {
504     local -a Ta
505 
```

```
506     eval Ta=\( \"\$\{$1\[@\]\}\" \)
507     echo
508     echo "-*-*- List of Array -*-*-"
509     echo "Size of array $1: ${#Ta[*]}"
510     echo "Contents of array $1:"
511     for (( i=0 ; i<${#Ta[*]} ; i++ ))
512     do
513         echo -e "\tElement $i: ${Ta[$i]}"
514     done
515     return 0
516 }
517
518 declare -a CUR_DIR
519 # For small arrays
520 ListDirectory "${PWD}" CUR_DIR
521 ListArray CUR_DIR
522
523 declare -a DIR_DIG
524 DigestFile CUR_DIR DIR_DIG
525 echo "The new \"name\" (checksum) for ${CUR_DIR[9]} is ${DIR_DIG[0]}"
526
527 declare -a DIR_ENT
528 # BIG_DIR # For really big arrays - use a temporary file in ramdisk
529 # BIG-DIR # ListDirectory -of "${CUR_DIR[11]}/*" "/tmpfs/junk2"
530 ListDirectory "${CUR_DIR[11]}/*" DIR_ENT
531
532 declare -a DIR_IDX
533 # BIG-DIR # IndexList -if "/tmpfs/junk2" DIR_IDX
534 IndexList DIR_ENT DIR_IDX
535
536 declare -a IDX_DIG
537 # BIG-DIR # DIR_ENT=( $(cat /tmpfs/junk2) )
538 # BIG-DIR # DigestFile -if /tmpfs/junk2 IDX_DIG
539 DigestFile DIR_ENT IDX_DIG
540 # Small (should) be able to parallize IndexList & DigestFile
541 # Large (should) be able to parallize IndexList & DigestFile & the
assignment
542 echo "The \"name\" (checksum) for the contents of ${PWD} is ${IDX_DIG[0]}"
543
544 declare -a FILE_LOC
545 LocateFile ${PWD} FILE_LOC
546 ListArray FILE_LOC
547
548 exit 0
```

Stephane Chazelas dà un esempio di programmazione object-oriented con uno script Bash.

Esempio A-21. Database object-oriented

```
 1 #!/bin/bash
 2 # obj-oriented.sh: Object-oriented programming in a shell script.
 3 # Script by Stephane Chazelas.
 4
 5
 6 person.new()        # Looks almost like a class declaration in C++.
 7 {
 8   local obj_name=$1 name=$2 firstname=$3 birthdate=$4
 9
10   eval "$obj_name.set_name() {
11           eval \"$obj_name.get_name() {
12                   echo \$1
```

```
 13                  }\"
 14         }"
 15
 16   eval "$obj_name.set_firstname() {
 17           eval \"$obj_name.get_firstname() {
 18                   echo \$1
 19                  }\"
 20         }"
 21
 22   eval "$obj_name.set_birthdate() {
 23           eval \"$obj_name.get_birthdate() {
 24             echo \$1
 25           }\"
 26           eval \"$obj_name.show_birthdate() {
 27             echo \$(date -d \"1/1/1970 0:0:\$1 GMT\")
 28           }\"
 29           eval \"$obj_name.get_age() {
 30             echo \$(( (\$(date +%s) - \$1) / 3600 / 24 / 365 ))
 31           }\"
 32         }"
 33
 34   $obj_name.set_name $name
 35   $obj_name.set_firstname $firstname
 36   $obj_name.set_birthdate $birthdate
 37 }
 38
 39 echo
 40
 41 person.new self Bozeman Bozo 101272413
 42 # Create an instance of "person.new" (actually passing args to the
function).
 43
 44 self.get_firstname       #   Bozo
 45 self.get_name            #   Bozeman
 46 self.get_age             #   28
 47 self.get_birthdate       #   101272413
 48 self.show_birthdate      #   Sat Mar 17 20:13:33 MST 1973
 49
 50 echo
 51
 52 # typeset -f
 53 # to see the created functions (careful, it scrolls off the page).
 54
 55 exit 0
```

Ora uno script che fa qualcosa di veramente utile: installare e montare quelle graziose "chiavi" USB.

Esempio A-22. Montare le chiavi di memoria USB

```
  1 #!/bin/bash
  2 # ==> usb.sh
  3 # ==> Script for mounting and installing pen/keychain USB storage devices.
  4 # ==> Runs as root at system startup (see below).
  5
  6 #  This code is free software covered by GNU GPL license version 2 or above.
  7 #  Please refer to http://www.gnu.org/ for the full license text.
  8 #
  9 #  Some code lifted from usb-mount by Michael Hamilton's usb-mount (LGPL)
 10 #+ see http://users.actrix.co.nz/michael/usbmount.html
```

```
 11 #
 12 #  INSTALL
 13 #  -------
 14 #  Put this in /etc/hotplug/usb/diskonkey.
 15 #  Then look in /etc/hotplug/usb.distmap, and copy all usb-storage entries
 16 #+ into /etc/hotplug/usb.usermap, substituting "usb-storage" for
"diskonkey".
 17 #  Otherwise this code is only run during the kernel module
invocation/removal
 18 #+ (at least in my tests), which defeats the purpose.
 19 #
 20 #  TODO
 21 #  ----
 22 #  Handle more than one diskonkey device at one time (e.g. /dev/diskonkey1
 23 #+ and /mnt/diskonkey1), etc. The biggest problem here is the handling in
 24 #+ devlabel, which I haven't yet tried.
 25 #
 26 #  AUTHOR and SUPPORT
 27 #  ------------------
 28 #  Konstantin Riabitsev, <icon linux duke edu>.
 29 #  Send any problem reports to my email address at the moment.
 30 #
 31 # ==> Comments added by ABS Guide author.
 32
 33
 34
 35 SYMLINKDEV=/dev/diskonkey
 36 MOUNTPOINT=/mnt/diskonkey
 37 DEVLABEL=/sbin/devlabel
 38 DEVLABELCONFIG=/etc/sysconfig/devlabel
 39 IAM=$0
 40
 41 ##
 42 # Functions lifted near-verbatim from usb-mount code.
 43 #
 44 function allAttachedScsiUsb {
 45     find /proc/scsi/ -path '/proc/scsi/usb-storage*' -type f | xargs grep -l
'Attached: Yes'
 46 }
 47 function scsiDevFromScsiUsb {
 48     echo $1 | awk -F"[-/]" '{ n=$(NF-1);  print "/dev/sd"
substr("abcdefghijklmnopqrstuvwxyz", n+1,
 49  1) }'
 50 }
 51
 52 if [ "${ACTION}" = "add" ] && [ -f "${DEVICE}" ]; then
 53     ##
 54     # lifted from usbcam code.
 55     #
 56     if [ -f /var/run/console.lock ]; then
 57         CONSOLEOWNER=`cat /var/run/console.lock`
 58     elif [ -f /var/lock/console.lock ]; then
 59         CONSOLEOWNER=`cat /var/lock/console.lock`
 60     else
 61         CONSOLEOWNER=
 62     fi
 63     for procEntry in $(allAttachedScsiUsb); do
 64         scsiDev=$(scsiDevFromScsiUsb $procEntry)
 65         #  Some bug with usb-storage?
 66         #  Partitions are not in /proc/partitions until they are accessed
 67         #+ somehow.
 68         /sbin/fdisk -l $scsiDev >/dev/null
```

```
 69         ##
 70         #  Most devices have partitioning info, so the data would be on
 71         #+ /dev/sd?1. However, some stupider ones don't have any
partitioning
 72         #+ and use the entire device for data storage. This tries to
 73         #+ guess semi-intelligently if we have a /dev/sd?1 and if not, then
 74         #+ it uses the entire device and hopes for the better.
 75         #
 76         if grep -q `basename $scsiDev`1 /proc/partitions; then
 77             part="$scsiDev""1"
 78         else
 79             part=$scsiDev
 80         fi
 81         ##
 82         #  Change ownership of the partition to the console user so they can
 83         #+ mount it.
 84         #
 85         if [ ! -z "$CONSOLEOWNER" ]; then
 86             chown $CONSOLEOWNER:disk $part
 87         fi
 88         ##
 89         # This checks if we already have this UUID defined with devlabel.
 90         # If not, it then adds the device to the list.
 91         #
 92         prodid=`$DEVLABEL printid -d $part`
 93         if ! grep -q $prodid $DEVLABELCONFIG; then
 94             # cross our fingers and hope it works
 95             $DEVLABEL add -d $part -s $SYMLINKDEV 2>/dev/null
 96         fi
 97         ##
 98         # Check if the mount point exists and create if it doesn't.
 99         #
100         if [ ! -e $MOUNTPOINT ]; then
101             mkdir -p $MOUNTPOINT
102         fi
103         ##
104         # Take care of /etc/fstab so mounting is easy.
105         #
106         if ! grep -q "^$SYMLINKDEV" /etc/fstab; then
107             # Add an fstab entry
108             echo -e \
109                 "$SYMLINKDEV\t\t$MOUNTPOINT\t\tauto\tnoauto,owner,kudzu 0 0"
\
110                 >> /etc/fstab
111         fi
112     done
113     if [ ! -z "$REMOVER" ]; then
114         ##
115         # Make sure this script is triggered on device removal.
116         #
117         mkdir -p `dirname $REMOVER`
118         ln -s $IAM $REMOVER
119     fi
120 elif [ "${ACTION}" = "remove" ]; then
121     ##
122     # If the device is mounted, unmount it cleanly.
123     #
124     if grep -q "$MOUNTPOINT" /etc/mtab; then
125         # unmount cleanly
126         umount -l $MOUNTPOINT
127     fi
128     ##
```

```
129     # Remove it from /etc/fstab if it's there.
130     #
131     if grep -q "^$SYMLINKDEV" /etc/fstab; then
132         grep -v "^$SYMLINKDEV" /etc/fstab > /etc/.fstab.new
133         mv -f /etc/.fstab.new /etc/fstab
134     fi
135 fi
136
137 exit 0
```

Ecco qualcosa che riscalderà i cuori di webmaster e insegnanti di ogni dove: uno script che salva i weblog.

Esempio A-23. Preservare i weblog

```
  1 #!/bin/bash
  2 # archiveweblogs.sh v1.0
  3
  4 # Troy Engel <tengel@fluid.com>
  5 # Slightly modified by document author.
  6 # Used with permission.
  7 #
  8 #  This script will preserve the normally rotated and
  9 #+ thrown away weblogs from a default RedHat/Apache installation.
 10 #  It will save the files with a date/time stamp in the filename,
 11 #+ bzipped, to a given directory.
 12 #
 13 #  Run this from crontab nightly at an off hour,
 14 #+ as bzip2 can suck up some serious CPU on huge logs:
 15 #  0 2 * * * /opt/sbin/archiveweblogs.sh
 16
 17
 18 PROBLEM=66
 19
 20 # Set this to your backup dir.
 21 BKP_DIR=/opt/backups/weblogs
 22
 23 # Default Apache/RedHat stuff
 24 LOG_DAYS="4 3 2 1"
 25 LOG_DIR=/var/log/httpd
 26 LOG_FILES="access_log error_log"
 27
 28 # Default RedHat program locations
 29 LS=/bin/ls
 30 MV=/bin/mv
 31 ID=/usr/bin/id
 32 CUT=/bin/cut
 33 COL=/usr/bin/column
 34 BZ2=/usr/bin/bzip2
 35
 36 # Are we root?
 37 USER=`$ID -u`
 38 if [ "X$USER" != "X0" ]; then
 39   echo "PANIC: Only root can run this script!"
 40   exit $PROBLEM
 41 fi
 42
 43 # Backup dir exists/writable?
 44 if [ ! -x $BKP_DIR ]; then
 45   echo "PANIC: $BKP_DIR doesn't exist or isn't writable!"
```

```
 46   exit $PROBLEM
 47 fi
 48
 49 # Move, rename and bzip2 the logs
 50 for logday in $LOG_DAYS; do
 51   for logfile in $LOG_FILES; do
 52     MYFILE="$LOG_DIR/$logfile.$logday"
 53     if [ -w $MYFILE ]; then
 54       DTS=`$LS -lgo --time-style=+%Y%m%d $MYFILE | $COL -t | $CUT -d ' ' -
f7`
 55       $MV $MYFILE $BKP_DIR/$logfile.$DTS
 56       $BZ2 $BKP_DIR/$logfile.$DTS
 57     else
 58       # Only spew an error if the file exits (ergo non-writable).
 59       if [ -f $MYFILE ]; then
 60         echo "ERROR: $MYFILE not writable. Skipping."
 61       fi
 62     fi
 63   done
 64 done
 65
 66 exit 0
```

Come impedire alla shell di espandere e reinterpretare le stringhe?

Esempio A-24. Proteggere le stringhe letterali

```
  1 #! /bin/bash
  2 # protect_literal.sh
  3
  4 # set -vx
  5
  6 :<<-'_Protect_Literal_String_Doc'
  7
  8     Copyright (c) Michael S. Zick, 2003; All Rights Reserved
  9     License: Unrestricted reuse in any form, for any purpose.
 10     Warranty: None
 11     Revision: $ID$
 12
 13     Documentation redirected to the Bash no-operation.
 14     Bash will '/dev/null' this block when the script is first read.
 15     (Uncomment the above set command to see this action.)
 16
 17     Remove the first (Sha-Bang) line when sourcing this as a library
 18     procedure.  Also comment out the example use code in the two
 19     places where shown.
 20
 21
 22     Usage:
 23         _protect_literal_str 'Whatever string meets your ${fancy}'
 24         Just echos the argument to standard out, hard quotes
 25         restored.
 26
 27         $(_protect_literal_str 'Whatever string meets your ${fancy}')
 28         as the right-hand-side of an assignment statement.
 29
 30     Does:
 31         As the right-hand-side of an assignment, preserves the
 32         hard quotes protecting the contents of the literal during
 33         assignment.
 34
```

```
35     Notes:
36         The strange names (_*) are used to avoid trampling on
37         the user's chosen names when this is sourced as a
38         library.
39
40 _Protect_Literal_String_Doc
41
42 # The 'for illustration' function form
43
44 _protect_literal_str() {
45
46 # Pick an un-used, non-printing character as local IFS.
47 # Not required, but shows that we are ignoring it.
48     local IFS=$'\x1B'               # \ESC character
49
50 # Enclose the All-Elements-Of in hard quotes during assignment.
51     local tmp=$'\x27'$@$'\x27'
52 #    local tmp=$'\''$@$'\''         # Even uglier.
53
54     local len=${#tmp}               # Info only.
55     echo $tmp is $len long.         # Output AND information.
56 }
57
58 # This is the short-named version.
59 _pls() {
60     local IFS=$'x1B'                # \ESC character (not required)
61     echo $'\x27'$@$'\x27'           # Hard quoted parameter glob
62 }
63
64 # :<<-'_Protect_Literal_String_Test'
65 # # # Remove the above "# " to disable this code. # # #
66
67 # See how that looks when printed.
68 echo
69 echo "- - Test One - -"
70 _protect_literal_str 'Hello $user'
71 _protect_literal_str 'Hello "${username}"'
72 echo
73
74 # Which yields:
75 # - - Test One - -
76 # 'Hello $user' is 13 long.
77 # 'Hello "${username}"' is 21 long.
78
79 #  Looks as expected, but why all of the trouble?
80 #  The difference is hidden inside the Bash internal order
81 #+ of operations.
82 #  Which shows when you use it on the RHS of an assignment.
83
84 # Declare an array for test values.
85 declare -a arrayZ
86
87 # Assign elements with various types of quotes and escapes.
88 arrayZ=( zero "$(_pls 'Hello ${Me}')" 'Hello ${You}' "\'Pass: ${pw}\'" )
89
90 # Now list that array and see what is there.
91 echo "- - Test Two - -"
92 for (( i=0 ; i<${#arrayZ[*]} ; i++ ))
93 do
94     echo  Element $i: ${arrayZ[$i]} is: ${#arrayZ[$i]} long.
95 done
96 echo
```

```
 97
 98 # Which yields:
 99 # - - Test Two - -
100 # Element 0: zero is: 4 long.           # Our marker element
101 # Element 1: 'Hello ${Me}' is: 13 long. # Our "$(_pls '...' )"
102 # Element 2: Hello ${You} is: 12 long.  # Quotes are missing
103 # Element 3: \'Pass: \' is: 10 long.    # ${pw} expanded to nothing
104
105 # Now make an assignment with that result.
106 declare -a array2=( ${arrayZ[@]} )
107
108 # And print what happened.
109 echo "- - Test Three - -"
110 for (( i=0 ; i<${#array2[*]} ; i++ ))
111 do
112     echo  Element $i: ${array2[$i]} is: ${#array2[$i]} long.
113 done
114 echo
115
116 # Which yields:
117 # - - Test Three - -
118 # Element 0: zero is: 4 long.           # Our marker element.
119 # Element 1: Hello ${Me} is: 11 long.   # Intended result.
120 # Element 2: Hello is: 5 long.          # ${You} expanded to nothing.
121 # Element 3: 'Pass: is: 6 long.         # Split on the whitespace.
122 # Element 4: ' is: 1 long.              # The end quote is here now.
123
124 #  Our Element 1 has had its leading and trailing hard quotes stripped.
125 #  Although not shown, leading and trailing whitespace is also stripped.
126 #  Now that the string contents are set, Bash will always, internally,
127 #+ hard quote the contents as required during its operations.
128
129 #  Why?
130 #  Considering our "$(_pls 'Hello ${Me}')" construction:
131 #  " ... " -> Expansion required, strip the quotes.
132 #  $( ... ) -> Replace with the result of..., strip this.
133 #  _pls ' ... ' -> called with literal arguments, strip the quotes.
134 #  The result returned includes hard quotes; BUT the above processing
135 #+ has already been done, so they become part of the value assigned.
136 #
137 #  Similarly, during further usage of the string variable, the ${Me}
138 #+ is part of the contents (result) and survives any operations
139 #  (Until explicitly told to evaluate the string).
140
141 #  Hint: See what happens when the hard quotes ($'\x27') are replaced
142 #+ with soft quotes ($'\x22') in the above procedures.
143 #  Interesting also is to remove the addition of any quoting.
144
145 # _Protect_Literal_String_Test
146 # # # Remove the above "# " to disable this code. # # #
147
148 exit 0
```

E se si *volesse* che la shell espanda e reinterpreti le stringhe?

Esempio A-25. Stringhe letterali non protette

```
 1 #! /bin/bash
 2 # unprotect_literal.sh
 3
 4 # set -vx
```

```
 5
 6 :<<-'_UnProtect_Literal_String_Doc'
 7
 8     Copyright (c) Michael S. Zick, 2003; All Rights Reserved
 9     License: Unrestricted reuse in any form, for any purpose.
10     Warranty: None
11     Revision: $ID$
12
13     Documentation redirected to the Bash no-operation. Bash will
14     '/dev/null' this block when the script is first read.
15     (Uncomment the above set command to see this action.)
16
17     Remove the first (Sha-Bang) line when sourcing this as a library
18     procedure.  Also comment out the example use code in the two
19     places where shown.
20
21
22     Usage:
23         Complement of the "$(_pls 'Literal String')" function.
24         (See the protect_literal.sh example.)
25
26         StringVar=$(_upls ProtectedSringVariable)
27
28     Does:
29         When used on the right-hand-side of an assignment statement;
30         makes the substitions embedded in the protected string.
31
32     Notes:
33         The strange names (_*) are used to avoid trampling on
34         the user's chosen names when this is sourced as a
35         library.
36
37
38 _UnProtect_Literal_String_Doc
39
40 _upls() {
41     local IFS=$'x1B'                # \ESC character (not required)
42     eval echo $@                    # Substitution on the glob.
43 }
44
45 # :<<-'_UnProtect_Literal_String_Test'
46 # # # Remove the above "# " to disable this code. # # #
47
48
49 _pls() {
50     local IFS=$'x1B'                # \ESC character (not required)
51     echo $'\x27'$@$'\x27'           # Hard quoted parameter glob
52 }
53
54 # Declare an array for test values.
55 declare -a arrayZ
56
57 # Assign elements with various types of quotes and escapes.
58 arrayZ=( zero "$(_pls 'Hello ${Me}')" 'Hello ${You}' "\'Pass: ${pw}\'" )
59
60 # Now make an assignment with that result.
61 declare -a array2=( ${arrayZ[@]} )
62
63 # Which yielded:
64 # - - Test Three - -
65 # Element 0: zero is: 4 long            # Our marker element.
66 # Element 1: Hello ${Me} is: 11 long    # Intended result.
```

```
 67 # Element 2: Hello is: 5 long           # ${You} expanded to nothing.
 68 # Element 3: 'Pass: is: 6 long          # Split on the whitespace.
 69 # Element 4: ' is: 1 long               # The end quote is here now.
 70
 71 # set -vx
 72
 73 #  Initialize 'Me' to something for the embedded ${Me} substitution.
 74 #  This needs to be done ONLY just prior to evaluating the
 75 #+ protected string.
 76 #  (This is why it was protected to begin with.)
 77
 78 Me="to the array guy."
 79
 80 # Set a string variable destination to the result.
 81 newVar=$(_upls ${array2[1]})
 82
 83 # Show what the contents are.
 84 echo $newVar
 85
 86 # Do we really need a function to do this?
 87 newerVar=$(eval echo ${array2[1]})
 88 echo $newerVar
 89
 90 #  I guess not, but the _upls function gives us a place to hang
 91 #+ the documentation on.
 92 #  This helps when we forget what a # construction like:
 93 #+ $(eval echo ... ) means.
 94
 95 # What if Me isn't set when the protected string is evaluated?
 96 unset Me
 97 newestVar=$(_upls ${array2[1]})
 98 echo $newestVar
 99
100 # Just gone, no hints, no runs, no errors.
101
102 #  Why in the world?
103 #  Setting the contents of a string variable containing character
104 #+ sequences that have a meaning to Bash is a general problem in
105 #+ script programming.
106 #
107 #  This problem is now solved in eight lines of code
108 #+ (and four pages of description).
109
110 #  Where is all this going?
111 #  Dynamic content Web pages as an array of Bash strings.
112 #  Content set per request by a Bash 'eval' command
113 #+ on the stored page template.
114 #  Not intended to replace PHP, just an interesting thing to do.
115 ###
116 #  Don't have a webserver application?
117 #  No problem, check the example directory of the Bash source;
118 #+ there is a Bash script for that also.
119
120 # _UnProtect_Literal_String_Test
121 # # # Remove the above "# " to disable this code. # # #
122
123 exit 0
```

Questo è uno script molto potente che aiuta a scovare gli spammer.

Esempio A-26. Identificare uno spammer

```
#!/bin/bash

# $Id: is_spammer.bash,v 1.12.2.11 2004/10/01 21:42:33 mszick Exp $
# Above line is RCS info.

# The latest version of this script is available from
ftp://ftp.morethan.org.
#
# Spammer-identification
# by Michael S. Zick
# Used in the ABS Guide with permission.



############################################################
# Documentation
# See also "Quickstart" at end of script.
############################################################

:<<-'__is_spammer_Doc_'

    Copyright (c) Michael S. Zick, 2004
    License: Unrestricted reuse in any form, for any purpose.
    Warranty: None -{Its a script; the user is on their own.}-

Impatient?
    Application code: goto "# # # Hunt the Spammer' program code # # #"
    Example output: ":<<-'_is_spammer_outputs_'"
    How to use: Enter script name without arguments.
                Or goto "Quickstart" at end of script.

Provides
    Given a domain name or IP(v4) address as input:

    Does an exhaustive set of queries to find the associated
    network resources (short of recursing into TLDs).

    Checks the IP(v4) addresses found against Blacklist
    nameservers.

    If found to be a blacklisted IP(v4) address,
    reports the blacklist text records.
    (Usually hyper-links to the specific report.)

Requires
    A working Internet connection.
    (Exercise: Add check and/or abort if not on-line when running script.)
    Bash with arrays (2.05b+).

    The external program 'dig' --
    a utility program provided with the 'bind' set of programs.
    Specifically, the version which is part of Bind series 9.x
    See: http://www.isc.org

    All usages of 'dig' are limited to wrapper functions,
    which may be rewritten as required.
    See: dig_wrappers.bash for details.
         ("Additional documentation" -- below)

Usage
    Script requires a single argument, which may be:
    1) A domain name;
```

```
 62     2) An IP(v4) address;
 63     3) A filename, with one name or address per line.
 64
 65     Script accepts an optional second argument, which may be:
 66     1) A Blacklist server name;
 67     2) A filename, with one Blacklist server name per line.
 68
 69     If the second argument is not provided, the script uses
 70     a built-in set of (free) Blacklist servers.
 71
 72     See also, the Quickstart at the end of this script (after 'exit').
 73
 74 Return Codes
 75     0 - All OK
 76     1 - Script failure
 77     2 - Something is Blacklisted
 78
 79 Optional environment variables
 80     SPAMMER_TRACE
 81         If set to a writable file,
 82         script will log an execution flow trace.
 83
 84     SPAMMER_DATA
 85         If set to a writable file, script will dump its
 86         discovered data in the form of GraphViz file.
 87         See: http://www.research.att.com/sw/tools/graphviz
 88
 89     SPAMMER_LIMIT
 90         Limits the depth of resource tracing.
 91
 92         Default is 2 levels.
 93
 94         A setting of 0 (zero) means 'unlimited' . . .
 95           Caution: script might recurse the whole Internet!
 96
 97         A limit of 1 or 2 is most useful when processing
 98         a file of domain names and addresses.
 99         A higher limit can be useful when hunting spam gangs.
100
101
102 Additional documentation
103     Download the archived set of scripts
104     explaining and illustrating the function contained within this script.
105     http://personal.riverusers.com/mszick_clf.tar.bz2
106
107
108 Study notes
109     This script uses a large number of functions.
110     Nearly all general functions have their own example script.
111     Each of the example scripts have tutorial level comments.
112
113 Scripting project
114     Add support for IP(v6) addresses.
115     IP(v6) addresses are recognized but not processed.
116
117 Advanced project
118     Add the reverse lookup detail to the discovered information.
119
120     Report the delegation chain and abuse contacts.
121
122     Modify the GraphViz file output to include the
123     newly discovered information.
```

```
124
125 __is_spammer_Doc_
126
127 ############################################################
128
129
130
131
132 #### Special IFS settings used for string parsing. ####
133
134 # Whitespace == :Space:Tab:Line Feed:Carriage Return:
135 WSP_IFS=$'\x20'$'\x09'$'\x0A'$'\x0D'
136
137 # No Whitespace == Line Feed:Carriage Return
138 NO_WSP=$'\x0A'$'\x0D'
139
140 # Field separator for dotted decimal IP addresses
141 ADR_IFS=${NO_WSP}'.'
142
143 # Array to dotted string conversions
144 DOT_IFS='.'${WSP_IFS}
145
146 # # # Pending operations stack machine # # #
147 # This set of functions described in func_stack.bash.
148 # (See "Additional documentation" above.)
149 # # #
150
151 # Global stack of pending operations.
152 declare -f -a _pending_
153 # Global sentinel for stack runners
154 declare -i _p_ctrl_
155 # Global holder for currently executing function
156 declare -f _pend_current_
157
158 # # # Debug version only - remove for regular use # # #
159 #
160 # The function stored in _pend_hook_ is called
161 # immediately before each pending function is
162 # evaluated.  Stack clean, _pend_current_ set.
163 #
164 # This thingy demonstrated in pend_hook.bash.
165 declare -f _pend_hook_
166 # # #
167
168 # The do nothing function
169 pend_dummy() { : ; }
170
171 # Clear and initialize the function stack.
172 pend_init() {
173     unset _pending_[@]
174     pend_func pend_stop_mark
175     _pend_hook_='pend_dummy'  # Debug only
176 }
177
178 # Discard the top function on the stack.
179 pend_pop() {
180     if [ ${#_pending_[@]} -gt 0 ]
181     then
182         local -i _top_
183         _top_=${#_pending_[@]}-1
184         unset _pending_[$_top_]
185     fi
```

```
}

# pend_func function_name [$(printf '%q\n' arguments)]
pend_func() {
    local IFS=${NO_WSP}
    set -f
    _pending_[${#_pending_[@]}]=$@
    set +f
}

# The function which stops the release:
pend_stop_mark() {
    _p_ctrl_=0
}

pend_mark() {
    pend_func pend_stop_mark
}

# Execute functions until 'pend_stop_mark' . . .
pend_release() {
    local -i _top_             # Declare _top_ as integer.
    _p_ctrl_=${#_pending_[@]}
    while [ ${_p_ctrl_} -gt 0 ]
    do
       _top_=${#_pending_[@]}-1
       _pend_current_=${_pending_[$_top_]}
       unset _pending_[$_top_]
       $_pend_hook_            # Debug only
       eval $_pend_current_
    done
}

# Drop functions until 'pend_stop_mark' . . .
pend_drop() {
    local -i _top_
    local _pd_ctrl_=${#_pending_[@]}
    while [ ${_pd_ctrl_} -gt 0 ]
    do
       _top_=$_pd_ctrl_-1
       if [ "${_pending_[$_top_]}" == 'pend_stop_mark' ]
       then
           unset _pending_[$_top_]
           break
       else
           unset _pending_[$_top_]
           _pd_ctrl_=$_top_
       fi
    done
    if [ ${#_pending_[@]} -eq 0 ]
    then
        pend_func pend_stop_mark
    fi
}

#### Array editors ####

# This function described in edit_exact.bash.
# (See "Additional documentation," above.)
# edit_exact <excludes_array_name> <target_array_name>
edit_exact() {
    [ $# -eq 2 ] ||
```

```
248     [ $# -eq 3 ] || return 1
249     local -a _ee_Excludes
250     local -a _ee_Target
251     local _ee_x
252     local _ee_t
253     local IFS=${NO_WSP}
254     set -f
255     eval _ee_Excludes=\( \$\{$1\[@\]\} \)
256     eval _ee_Target=\( \$\{$2\[@\]\} \)
257     local _ee_len=${#_ee_Target[@]}     # Original length.
258     local _ee_cnt=${#_ee_Excludes[@]}   # Exclude list length.
259     [ ${_ee_len} -ne 0 ] || return 0    # Can't edit zero length.
260     [ ${_ee_cnt} -ne 0 ] || return 0    # Can't edit zero length.
261     for (( x = 0; x < ${_ee_cnt} ; x++ ))
262     do
263         _ee_x=${_ee_Excludes[$x]}
264         for (( n = 0 ; n < ${_ee_len} ; n++ ))
265         do
266             _ee_t=${_ee_Target[$n]}
267             if [ x"${_ee_t}" == x"${_ee_x}" ]
268             then
269                 unset _ee_Target[$n]     # Discard match.
270                 [ $# -eq 2 ] && break    # If 2 arguments, then done.
271             fi
272         done
273     done
274     eval $2=\( \$\{_ee_Target\[@\]\} \)
275     set +f
276     return 0
277 }
278
279 # This function described in edit_by_glob.bash.
280 # edit_by_glob <excludes_array_name> <target_array_name>
281 edit_by_glob() {
282     [ $# -eq 2 ] ||
283     [ $# -eq 3 ] || return 1
284     local -a _ebg_Excludes
285     local -a _ebg_Target
286     local _ebg_x
287     local _ebg_t
288     local IFS=${NO_WSP}
289     set -f
290     eval _ebg_Excludes=\( \$\{$1\[@\]\} \)
291     eval _ebg_Target=\( \$\{$2\[@\]\} \)
292     local _ebg_len=${#_ebg_Target[@]}
293     local _ebg_cnt=${#_ebg_Excludes[@]}
294     [ ${_ebg_len} -ne 0 ] || return 0
295     [ ${_ebg_cnt} -ne 0 ] || return 0
296     for (( x = 0; x < ${_ebg_cnt} ; x++ ))
297     do
298         _ebg_x=${_ebg_Excludes[$x]}
299         for (( n = 0 ; n < ${_ebg_len} ; n++ ))
300         do
301             [ $# -eq 3 ] && _ebg_x=${_ebg_x}'*'  #  Do prefix edit
302             if [ ${_ebg_Target[$n]:=} ]          #+ if defined & set.
303             then
304                 _ebg_t=${_ebg_Target[$n]/#${_ebg_x}/}
305                 [ ${#_ebg_t} -eq 0 ] && unset _ebg_Target[$n]
306             fi
307         done
308     done
309     eval $2=\( \$\{_ebg_Target\[@\]\} \)
```

```
    set +f
    return 0
}

# This function described in unique_lines.bash.
# unique_lines <in_name> <out_name>
unique_lines() {
    [ $# -eq 2 ] || return 1
    local -a _ul_in
    local -a _ul_out
    local -i _ul_cnt
    local -i _ul_pos
    local _ul_tmp
    local IFS=${NO_WSP}
    set -f
    eval _ul_in=\( \$\{$1\[@\]\} \)
    _ul_cnt=${#_ul_in[@]}
    for (( _ul_pos = 0 ; _ul_pos < ${_ul_cnt} ; _ul_pos++ ))
    do
        if [ ${_ul_in[${_ul_pos}]:=} ]      # If defined & not empty
        then
            _ul_tmp=${_ul_in[${_ul_pos}]}
            _ul_out[${#_ul_out[@]}]=${_ul_tmp}
            for (( zap = _ul_pos ; zap < ${_ul_cnt} ; zap++ ))
            do
                [ ${_ul_in[${zap}]:=} ] &&
                [ 'x'${_ul_in[${zap}]} == 'x'${_ul_tmp} ] &&
                    unset _ul_in[${zap}]
            done
        fi
    done
    eval $2=\( \$\{_ul_out\[@\]\} \)
    set +f
    return 0
}

# This function described in char_convert.bash.
# to_lower <string>
to_lower() {
    [ $# -eq 1 ] || return 1
    local _tl_out
    _tl_out=${1//A/a}
    _tl_out=${_tl_out//B/b}
    _tl_out=${_tl_out//C/c}
    _tl_out=${_tl_out//D/d}
    _tl_out=${_tl_out//E/e}
    _tl_out=${_tl_out//F/f}
    _tl_out=${_tl_out//G/g}
    _tl_out=${_tl_out//H/h}
    _tl_out=${_tl_out//I/i}
    _tl_out=${_tl_out//J/j}
    _tl_out=${_tl_out//K/k}
    _tl_out=${_tl_out//L/l}
    _tl_out=${_tl_out//M/m}
    _tl_out=${_tl_out//N/n}
    _tl_out=${_tl_out//O/o}
    _tl_out=${_tl_out//P/p}
    _tl_out=${_tl_out//Q/q}
    _tl_out=${_tl_out//R/r}
    _tl_out=${_tl_out//S/s}
    _tl_out=${_tl_out//T/t}
    _tl_out=${_tl_out//U/u}
```

```
372     _tl_out=${_tl_out//V/v}
373     _tl_out=${_tl_out//W/w}
374     _tl_out=${_tl_out//X/x}
375     _tl_out=${_tl_out//Y/y}
376     _tl_out=${_tl_out//Z/z}
377     echo ${_tl_out}
378     return 0
379 }
380
381 #### Application helper functions ####
382
383 # Not everybody uses dots as separators (APNIC, for example).
384 # This function described in to_dot.bash
385 # to_dot <string>
386 to_dot() {
387     [ $# -eq 1 ] || return 1
388     echo ${1//[#|@|%]/.}
389     return 0
390 }
391
392 # This function described in is_number.bash.
393 # is_number <input>
394 is_number() {
395     [ "$#" -eq 1 ]    || return 1  # is blank?
396     [ x"$1" == 'x0' ] && return 0  # is zero?
397     local -i tst
398     let tst=$1 2>/dev/null         # else is numeric!
399     return $?
400 }
401
402 # This function described in is_address.bash.
403 # is_address <input>
404 is_address() {
405     [ $# -eq 1 ] || return 1    # Blank ==> false
406     local -a _ia_input
407     local IFS=${ADR_IFS}
408     _ia_input=( $1 )
409     if  [ ${#_ia_input[@]} -eq 4 ]  &&
410         is_number ${_ia_input[0]}   &&
411         is_number ${_ia_input[1]}   &&
412         is_number ${_ia_input[2]}   &&
413         is_number ${_ia_input[3]}   &&
414         [ ${_ia_input[0]} -lt 256 ] &&
415         [ ${_ia_input[1]} -lt 256 ] &&
416         [ ${_ia_input[2]} -lt 256 ] &&
417         [ ${_ia_input[3]} -lt 256 ]
418     then
419         return 0
420     else
421         return 1
422     fi
423 }
424
425 # This function described in split_ip.bash.
426 # split_ip <IP_address> <array_name_norm> [<array_name_rev>]
427 split_ip() {
428     [ $# -eq 3 ] ||              #  Either three
429     [ $# -eq 2 ] || return 1     #+ or two arguments
430     local -a _si_input
431     local IFS=${ADR_IFS}
432     _si_input=( $1 )
433     IFS=${WSP_IFS}
```

```
434     eval $2=\(\ \$\{_si_input\[@\]\}\ \)
435     if [ $# -eq 3 ]
436     then
437         # Build query order array.
438         local -a _dns_ip
439         _dns_ip[0]=${_si_input[3]}
440         _dns_ip[1]=${_si_input[2]}
441         _dns_ip[2]=${_si_input[1]}
442         _dns_ip[3]=${_si_input[0]}
443         eval $3=\(\ \$\{_dns_ip\[@\]\}\ \)
444     fi
445     return 0
446 }
447
448 # This function described in dot_array.bash.
449 # dot_array <array_name>
450 dot_array() {
451     [ $# -eq 1 ] || return 1     # Single argument required.
452     local -a _da_input
453     eval _da_input=\(\ \$\{$1\[@\]\}\ \)
454     local IFS=${DOT_IFS}
455     local _da_output=${_da_input[@]}
456     IFS=${WSP_IFS}
457     echo ${_da_output}
458     return 0
459 }
460
461 # This function described in file_to_array.bash
462 # file_to_array <file_name> <line_array_name>
463 file_to_array() {
464     [ $# -eq 2 ] || return 1  # Two arguments required.
465     local IFS=${NO_WSP}
466     local -a _fta_tmp_
467     _fta_tmp_=( $(cat $1) )
468     eval $2=\( \$\{_fta_tmp_\[@\]\} \)
469     return 0
470 }
471
472 # Columnized print of an array of multi-field strings.
473 # col_print <array_name> <min_space> <tab_stop [tab_stops]>
474 col_print() {
475     [ $# -gt 2 ] || return 0
476     local -a _cp_inp
477     local -a _cp_spc
478     local -a _cp_line
479     local _cp_min
480     local _cp_mcnt
481     local _cp_pos
482     local _cp_cnt
483     local _cp_tab
484     local -i _cp
485     local -i _cpf
486     local _cp_fld
487     # WARNING: FOLLOWING LINE NOT BLANK -- IT IS QUOTED SPACES.
488     local _cp_max='
'
489     set -f
490     local IFS=${NO_WSP}
491     eval _cp_inp=\(\ \$\{$1\[@\]\}\ \)
492     [ ${#_cp_inp[@]} -gt 0 ] || return 0 # Empty is easy.
493     _cp_mcnt=$2
494     _cp_min=${_cp_max:1:${_cp_mcnt}}
```

```
495     shift
496     shift
497     _cp_cnt=$#
498     for (( _cp = 0 ; _cp < _cp_cnt ; _cp++ ))
499     do
500         _cp_spc[${#_cp_spc[@]}]="${_cp_max:2:$1}" #"
501         shift
502     done
503     _cp_cnt=${#_cp_inp[@]}
504     for (( _cp = 0 ; _cp < _cp_cnt ; _cp++ ))
505     do
506         _cp_pos=1
507         IFS=${NO_WSP}$'\x20'
508         _cp_line=( ${_cp_inp[${_cp}]} )
509         IFS=${NO_WSP}
510         for (( _cpf = 0 ; _cpf < ${#_cp_line[@]} ; _cpf++ ))
511         do
512             _cp_tab=${_cp_spc[${_cpf}]:${_cp_pos}}
513             if [ ${#_cp_tab} -lt ${_cp_mcnt} ]
514             then
515                 _cp_tab="${_cp_min}"
516             fi
517             echo -n "${_cp_tab}"
518             (( _cp_pos = ${_cp_pos} + ${#_cp_tab} ))
519             _cp_fld="${_cp_line[${_cpf}]}"
520             echo -n ${_cp_fld}
521             (( _cp_pos = ${_cp_pos} + ${#_cp_fld} ))
522         done
523         echo
524     done
525     set +f
526     return 0
527 }
528
529 # # # # 'Hunt the Spammer' data flow # # # #
530
531 # Application return code
532 declare -i _hs_RC
533
534 # Original input, from which IP addresses are removed
535 # After which, domain names to check
536 declare -a uc_name
537
538 # Original input IP addresses are moved here
539 # After which, IP addresses to check
540 declare -a uc_address
541
542 # Names against which address expansion run
543 # Ready for name detail lookup
544 declare -a chk_name
545
546 # Addresses against which name expansion run
547 # Ready for address detail lookup
548 declare -a chk_address
549
550 #  Recursion is depth-first-by-name.
551 #  The expand_input_address maintains this list
552 #+ to prohibit looking up addresses twice during
553 #+ domain name recursion.
554 declare -a been_there_addr
555 been_there_addr=( '127.0.0.1' ) # Whitelist localhost
556
```

```
557 # Names which we have checked (or given up on)
558 declare -a known_name
559
560 # Addresses which we have checked (or given up on)
561 declare -a known_address
562
563 #  List of zero or more Blacklist servers to check.
564 #  Each 'known_address' will be checked against each server,
565 #+ with negative replies and failures suppressed.
566 declare -a list_server
567
568 # Indirection limit - set to zero == no limit
569 indirect=${SPAMMER_LIMIT:=2}
570
571 # # # # 'Hunt the Spammer' information output data # # # #
572
573 # Any domain name may have multiple IP addresses.
574 # Any IP address may have multiple domain names.
575 # Therefore, track unique address-name pairs.
576 declare -a known_pair
577 declare -a reverse_pair
578
579 #  In addition to the data flow variables; known_address
580 #+ known_name and list_server, the following are output to the
581 #+ external graphics interface file.
582
583 # Authority chain, parent -> SOA fields.
584 declare -a auth_chain
585
586 # Reference chain, parent name -> child name
587 declare -a ref_chain
588
589 # DNS chain - domain name -> address
590 declare -a name_address
591
592 # Name and service pairs - domain name -> service
593 declare -a name_srvc
594
595 # Name and resource pairs - domain name -> Resource Record
596 declare -a name_resource
597
598 # Parent and Child pairs - parent name -> child name
599 # This MAY NOT be the same as the ref_chain followed!
600 declare -a parent_child
601
602 # Address and Blacklist hit pairs - address->server
603 declare -a address_hits
604
605 # Dump interface file data
606 declare -f _dot_dump
607 _dot_dump=pend_dummy   # Initially a no-op
608
609 #  Data dump is enabled by setting the environment variable SPAMMER_DATA
610 #+ to the name of a writable file.
611 declare _dot_file
612
613 # Helper function for the dump-to-dot-file function
614 # dump_to_dot <array_name> <prefix>
615 dump_to_dot() {
616     local -a _dda_tmp
617     local -i _dda_cnt
618     local _dda_form='    '${2}'%04u %s\n'
```

```
619     local IFS=${NO_WSP}
620     eval _dda_tmp=\(\ \$\{$1\[@\]\}\ \)
621     _dda_cnt=${#_dda_tmp[@]}
622     if [ ${_dda_cnt} -gt 0 ]
623     then
624         for (( _dda = 0 ; _dda < _dda_cnt ; _dda++ ))
625         do
626             printf "${_dda_form}" \
627                    "${_dda}" "${_dda_tmp[${_dda}]}" >>${_dot_file}
628         done
629     fi
630 }
631
632 # Which will also set _dot_dump to this function . . .
633 dump_dot() {
634     local -i _dd_cnt
635     echo '# Data vintage: '$(date -R) >${_dot_file}
636     echo '# ABS Guide: is_spammer.bash; v2, 2004-msz' >>${_dot_file}
637     echo >>${_dot_file}
638     echo 'digraph G {' >>${_dot_file}
639
640     if [ ${#known_name[@]} -gt 0 ]
641     then
642         echo >>${_dot_file}
643         echo '# Known domain name nodes' >>${_dot_file}
644         _dd_cnt=${#known_name[@]}
645         for (( _dd = 0 ; _dd < _dd_cnt ; _dd++ ))
646         do
647             printf '    N%04u [label="%s"] ;\n' \
648                    "${_dd}" "${known_name[${_dd}]}" >>${_dot_file}
649         done
650     fi
651
652     if [ ${#known_address[@]} -gt 0 ]
653     then
654         echo >>${_dot_file}
655         echo '# Known address nodes' >>${_dot_file}
656         _dd_cnt=${#known_address[@]}
657         for (( _dd = 0 ; _dd < _dd_cnt ; _dd++ ))
658         do
659             printf '    A%04u [label="%s"] ;\n' \
660                    "${_dd}" "${known_address[${_dd}]}" >>${_dot_file}
661         done
662     fi
663
664     echo                                    >>${_dot_file}
665     echo '/*'                               >>${_dot_file}
666     echo ' * Known relationships :: User conversion to'  >>${_dot_file}
667     echo ' * graphic form by hand or program required.'  >>${_dot_file}
668     echo ' *'                               >>${_dot_file}
669
670     if [ ${#auth_chain[@]} -gt 0 ]
671     then
672         echo >>${_dot_file}
673         echo '# Authority reference edges followed and field source.'
>>${_dot_file}
674         dump_to_dot auth_chain AC
675     fi
676
677     if [ ${#ref_chain[@]} -gt 0 ]
678     then
679         echo >>${_dot_file}
```

```
680         echo '# Name reference edges followed and field source.'
>>${_dot_file}
681         dump_to_dot ref_chain RC
682     fi
683
684     if [ ${#name_address[@]} -gt 0 ]
685     then
686         echo >>${_dot_file}
687         echo '# Known name->address edges' >>${_dot_file}
688         dump_to_dot name_address NA
689     fi
690
691     if [ ${#name_srvc[@]} -gt 0 ]
692     then
693         echo >>${_dot_file}
694         echo '# Known name->service edges' >>${_dot_file}
695         dump_to_dot name_srvc NS
696     fi
697
698     if [ ${#name_resource[@]} -gt 0 ]
699     then
700         echo >>${_dot_file}
701         echo '# Known name->resource edges' >>${_dot_file}
702         dump_to_dot name_resource NR
703     fi
704
705     if [ ${#parent_child[@]} -gt 0 ]
706     then
707         echo >>${_dot_file}
708         echo '# Known parent->child edges' >>${_dot_file}
709         dump_to_dot parent_child PC
710     fi
711
712     if [ ${#list_server[@]} -gt 0 ]
713     then
714         echo >>${_dot_file}
715         echo '# Known Blacklist nodes' >>${_dot_file}
716         _dd_cnt=${#list_server[@]}
717         for (( _dd = 0 ; _dd < _dd_cnt ; _dd++ ))
718         do
719             printf '    LS%04u [label="%s"] ;\n' \
720                    "${_dd}" "${list_server[${_dd}]}" >>${_dot_file}
721         done
722     fi
723
724     unique_lines address_hits address_hits
725     if [ ${#address_hits[@]} -gt 0 ]
726     then
727         echo >>${_dot_file}
728         echo '# Known address->Blacklist_hit edges' >>${_dot_file}
729         echo '# CAUTION: dig warnings can trigger false hits.'
>>${_dot_file}
730         dump_to_dot address_hits AH
731     fi
732     echo          >>${_dot_file}
733     echo ' *'     >>${_dot_file}
734     echo ' * That is a lot of relationships. Happy graphing.' >>${_dot_file}
735     echo ' */'    >>${_dot_file}
736     echo '}'      >>${_dot_file}
737     return 0
738 }
739
```

```
# # # # 'Hunt the Spammer' execution flow # # # #

#  Execution trace is enabled by setting the
#+ environment variable SPAMMER_TRACE to the name of a writable file.
declare -a _trace_log
declare _log_file

# Function to fill the trace log
trace_logger() {
    _trace_log[${#_trace_log[@]}]=${_pend_current_}
}

# Dump trace log to file function variable.
declare -f _log_dump
_log_dump=pend_dummy   # Initially a no-op.

# Dump the trace log to a file.
dump_log() {
    local -i _dl_cnt
    _dl_cnt=${#_trace_log[@]}
    for (( _dl = 0 ; _dl < _dl_cnt ; _dl++ ))
    do
        echo ${_trace_log[${_dl}]} >> ${_log_file}
    done
    _dl_cnt=${#_pending_[@]}
    if [ ${_dl_cnt} -gt 0 ]
    then
        _dl_cnt=${_dl_cnt}-1
        echo '# # # Operations stack not empty # # #' >> ${_log_file}
        for (( _dl = ${_dl_cnt} ; _dl >= 0 ; _dl-- ))
        do
            echo ${_pending_[${_dl}]} >> ${_log_file}
        done
    fi
}

# # # Utility program 'dig' wrappers # # #
#
#  These wrappers are derived from the
#+ examples shown in dig_wrappers.bash.
#
#  The major difference is these return
#+ their results as a list in an array.
#
#  See dig_wrappers.bash for details and
#+ use that script to develop any changes.
#
# # #

# Short form answer: 'dig' parses answer.

# Forward lookup :: Name -> Address
# short_fwd <domain_name> <array_name>
short_fwd() {
    local -a _sf_reply
    local -i _sf_rc
    local -i _sf_cnt
    IFS=${NO_WSP}
echo -n '.'
# echo 'sfwd: '${1}
    _sf_reply=( $(dig +short ${1} -c in -t a 2>/dev/null) )
    _sf_rc=$?
```

```
802     if [ ${_sf_rc} -ne 0 ]
803     then
804         _trace_log[${#_trace_log[@]}]='# # # Lookup error '${_sf_rc}' on
'${1}' # # #'
805 # [ ${_sf_rc} -ne 9 ] && pend_drop
806         return ${_sf_rc}
807     else
808         # Some versions of 'dig' return warnings on stdout.
809         _sf_cnt=${#_sf_reply[@]}
810         for (( _sf = 0 ; _sf < ${_sf_cnt} ; _sf++ ))
811         do
812             [ 'x'${_sf_reply[${_sf}]:0:2} == 'x;;' ] &&
813                 unset _sf_reply[${_sf}]
814         done
815         eval $2=\( \$\{_sf_reply\[@\]\} \)
816     fi
817     return 0
818 }
819
820 # Reverse lookup :: Address -> Name
821 # short_rev <ip_address> <array_name>
822 short_rev() {
823     local -a _sr_reply
824     local -i _sr_rc
825     local -i _sr_cnt
826     IFS=${NO_WSP}
827 echo -n '.'
828 # echo 'srev: '${1}
829     _sr_reply=( $(dig +short -x ${1} 2>/dev/null) )
830     _sr_rc=$?
831     if [ ${_sr_rc} -ne 0 ]
832     then
833         _trace_log[${#_trace_log[@]}]='# # # Lookup error '${_sr_rc}' on
'${1}' # # #'
834 # [ ${_sr_rc} -ne 9 ] && pend_drop
835         return ${_sr_rc}
836     else
837         # Some versions of 'dig' return warnings on stdout.
838         _sr_cnt=${#_sr_reply[@]}
839         for (( _sr = 0 ; _sr < ${_sr_cnt} ; _sr++ ))
840         do
841             [ 'x'${_sr_reply[${_sr}]:0:2} == 'x;;' ] &&
842                 unset _sr_reply[${_sr}]
843         done
844         eval $2=\( \$\{_sr_reply\[@\]\} \)
845     fi
846     return 0
847 }
848
849 # Special format lookup used to query blacklist servers.
850 # short_text <ip_address> <array_name>
851 short_text() {
852     local -a _st_reply
853     local -i _st_rc
854     local -i _st_cnt
855     IFS=${NO_WSP}
856 # echo 'stxt: '${1}
857     _st_reply=( $(dig +short ${1} -c in -t txt 2>/dev/null) )
858     _st_rc=$?
859     if [ ${_st_rc} -ne 0 ]
860     then
861         _trace_log[${#_trace_log[@]}]='# # # Text lookup error '${_st_rc}'
```

```
on '${1}' # # #'
862 # [ ${_st_rc} -ne 9 ] && pend_drop
863         return ${_st_rc}
864     else
865         # Some versions of 'dig' return warnings on stdout.
866         _st_cnt=${#_st_reply[@]}
867         for (( _st = 0 ; _st < ${#_st_cnt} ; _st++ ))
868         do
869             [ 'x'${_st_reply[${_st}]:0:2} == 'x;;' ] &&
870                 unset _st_reply[${_st}]
871         done
872         eval $2=\( \$\{_st_reply\[@\]\} \)
873     fi
874     return 0
875 }
876
877 # The long forms, a.k.a., the parse it yourself versions
878
879 # RFC 2782   Service lookups
880 # dig +noall +nofail +answer _ldap._tcp.openldap.org -t srv
881 # _<service>._<protocol>.<domain_name>
882 # _ldap._tcp.openldap.org. 3600   IN      SRV     0 0 389 ldap.openldap.org.
883 # domain TTL Class SRV Priority Weight Port Target
884
885 # Forward lookup :: Name -> poor man's zone transfer
886 # long_fwd <domain_name> <array_name>
887 long_fwd() {
888     local -a _lf_reply
889     local -i _lf_rc
890     local -i _lf_cnt
891     IFS=${NO_WSP}
892 echo -n ':'
893 # echo 'lfwd: '${1}
894     _lf_reply=( $(
895         dig +noall +nofail +answer +authority +additional \
896             ${1} -t soa ${1} -t mx ${1} -t any 2>/dev/null) )
897     _lf_rc=$?
898     if [ ${_lf_rc} -ne 0 ]
899     then
900         _trace_log[${#_trace_log[@]}]='# # # Zone lookup error '${_lf_rc}'
on '${1}' # # #'
901 # [ ${_lf_rc} -ne 9 ] && pend_drop
902         return ${_lf_rc}
903     else
904         # Some versions of 'dig' return warnings on stdout.
905         _lf_cnt=${#_lf_reply[@]}
906         for (( _lf = 0 ; _lf < ${_lf_cnt} ; _lf++ ))
907         do
908             [ 'x'${_lf_reply[${_lf}]:0:2} == 'x;;' ] &&
909                 unset _lf_reply[${_lf}]
910         done
911         eval $2=\( \$\{_lf_reply\[@\]\} \)
912     fi
913     return 0
914 }
915 #   The reverse lookup domain name corresponding to the IPv6 address:
916 #       4321:0:1:2:3:4:567:89ab
917 #   would be (nibble, I.E: Hexdigit) reversed:
918 #
b.a.9.8.7.6.5.0.4.0.0.0.3.0.0.0.2.0.0.0.1.0.0.0.0.0.0.0.1.2.3.4.IP6.ARPA.
919
920 # Reverse lookup :: Address -> poor man's delegation chain
```

```
# long_rev <rev_ip_address> <array_name>
long_rev() {
    local -a _lr_reply
    local -i _lr_rc
    local -i _lr_cnt
    local _lr_dns
    _lr_dns=${1}'.in-addr.arpa.'
    IFS=${NO_WSP}
echo -n ':'
# echo 'lrev: '${1}
    _lr_reply=( $(
         dig +noall +nofail +answer +authority +additional \
             ${_lr_dns} -t soa ${_lr_dns} -t any 2>/dev/null) )
    _lr_rc=$?
    if [ ${_lr_rc} -ne 0 ]
    then
        _trace_log[${#_trace_log[@]}]='# # # Delegation lookup error
'${_lr_rc}' on '${1}' # # #'
# [ ${_lr_rc} -ne 9 ] && pend_drop
        return ${_lr_rc}
    else
        # Some versions of 'dig' return warnings on stdout.
        _lr_cnt=${#_lr_reply[@]}
        for (( _lr = 0 ; _lr < ${_lr_cnt} ; _lr++ ))
        do
            [ 'x'${_lr_reply[${_lr}]:0:2} == 'x;;' ] &&
                unset _lr_reply[${_lr}]
        done
        eval $2=\( \$\{_lr_reply\[@\]\} \)
    fi
    return 0
}

# # # Application specific functions # # #

# Mung a possible name; suppresses root and TLDs.
# name_fixup <string>
name_fixup(){
    local -a _nf_tmp
    local -i _nf_end
    local _nf_str
    local IFS
    _nf_str=$(to_lower ${1})
    _nf_str=$(to_dot ${_nf_str})
    _nf_end=${#_nf_str}-1
    [ ${_nf_str:${_nf_end}} != '.' ] &&
        _nf_str=${_nf_str}'.'
    IFS=${ADR_IFS}
    _nf_tmp=( ${_nf_str} )
    IFS=${WSP_IFS}
    _nf_end=${#_nf_tmp[@]}
    case ${_nf_end} in
    0) # No dots, only dots
        echo
        return 1
    ;;
    1) # Only a TLD.
        echo
        return 1
    ;;
    2) # Maybe okay.
       echo ${_nf_str}
```

```
982         return 0
983         # Needs a lookup table?
984         if [ ${#_nf_tmp[1]} -eq 2 ]
985         then # Country coded TLD.
986             echo
987             return 1
988         else
989             echo ${_nf_str}
990             return 0
991         fi
992     ;;
993     esac
994     echo ${_nf_str}
995     return 0
996 }
997
998 # Grope and mung original input(s).
999 split_input() {
1000     [ ${#uc_name[@]} -gt 0 ] || return 0
1001     local -i _si_cnt
1002     local -i _si_len
1003     local _si_str
1004     unique_lines uc_name uc_name
1005     _si_cnt=${#uc_name[@]}
1006     for (( _si = 0 ; _si < _si_cnt ; _si++ ))
1007     do
1008         _si_str=${uc_name[$_si]}
1009         if is_address ${_si_str}
1010         then
1011             uc_address[${#uc_address[@]}]=${_si_str}
1012             unset uc_name[$_si]
1013         else
1014             if ! uc_name[$_si]=$(name_fixup ${_si_str})
1015             then
1016                 unset ucname[$_si]
1017             fi
1018         fi
1019     done
1020     uc_name=( ${uc_name[@]} )
1021     _si_cnt=${#uc_name[@]}
1022     _trace_log[${#_trace_log[@]}]='# # # Input '${_si_cnt}' unchecked name
input(s). # # #'
1023     _si_cnt=${#uc_address[@]}
1024     _trace_log[${#_trace_log[@]}]='# # # Input '${_si_cnt}' unchecked
address input(s). # # #'
1025     return 0
1026 }
1027
1028 # # # Discovery functions -- recursively interlocked by external data # # #
1029 # # # The leading 'if list is empty; return 0' in each is required. # # #
1030
1031 # Recursion limiter
1032 # limit_chk() <next_level>
1033 limit_chk() {
1034     local -i _lc_lmt
1035     # Check indirection limit.
1036     if [ ${indirect} -eq 0 ] || [ $# -eq 0 ]
1037     then
1038         # The 'do-forever' choice
1039         echo 1                # Any value will do.
1040         return 0              # OK to continue.
1041     else
```

```
1042         # Limiting is in effect.
1043         if [ ${indirect} -lt ${1} ]
1044         then
1045             echo ${1}          # Whatever.
1046             return 1           # Stop here.
1047         else
1048             _lc_lmt=${1}+1     # Bump the given limit.
1049             echo ${_lc_lmt}    # Echo it.
1050             return 0           # OK to continue.
1051         fi
1052     fi
1053 }
1054
1055 # For each name in uc_name:
1056 #     Move name to chk_name.
1057 #     Add addresses to uc_address.
1058 #     Pend expand_input_address.
1059 #     Repeat until nothing new found.
1060 # expand_input_name <indirection_limit>
1061 expand_input_name() {
1062     [ ${#uc_name[@]} -gt 0 ] || return 0
1063     local -a _ein_addr
1064     local -a _ein_new
1065     local -i _ucn_cnt
1066     local -i _ein_cnt
1067     local _ein_tst
1068     _ucn_cnt=${#uc_name[@]}
1069
1070     if  ! _ein_cnt=$(limit_chk ${1})
1071     then
1072         return 0
1073     fi
1074
1075     for (( _ein = 0 ; _ein < _ucn_cnt ; _ein++ ))
1076     do
1077         if short_fwd ${uc_name[${_ein}]} _ein_new
1078         then
1079             for (( _ein_cnt = 0 ; _ein_cnt < ${#_ein_new[@]}; _ein_cnt++ ))
1080             do
1081                 _ein_tst=${_ein_new[${_ein_cnt}]}
1082                 if is_address ${_ein_tst}
1083                 then
1084                     _ein_addr[${#_ein_addr[@]}]=${_ein_tst}
1085                 fi
1086            done
1087         fi
1088     done
1089     unique_lines _ein_addr _ein_addr     # Scrub duplicates.
1090     edit_exact chk_address _ein_addr     # Scrub pending detail.
1091     edit_exact known_address _ein_addr   # Scrub already detailed.
1092     if [ ${#_ein_addr[@]} -gt 0 ]        # Anything new?
1093     then
1094         uc_address=( ${uc_address[@]} ${_ein_addr[@]} )
1095         pend_func expand_input_address ${1}
1096         _trace_log[${#_trace_log[@]}]='# # # Added '${#_ein_addr[@]}'
unchecked address input(s). # # #'
1097     fi
1098     edit_exact chk_name uc_name          # Scrub pending detail.
1099     edit_exact known_name uc_name        # Scrub already detailed.
1100     if [ ${#uc_name[@]} -gt 0 ]
1101     then
1102         chk_name=( ${chk_name[@]} ${uc_name[@]}  )
```

```
1103         pend_func detail_each_name ${1}
1104     fi
1105     unset uc_name[@]
1106     return 0
1107 }
1108
1109 # For each address in uc_address:
1110 #     Move address to chk_address.
1111 #     Add names to uc_name.
1112 #     Pend expand_input_name.
1113 #     Repeat until nothing new found.
1114 # expand_input_address <indirection_limit>
1115 expand_input_address() {
1116     [ ${#uc_address[@]} -gt 0 ] || return 0
1117     local -a _eia_addr
1118     local -a _eia_name
1119     local -a _eia_new
1120     local -i _uca_cnt
1121     local -i _eia_cnt
1122     local _eia_tst
1123     unique_lines uc_address _eia_addr
1124     unset uc_address[@]
1125     edit_exact been_there_addr _eia_addr
1126     _uca_cnt=${#_eia_addr[@]}
1127     [ ${_uca_cnt} -gt 0 ] &&
1128         been_there_addr=( ${been_there_addr[@]} ${_eia_addr[@]} )
1129
1130     for (( _eia = 0 ; _eia < _uca_cnt ; _eia++ ))
1131     do
1132             if short_rev ${_eia_addr[${_eia}]} _eia_new
1133             then
1134                 for (( _eia_cnt = 0 ; _eia_cnt < ${#_eia_new[@]} ;
_eia_cnt++ ))
1135                 do
1136                     _eia_tst=${_eia_new[${_eia_cnt}]}
1137                     if _eia_tst=$(name_fixup ${_eia_tst})
1138                     then
1139                         _eia_name[${#_eia_name[@]}]=${_eia_tst}
1140                     fi
1141                 done
1142             fi
1143     done
1144     unique_lines _eia_name _eia_name     # Scrub duplicates.
1145     edit_exact chk_name _eia_name        # Scrub pending detail.
1146     edit_exact known_name _eia_name      # Scrub already detailed.
1147     if [ ${#_eia_name[@]} -gt 0 ]        # Anything new?
1148     then
1149         uc_name=( ${uc_name[@]} ${_eia_name[@]} )
1150         pend_func expand_input_name ${1}
1151         _trace_log[${#_trace_log[@]}]='# # # Added '${#_eia_name[@]}'
unchecked name input(s). # # #'
1152     fi
1153     edit_exact chk_address _eia_addr     # Scrub pending detail.
1154     edit_exact known_address _eia_addr   # Scrub already detailed.
1155     if [ ${#_eia_addr[@]} -gt 0 ]        # Anything new?
1156     then
1157         chk_address=( ${chk_address[@]} ${_eia_addr[@]} )
1158         pend_func detail_each_address ${1}
1159     fi
1160     return 0
1161 }
1162
```

```
1163 # The parse-it-yourself zone reply.
1164 # The input is the chk_name list.
1165 # detail_each_name <indirection_limit>
1166 detail_each_name() {
1167     [ ${#chk_name[@]} -gt 0 ] || return 0
1168     local -a _den_chk       # Names to check
1169     local -a _den_name      # Names found here
1170     local -a _den_address   # Addresses found here
1171     local -a _den_pair      # Pairs found here
1172     local -a _den_rev       # Reverse pairs found here
1173     local -a _den_tmp       # Line being parsed
1174     local -a _den_auth      # SOA contact being parsed
1175     local -a _den_new       # The zone reply
1176     local -a _den_pc        # Parent-Child gets big fast
1177     local -a _den_ref       # So does reference chain
1178     local -a _den_nr        # Name-Resource can be big
1179     local -a _den_na        # Name-Address
1180     local -a _den_ns        # Name-Service
1181     local -a _den_achn      # Chain of Authority
1182     local -i _den_cnt       # Count of names to detail
1183     local -i _den_lmt       # Indirection limit
1184     local _den_who          # Named being processed
1185     local _den_rec          # Record type being processed
1186     local _den_cont         # Contact domain
1187     local _den_str          # Fixed up name string
1188     local _den_str2         # Fixed up reverse
1189     local IFS=${WSP_IFS}
1190
1191     # Local, unique copy of names to check
1192     unique_lines chk_name _den_chk
1193     unset chk_name[@]       # Done with globals.
1194
1195     # Less any names already known
1196     edit_exact known_name _den_chk
1197     _den_cnt=${#_den_chk[@]}
1198
1199     # If anything left, add to known_name.
1200     [ ${_den_cnt} -gt 0 ] &&
1201         known_name=( ${known_name[@]} ${_den_chk[@]} )
1202
1203     # for the list of (previously) unknown names . . .
1204     for (( _den = 0 ; _den < _den_cnt ; _den++ ))
1205     do
1206         _den_who=${_den_chk[${_den}]}
1207         if long_fwd ${_den_who} _den_new
1208         then
1209             unique_lines _den_new _den_new
1210             if [ ${#_den_new[@]} -eq 0 ]
1211             then
1212                 _den_pair[${#_den_pair[@]}]='0.0.0.0 '${_den_who}
1213             fi
1214
1215             # Parse each line in the reply.
1216             for (( _line = 0 ; _line < ${#_den_new[@]} ; _line++ ))
1217             do
1218                 IFS=${NO_WSP}$'\x09'$'\x20'
1219                 _den_tmp=( ${_den_new[${_line}]} )
1220                 IFS=${WSP_IFS}
1221                 # If usable record and not a warning message . . .
1222                 if [ ${#_den_tmp[@]} -gt 4 ] && [ 'x'${_den_tmp[0]} !=
'x;;' ]
1223                 then
```

```
1224                    _den_rec=${_den_tmp[3]}
1225                    _den_nr[${#_den_nr[@]}]=${_den_who}' '${_den_rec}
1226                    # Begin at RFC1033 (+++)
1227                    case ${_den_rec} in
1228
1229                         #<name>  [<ttl>]  [<class>]  SOA  <origin>
<person>
1230                    SOA) # Start Of Authority
1231                        if _den_str=$(name_fixup ${_den_tmp[0]})
1232                        then
1233                            _den_name[${#_den_name[@]}]=${_den_str}
1234                            _den_achn[${#_den_achn[@]}]=${_den_who}'
'${_den_str}' SOA'
1235                            # SOA origin -- domain name of master zone
record
1236                            if _den_str2=$(name_fixup ${_den_tmp[4]})
1237                            then
1238                                _den_name[${#_den_name[@]}]=${_den_str2}
1239                                _den_achn[${#_den_achn[@]}]=${_den_who}'
'${_den_str2}' SOA.O'
1240                            fi
1241                            # Responsible party e-mail address (possibly
bogus).
1242                            # Possibility of first.last@domain.name
ignored.
1243                            set -f
1244                            if _den_str2=$(name_fixup ${_den_tmp[5]})
1245                            then
1246                                IFS=${ADR_IFS}
1247                                _den_auth=( ${_den_str2} )
1248                                IFS=${WSP_IFS}
1249                                if [ ${#_den_auth[@]} -gt 2 ]
1250                                then
1251                                     _den_cont=${_den_auth[1]}
1252                                     for (( _auth = 2 ; _auth <
${#_den_auth[@]} ; _auth++ ))
1253                                     do
1254
_den_cont=${_den_cont}'.'${_den_auth[${_auth}]}
1255                                     done
1256
_den_name[${#_den_name[@]}]=${_den_cont}'.'
1257
_den_achn[${#_den_achn[@]}]=${_den_who}' '${_den_cont}'. SOA.C'
1258                                fi
1259                            fi
1260                            set +f
1261                        fi
1262                    ;;
1263
1264
1265                    A) # IP(v4) Address Record
1266                        if _den_str=$(name_fixup ${_den_tmp[0]})
1267                        then
1268                            _den_name[${#_den_name[@]}]=${_den_str}
1269                            _den_pair[${#_den_pair[@]}]=${_den_tmp[4]}'
'${_den_str}
1270                            _den_na[${#_den_na[@]}]=${_den_str}'
'${_den_tmp[4]}
1271                            _den_ref[${#_den_ref[@]}]=${_den_who}'
'${_den_str}' A'
1272                        else
```

```
1273                            _den_pair[${#_den_pair[@]}]=${_den_tmp[4]}'
unknown.domain'
1274                            _den_na[${#_den_na[@]}]='unknown.domain
'${_den_tmp[4]}
1275                            _den_ref[${#_den_ref[@]}]=${_den_who}'
unknown.domain A'
1276                        fi
1277                        _den_address[${#_den_address[@]}]=${_den_tmp[4]}
1278                        _den_pc[${#_den_pc[@]}]=${_den_who}'
'${_den_tmp[4]}
1279                    ;;
1280
1281                    NS) # Name Server Record
1282                        # Domain name being serviced (may be other than
current)
1283                        if _den_str=$(name_fixup ${_den_tmp[0]})
1284                        then
1285                            _den_name[${#_den_name[@]}]=${_den_str}
1286                            _den_ref[${#_den_ref[@]}]=${_den_who}'
'${_den_str}' NS'
1287
1288                            # Domain name of service provider
1289                            if _den_str2=$(name_fixup ${_den_tmp[4]})
1290                            then
1291                                _den_name[${#_den_name[@]}]=${_den_str2}
1292                                _den_ref[${#_den_ref[@]}]=${_den_who}'
'${_den_str2}' NSH'
1293                                _den_ns[${#_den_ns[@]}]=${_den_str2}' NS'
1294                                _den_pc[${#_den_pc[@]}]=${_den_str}'
'${_den_str2}
1295                            fi
1296                        fi
1297                    ;;
1298
1299                    MX) # Mail Server Record
1300                        # Domain name being serviced (wildcards not handled
here)
1301                        if _den_str=$(name_fixup ${_den_tmp[0]})
1302                        then
1303                            _den_name[${#_den_name[@]}]=${_den_str}
1304                            _den_ref[${#_den_ref[@]}]=${_den_who}'
'${_den_str}' MX'
1305                        fi
1306                        # Domain name of service provider
1307                        if _den_str=$(name_fixup ${_den_tmp[5]})
1308                        then
1309                            _den_name[${#_den_name[@]}]=${_den_str}
1310                            _den_ref[${#_den_ref[@]}]=${_den_who}'
'${_den_str}' MXH'
1311                            _den_ns[${#_den_ns[@]}]=${_den_str}' MX'
1312                            _den_pc[${#_den_pc[@]}]=${_den_who}'
'${_den_str}
1313                        fi
1314                    ;;
1315
1316                    PTR) # Reverse address record
1317                         # Special name
1318                        if _den_str=$(name_fixup ${_den_tmp[0]})
1319                        then
1320                            _den_ref[${#_den_ref[@]}]=${_den_who}'
'${_den_str}' PTR'
1321                            # Host name (not a CNAME)
```

```
1322                            if _den_str2=$(name_fixup ${_den_tmp[4]})
1323                            then
1324                                _den_rev[${#_den_rev[@]}]=${_den_str}'
'${_den_str2}
1325                                _den_ref[${#_den_ref[@]}]=${_den_who}'
'${_den_str2}' PTRH'
1326                                _den_pc[${#_den_pc[@]}]=${_den_who}'
'${_den_str}
1327                            fi
1328                        fi
1329                    ;;
1330
1331                    AAAA) # IP(v6) Address Record
1332                        if _den_str=$(name_fixup ${_den_tmp[0]})
1333                        then
1334                            _den_name[${#_den_name[@]}]=${_den_str}
1335                            _den_pair[${#_den_pair[@]}]=${_den_tmp[4]}'
'${_den_str}
1336                            _den_na[${#_den_na[@]}]=${_den_str}'
'${_den_tmp[4]}
1337                            _den_ref[${#_den_ref[@]}]=${_den_who}'
'${_den_str}' AAAA'
1338                        else
1339                            _den_pair[${#_den_pair[@]}]=${_den_tmp[4]}'
unknown.domain'
1340                            _den_na[${#_den_na[@]}]='unknown.domain
'${_den_tmp[4]}
1341                            _den_ref[${#_den_ref[@]}]=${_den_who}'
unknown.domain'
1342                        fi
1343                        # No processing for IPv6 addresses
1344                            _den_pc[${#_den_pc[@]}]=${_den_who}'
'${_den_tmp[4]}
1345                    ;;
1346
1347                    CNAME) # Alias name record
1348                           # Nickname
1349                        if _den_str=$(name_fixup ${_den_tmp[0]})
1350                        then
1351                            _den_name[${#_den_name[@]}]=${_den_str}
1352                            _den_ref[${#_den_ref[@]}]=${_den_who}'
'${_den_str}' CNAME'
1353                            _den_pc[${#_den_pc[@]}]=${_den_who}'
'${_den_str}
1354                        fi
1355                        # Hostname
1356                        if _den_str=$(name_fixup ${_den_tmp[4]})
1357                        then
1358                            _den_name[${#_den_name[@]}]=${_den_str}
1359                            _den_ref[${#_den_ref[@]}]=${_den_who}'
'${_den_str}' CHOST'
1360                            _den_pc[${#_den_pc[@]}]=${_den_who}'
'${_den_str}
1361                        fi
1362                    ;;
1363 #                  TXT)
1364 #                  ;;
1365                    esac
1366                fi
1367            done
1368        else # Lookup error == 'A' record 'unknown address'
1369            _den_pair[${#_den_pair[@]}]='0.0.0.0 '${_den_who}
```

```
1370        fi
1371    done
1372
1373    # Control dot array growth.
1374    unique_lines _den_achn _den_achn      # Works best, all the same.
1375    edit_exact auth_chain _den_achn       # Works best, unique items.
1376    if [ ${#_den_achn[@]} -gt 0 ]
1377    then
1378        IFS=${NO_WSP}
1379        auth_chain=( ${auth_chain[@]} ${_den_achn[@]} )
1380        IFS=${WSP_IFS}
1381    fi
1382
1383    unique_lines _den_ref _den_ref      # Works best, all the same.
1384    edit_exact ref_chain _den_ref       # Works best, unique items.
1385    if [ ${#_den_ref[@]} -gt 0 ]
1386    then
1387        IFS=${NO_WSP}
1388        ref_chain=( ${ref_chain[@]} ${_den_ref[@]} )
1389        IFS=${WSP_IFS}
1390    fi
1391
1392    unique_lines _den_na _den_na
1393    edit_exact name_address _den_na
1394    if [ ${#_den_na[@]} -gt 0 ]
1395    then
1396        IFS=${NO_WSP}
1397        name_address=( ${name_address[@]} ${_den_na[@]} )
1398        IFS=${WSP_IFS}
1399    fi
1400
1401    unique_lines _den_ns _den_ns
1402    edit_exact name_srvc _den_ns
1403    if [ ${#_den_ns[@]} -gt 0 ]
1404    then
1405        IFS=${NO_WSP}
1406        name_srvc=( ${name_srvc[@]} ${_den_ns[@]} )
1407        IFS=${WSP_IFS}
1408    fi
1409
1410    unique_lines _den_nr _den_nr
1411    edit_exact name_resource _den_nr
1412    if [ ${#_den_nr[@]} -gt 0 ]
1413    then
1414        IFS=${NO_WSP}
1415        name_resource=( ${name_resource[@]} ${_den_nr[@]} )
1416        IFS=${WSP_IFS}
1417    fi
1418
1419    unique_lines _den_pc _den_pc
1420    edit_exact parent_child _den_pc
1421    if [ ${#_den_pc[@]} -gt 0 ]
1422    then
1423        IFS=${NO_WSP}
1424        parent_child=( ${parent_child[@]} ${_den_pc[@]} )
1425        IFS=${WSP_IFS}
1426    fi
1427
1428    # Update list known_pair (Address and Name).
1429    unique_lines _den_pair _den_pair
1430    edit_exact known_pair _den_pair
1431    if [ ${#_den_pair[@]} -gt 0 ]  # Anything new?
```

```
1432     then
1433         IFS=${NO_WSP}
1434         known_pair=( ${known_pair[@]} ${_den_pair[@]} )
1435         IFS=${WSP_IFS}
1436     fi
1437
1438     # Update list of reverse pairs.
1439     unique_lines _den_rev _den_rev
1440     edit_exact reverse_pair _den_rev
1441     if [ ${#_den_rev[@]} -gt 0 ]   # Anything new?
1442     then
1443         IFS=${NO_WSP}
1444         reverse_pair=( ${reverse_pair[@]} ${_den_rev[@]} )
1445         IFS=${WSP_IFS}
1446     fi
1447
1448     # Check indirection limit -- give up if reached.
1449     if ! _den_lmt=$(limit_chk ${1})
1450     then
1451         return 0
1452     fi
1453
1454     # Execution engine is LIFO. Order of pend operations is important.
1455     # Did we define any new addresses?
1456     unique_lines _den_address _den_address    # Scrub duplicates.
1457     edit_exact known_address _den_address     # Scrub already processed.
1458     edit_exact un_address _den_address        # Scrub already waiting.
1459     if [ ${#_den_address[@]} -gt 0 ]          # Anything new?
1460     then
1461         uc_address=( ${uc_address[@]} ${_den_address[@]} )
1462         pend_func expand_input_address ${_den_lmt}
1463         _trace_log[${#_trace_log[@]}]='# # # Added '${#_den_address[@]}'
unchecked address(s). # # #'
1464     fi
1465
1466     # Did we find any new names?
1467     unique_lines _den_name _den_name          # Scrub duplicates.
1468     edit_exact known_name _den_name           # Scrub already processed.
1469     edit_exact uc_name _den_name              # Scrub already waiting.
1470     if [ ${#_den_name[@]} -gt 0 ]             # Anything new?
1471     then
1472         uc_name=( ${uc_name[@]} ${_den_name[@]} )
1473         pend_func expand_input_name ${_den_lmt}
1474         _trace_log[${#_trace_log[@]}]='# # # Added '${#_den_name[@]}'
unchecked name(s). # # #'
1475     fi
1476     return 0
1477 }
1478
1479 # The parse-it-yourself delegation reply
1480 # Input is the chk_address list.
1481 # detail_each_address <indirection_limit>
1482 detail_each_address() {
1483     [ ${#chk_address[@]} -gt 0 ] || return 0
1484     unique_lines chk_address chk_address
1485     edit_exact known_address chk_address
1486     if [ ${#chk_address[@]} -gt 0 ]
1487     then
1488         known_address=( ${known_address[@]} ${chk_address[@]} )
1489         unset chk_address[@]
1490     fi
1491     return 0
```

```
1492 }
1493
1494 # # # Application specific output functions # # #
1495
1496 # Pretty print the known pairs.
1497 report_pairs() {
1498     echo
1499     echo 'Known network pairs.'
1500     col_print known_pair 2 5 30
1501
1502     if [ ${#auth_chain[@]} -gt 0 ]
1503     then
1504         echo
1505         echo 'Known chain of authority.'
1506         col_print auth_chain 2 5 30 55
1507     fi
1508
1509     if [ ${#reverse_pair[@]} -gt 0 ]
1510     then
1511         echo
1512         echo 'Known reverse pairs.'
1513         col_print reverse_pair 2 5 55
1514     fi
1515     return 0
1516 }
1517
1518 # Check an address against the list of blacklist servers.
1519 # A good place to capture for GraphViz: address->status(server(reports))
1520 # check_lists <ip_address>
1521 check_lists() {
1522     [ $# -eq 1 ] || return 1
1523     local -a _cl_fwd_addr
1524     local -a _cl_rev_addr
1525     local -a _cl_reply
1526     local -i _cl_rc
1527     local -i _ls_cnt
1528     local _cl_dns_addr
1529     local _cl_lkup
1530
1531     split_ip ${1} _cl_fwd_addr _cl_rev_addr
1532     _cl_dns_addr=$(dot_array _cl_rev_addr)'.'
1533     _ls_cnt=${#list_server[@]}
1534     echo '    Checking address '${1}
1535     for (( _cl = 0 ; _cl < _ls_cnt ; _cl++ ))
1536     do
1537         _cl_lkup=${_cl_dns_addr}${list_server[${_cl}]}
1538         if short_text ${_cl_lkup} _cl_reply
1539         then
1540             if [ ${#_cl_reply[@]} -gt 0 ]
1541             then
1542                 echo '        Records from '${list_server[${_cl}]}
1543                 address_hits[${#address_hits[@]}]=${1}'
'${list_server[${_cl}]}
1544                 _hs_RC=2
1545                 for (( _clr = 0 ; _clr < ${#_cl_reply[@]} ; _clr++ ))
1546                 do
1547                     echo '            '${_cl_reply[${_clr}]}
1548                 done
1549             fi
1550         fi
1551     done
1552     return 0
```

```
1553 }
1554
1555 # # # The usual application glue # # #
1556
1557 # Who did it?
1558 credits() {
1559    echo
1560    echo 'Advanced Bash Scripting Guide: is_spammer.bash, v2, 2004-msz'
1561 }
1562
1563 # How to use it?
1564 # (See also, "Quickstart" at end of script.)
1565 usage() {
1566     cat <<-'_usage_statement_'
1567     The script is_spammer.bash requires either one or two arguments.
1568
1569     arg 1) May be one of:
1570         a) A domain name
1571         b) An IPv4 address
1572         c) The name of a file with any mix of names
1573            and addresses, one per line.
1574
1575     arg 2) May be one of:
1576         a) A Blacklist server domain name
1577         b) The name of a file with Blacklist server
1578            domain names, one per line.
1579         c) If not present, a default list of (free)
1580            Blacklist servers is used.
1581         d) If a filename of an empty, readable, file
1582            is given,
1583            Blacklist server lookup is disabled.
1584
1585     All script output is written to stdout.
1586
1587     Return codes: 0 -> All OK, 1 -> Script failure,
1588                   2 -> Something is Blacklisted.
1589
1590     Requires the external program 'dig' from the 'bind-9'
1591     set of DNS programs.  See: http://www.isc.org
1592
1593     The domain name lookup depth limit defaults to 2 levels.
1594     Set the environment variable SPAMMER_LIMIT to change.
1595     SPAMMER_LIMIT=0 means 'unlimited'
1596
1597     Limit may also be set on the command line.
1598     If arg#1 is an integer, the limit is set to that value
1599     and then the above argument rules are applied.
1600
1601     Setting the environment variable 'SPAMMER_DATA' to a filename
1602     will cause the script to write a GraphViz graphic file.
1603
1604     For the development version;
1605     Setting the environment variable 'SPAMMER_TRACE' to a filename
1606     will cause the execution engine to log a function call trace.
1607
1608 _usage_statement_
1609 }
1610
1611 # The default list of Blacklist servers:
1612 # Many choices, see: http://www.spews.org/lists.html
1613
1614 declare -a default_servers
```

```
# See: http://www.spamhaus.org (Conservative, well maintained)
default_servers[0]='sbl-xbl.spamhaus.org'
# See: http://ordb.org (Open mail relays)
default_servers[1]='relays.ordb.org'
# See: http://www.spamcop.net/ (You can report spammers here)
default_servers[2]='bl.spamcop.net'
# See: http://www.spews.org (An 'early detect' system)
default_servers[3]='l2.spews.dnsbl.sorbs.net'
# See: http://www.dnsbl.us.sorbs.net/using.shtml
default_servers[4]='dnsbl.sorbs.net'
# See: http://dsbl.org/usage (Various mail relay lists)
default_servers[5]='list.dsbl.org'
default_servers[6]='multihop.dsbl.org'
default_servers[7]='unconfirmed.dsbl.org'

# User input argument #1
setup_input() {
    if [ -e ${1} ] && [ -r ${1} ]  # Name of readable file
    then
        file_to_array ${1} uc_name
        echo 'Using filename >'${1}'< as input.'
    else
        if is_address ${1}          # IP address?
        then
            uc_address=( ${1} )
            echo 'Starting with address >'${1}'<'
        else                        # Must be a name.
            uc_name=( ${1} )
            echo 'Starting with domain name >'${1}'<'
        fi
    fi
    return 0
}

# User input argument #2
setup_servers() {
    if [ -e ${1} ] && [ -r ${1} ]  # Name of a readable file
    then
        file_to_array ${1} list_server
        echo 'Using filename >'${1}'< as blacklist server list.'
    else
        list_server=( ${1} )
        echo 'Using blacklist server >'${1}'<'
    fi
    return 0
}

# User environment variable SPAMMER_TRACE
live_log_die() {
    if [ ${SPAMMER_TRACE:=} ]    # Wants trace log?
    then
        if [ ! -e ${SPAMMER_TRACE} ]
        then
            if ! touch ${SPAMMER_TRACE} 2>/dev/null
            then
                pend_func echo $(printf '%q\n' \
                'Unable to create log file >'${SPAMMER_TRACE}'<')
                pend_release
                exit 1
            fi
            _log_file=${SPAMMER_TRACE}
            _pend_hook_=trace_logger
```

```
1677             _log_dump=dump_log
1678         else
1679             if [ ! -w ${SPAMMER_TRACE} ]
1680             then
1681                 pend_func echo $(printf '%q\n' \
1682                 'Unable to write log file >'${SPAMMER_TRACE}'<')
1683                 pend_release
1684                 exit 1
1685             fi
1686             _log_file=${SPAMMER_TRACE}
1687             echo '' > ${_log_file}
1688             _pend_hook_=trace_logger
1689             _log_dump=dump_log
1690         fi
1691     fi
1692     return 0
1693 }
1694
1695 # User environment variable SPAMMER_DATA
1696 data_capture() {
1697     if [ ${SPAMMER_DATA:=} ]    # Wants a data dump?
1698     then
1699         if [ ! -e ${SPAMMER_DATA} ]
1700         then
1701             if ! touch ${SPAMMER_DATA} 2>/dev/null
1702             then
1703                 pend_func echo $(printf '%q]n' \
1704                 'Unable to create data output file >'${SPAMMER_DATA}'<')
1705                 pend_release
1706                 exit 1
1707             fi
1708             _dot_file=${SPAMMER_DATA}
1709             _dot_dump=dump_dot
1710         else
1711             if [ ! -w ${SPAMMER_DATA} ]
1712             then
1713                 pend_func echo $(printf '%q\n' \
1714                 'Unable to write data output file >'${SPAMMER_DATA}'<')
1715                 pend_release
1716                 exit 1
1717             fi
1718             _dot_file=${SPAMMER_DATA}
1719             _dot_dump=dump_dot
1720         fi
1721     fi
1722     return 0
1723 }
1724
1725 # Grope user specified arguments.
1726 do_user_args() {
1727     if [ $# -gt 0 ] && is_number $1
1728     then
1729         indirect=$1
1730         shift
1731     fi
1732
1733     case $# in                      # Did user treat us well?
1734         1)
1735             if ! setup_input $1    # Needs error checking.
1736             then
1737                 pend_release
1738                 $_log_dump
```

```
1739                exit 1
1740            fi
1741            list_server=( ${default_servers[@]} )
1742            _list_cnt=${#list_server[@]}
1743            echo 'Using default blacklist server list.'
1744            echo 'Search depth limit: '${indirect}
1745            ;;
1746        2)
1747            if ! setup_input $1    # Needs error checking.
1748            then
1749                pend_release
1750                $_log_dump
1751                exit 1
1752            fi
1753            if ! setup_servers $2  # Needs error checking.
1754            then
1755                pend_release
1756                $_log_dump
1757                exit 1
1758            fi
1759            echo 'Search depth limit: '${indirect}
1760            ;;
1761        *)
1762            pend_func usage
1763            pend_release
1764            $_log_dump
1765            exit 1
1766            ;;
1767    esac
1768    return 0
1769 }
1770
1771 # A general purpose debug tool.
1772 # list_array <array_name>
1773 list_array() {
1774    [ $# -eq 1 ] || return 1  # One argument required.
1775
1776    local -a _la_lines
1777    set -f
1778    local IFS=${NO_WSP}
1779    eval _la_lines=\(\ \$\{$1\[@\]\}\ \)
1780    echo
1781    echo "Element count "${#_la_lines[@]}" array "${1}
1782    local _ln_cnt=${#_la_lines[@]}
1783
1784    for (( _i = 0; _i < ${_ln_cnt}; _i++ ))
1785    do
1786        echo 'Element '$_i' >'${_la_lines[$_i]}'<'
1787    done
1788    set +f
1789    return 0
1790 }
1791
1792 # # # 'Hunt the Spammer' program code # # #
1793 pend_init                              # Ready stack engine.
1794 pend_func credits                      # Last thing to print.
1795
1796 # # # Deal with user # # #
1797 live_log_die                           # Setup debug trace log.
1798 data_capture                           # Setup data capture file.
1799 echo
1800 do_user_args $@
```

```
1801
1802 # # # Haven't exited yet - There is some hope # # #
1803 # Discovery group - Execution engine is LIFO - pend
1804 # in reverse order of execution.
1805 _hs_RC=0                                # Hunt the Spammer return code
1806 pend_mark
1807     pend_func report_pairs              # Report name-address pairs.
1808
1809     # The two detail_* are mutually recursive functions.
1810     # They also pend expand_* functions as required.
1811     # These two (the last of ???) exit the recursion.
1812     pend_func detail_each_address       # Get all resources of addresses.
1813     pend_func detail_each_name          # Get all resources of names.
1814
1815     #  The two expand_* are mutually recursive functions,
1816     #+ which pend additional detail_* functions as required.
1817     pend_func expand_input_address 1    # Expand input names by address.
1818     pend_func expand_input_name 1       # #xpand input addresses by name.
1819
1820     # Start with a unique set of names and addresses.
1821     pend_func unique_lines uc_address uc_address
1822     pend_func unique_lines uc_name uc_name
1823
1824     # Separate mixed input of names and addresses.
1825     pend_func split_input
1826 pend_release
1827
1828 # # # Pairs reported -- Unique list of IP addresses found
1829 echo
1830 _ip_cnt=${#known_address[@]}
1831 if [ ${#list_server[@]} -eq 0 ]
1832 then
1833     echo 'Blacklist server list empty, none checked.'
1834 else
1835     if [ ${_ip_cnt} -eq 0 ]
1836     then
1837         echo 'Known address list empty, none checked.'
1838     else
1839         _ip_cnt=${_ip_cnt}-1   # Start at top.
1840         echo 'Checking Blacklist servers.'
1841         for (( _ip = _ip_cnt ; _ip >= 0 ; _ip-- ))
1842         do
1843             pend_func check_lists $( printf '%q\n' ${known_address[$_ip]} )
1844         done
1845     fi
1846 fi
1847 pend_release
1848 $_dot_dump                  # Graphics file dump
1849 $_log_dump                  # Execution trace
1850 echo
1851
1852
1853 ###############################
1854 # Example output from script #
1855 ###############################
1856 :<<-'_is_spammer_outputs_'
1857
1858 ./is_spammer.bash 0 web4.alojamentos7.com
1859
1860 Starting with domain name >web4.alojamentos7.com<
1861 Using default blacklist server list.
1862 Search depth limit: 0
```

```
1863 .:...::::..:::..::::.....::..:::..::::.....::
1864 Known network pairs.
1865     66.98.208.97            web4.alojamentos7.com.
1866     66.98.208.97            ns1.alojamentos7.com.
1867     69.56.202.147           ns2.alojamentos.ws.
1868     66.98.208.97            alojamentos7.com.
1869     66.98.208.97            web.alojamentos7.com.
1870     69.56.202.146           ns1.alojamentos.ws.
1871     69.56.202.146           alojamentos.ws.
1872     66.235.180.113          ns1.alojamentos.org.
1873     66.235.181.192          ns2.alojamentos.org.
1874     66.235.180.113          alojamentos.org.
1875     66.235.180.113          web6.alojamentos.org.
1876     216.234.234.30          ns1.theplanet.com.
1877     12.96.160.115           ns2.theplanet.com.
1878     216.185.111.52          mail1.theplanet.com.
1879     69.56.141.4             spooling.theplanet.com.
1880     216.185.111.40          theplanet.com.
1881     216.185.111.40          www.theplanet.com.
1882     216.185.111.52          mail.theplanet.com.
1883
1884 Checking Blacklist servers.
1885     Checking address 66.98.208.97
1886         Records from dnsbl.sorbs.net
1887             "Spam Received See:
http://www.dnsbl.sorbs.net/lookup.shtml?66.98.208.97"
1888     Checking address 69.56.202.147
1889     Checking address 69.56.202.146
1890     Checking address 66.235.180.113
1891     Checking address 66.235.181.192
1892     Checking address 216.185.111.40
1893     Checking address 216.234.234.30
1894     Checking address 12.96.160.115
1895     Checking address 216.185.111.52
1896     Checking address 69.56.141.4
1897
1898 Advanced Bash Scripting Guide: is_spammer.bash, v2, 2004-msz
1899
1900 _is_spammer_outputs_
1901
1902 exit ${_hs_RC}
1903
1904 ######################################################
1905 #  The script ignores everything from here on down #
1906 #+ because of the 'exit' command, just above.      #
1907 ######################################################
1908
1909
1910
1911 Quickstart
1912 ==========
1913
1914  Prerequisites
1915
1916   Bash version 2.05b or 3.00 (bash --version)
1917   A version of Bash which supports arrays. Array
1918   support is included by default Bash configurations.
1919
1920   'dig,' version 9.x.x (dig $HOSTNAME, see first line of output)
1921   A version of dig which supports the +short options.
1922   See: dig_wrappers.bash for details.
1923
```

```
1924
1925  Optional Prerequisites
1926
1927   'named,' a local DNS caching program. Any flavor will do.
1928   Do twice: dig $HOSTNAME
1929   Check near bottom of output for: SERVER: 127.0.0.1#53
1930   That means you have one running.
1931
1932
1933  Optional Graphics Support
1934
1935   'date,' a standard *nix thing. (date -R)
1936
1937   dot Program to convert graphic description file to a
1938   diagram. (dot -V)
1939   A part of the Graph-Viz set of programs.
1940   See: [http://www.research.att.com/sw/tools/graphviz||GraphViz]
1941
1942   'dotty,' a visual editor for graphic description files.
1943   Also a part of the Graph-Viz set of programs.
1944
1945
1946
1947
1948  Quick Start
1949
1950 In the same directory as the is_spammer.bash script;
1951 Do: ./is_spammer.bash
1952
1953  Usage Details
1954
1955 1. Blacklist server choices.
1956
1957   (a) To use default, built-in list: Do nothing.
1958
1959   (b) To use your own list:
1960
1961     i. Create a file with a single Blacklist server
1962        domain name per line.
1963
1964     ii. Provide that filename as the last argument to
1965         the script.
1966
1967   (c) To use a single Blacklist server: Last argument
1968       to the script.
1969
1970   (d) To disable Blacklist lookups:
1971
1972     i. Create an empty file (touch spammer.nul)
1973        Your choice of filename.
1974
1975     ii. Provide the filename of that empty file as the
1976         last argument to the script.
1977
1978 2. Search depth limit.
1979
1980   (a) To use the default value of 2: Do nothing.
1981
1982   (b) To set a different limit:
1983       A limit of 0 means: no limit.
1984
1985     i. export SPAMMER_LIMIT=1
```

```
1986        or whatever limit you want.
1987
1988     ii. OR provide the desired limit as the first
1989         argument to the script.
1990
1991 3. Optional execution trace log.
1992
1993   (a) To use the default setting of no log output: Do nothing.
1994
1995   (b) To write an execution trace log:
1996       export SPAMMER_TRACE=spammer.log
1997       or whatever filename you want.
1998
1999 4. Optional graphic description file.
2000
2001   (a) To use the default setting of no graphic file: Do nothing.
2002
2003   (b) To write a Graph-Viz graphic description file:
2004       export SPAMMER_DATA=spammer.dot
2005       or whatever filename you want.
2006
2007 5. Where to start the search.
2008
2009   (a) Starting with a single domain name:
2010
2011     i. Without a command line search limit: First
2012        argument to script.
2013
2014     ii. With a command line search limit: Second
2015         argument to script.
2016
2017   (b) Starting with a single IP address:
2018
2019     i. Without a command line search limit: First
2020        argument to script.
2021
2022     ii. With a command line search limit: Second
2023         argument to script.
2024
2025   (c) Starting with (mixed) multiple name(s) and/or address(es):
2026       Create a file with one name or address per line.
2027       Your choice of filename.
2028
2029     i. Without a command line search limit: Filename as
2030        first argument to script.
2031
2032     ii. With a command line search limit: Filename as
2033         second argument to script.
2034
2035 6. What to do with the display output.
2036
2037   (a) To view display output on screen: Do nothing.
2038
2039   (b) To save display output to a file: Redirect stdout to a filename.
2040
2041   (c) To discard display output: Redirect stdout to /dev/null.
2042
2043 7. Temporary end of decision making.
2044    press RETURN
2045    wait (optionally, watch the dots and colons).
2046
2047 8. Optionally check the return code.
```

```
2048
2049   (a) Return code 0: All OK
2050
2051   (b) Return code 1: Script setup failure
2052
2053   (c) Return code 2: Something was blacklisted.
2054
2055 9. Where is my graph (diagram)?
2056
2057 The script does not directly produce a graph (diagram).
2058 It only produces a graphic description file. You can
2059 process the graphic descriptor file that was output
2060 with the 'dot' program.
2061
2062 Until you edit that descriptor file, to describe the
2063 relationships you want shown, all that you will get is
2064 a bunch of labeled name and address nodes.
2065
2066 All of the script's discovered relationships are within
2067 a comment block in the graphic descriptor file, each
2068 with a descriptive heading.
2069
2070 The editing required to draw a line between a pair of
2071 nodes from the information in the descriptor file may
2072 be done with a text editor.
2073
2074 Given these lines somewhere in the descriptor file:
2075
2076 # Known domain name nodes
2077
2078 N0000 [label="guardproof.info."] ;
2079
2080 N0002 [label="third.guardproof.info."] ;
2081
2082
2083
2084 # Known address nodes
2085
2086 A0000 [label="61.141.32.197"] ;
2087
2088
2089
2090 /*
2091
2092 # Known name->address edges
2093
2094 NA0000 third.guardproof.info. 61.141.32.197
2095
2096
2097
2098 # Known parent->child edges
2099
2100 PC0000 guardproof.info. third.guardproof.info.
2101
2102  */
2103
2104 Turn that into the following lines by substituting node
2105 identifiers into the relationships:
2106
2107 # Known domain name nodes
2108
2109 N0000 [label="guardproof.info."] ;
```

```
2110
2111 N0002 [label="third.guardproof.info."] ;
2112
2113
2114
2115 # Known address nodes
2116
2117 A0000 [label="61.141.32.197"] ;
2118
2119
2120
2121 # PC0000 guardproof.info. third.guardproof.info.
2122
2123 N0000->N0002 ;
2124
2125
2126
2127 # NA0000 third.guardproof.info. 61.141.32.197
2128
2129 N0002->A0000 ;
2130
2131
2132
2133 /*
2134
2135 # Known name->address edges
2136
2137 NA0000 third.guardproof.info. 61.141.32.197
2138
2139
2140
2141 # Known parent->child edges
2142
2143 PC0000 guardproof.info. third.guardproof.info.
2144
2145  */
2146
2147 Process that with the 'dot' program, and you have your
2148 first network diagram.
2149
2150 In addition to the conventional graphic edges, the
2151 descriptor file includes similar format pair-data that
2152 describes services, zone records (sub-graphs?),
2153 blacklisted addresses, and other things which might be
2154 interesting to include in your graph. This additional
2155 information could be displayed as different node
2156 shapes, colors, line sizes, etc.
2157
2158 The descriptor file can also be read and edited by a
2159 Bash script (of course). You should be able to find
2160 most of the functions required within the
2161 "is_spammer.bash" script.
2162
2163 # End Quickstart.
```

Per terminare la sezione, un ripasso dei fondamenti . . . ed altro.

Esempio A-27. Fondamenti rivisitati

```
  1 #!/bin/bash
  2 # basics-reviewed.bash
```

```
 3
 4 # File extension == *.bash == specific to Bash
 5
 6 #   Copyright (c) Michael S. Zick, 2003; All rights reserved.
 7 #   License: Use in any form, for any purpose.
 8 #   Revision: $ID$
 9 #
10 #             Edited for layout by M.C.
11 #   (author of the "Advanced Bash Scripting Guide")
12
13
14 #  This script tested under Bash versions 2.04, 2.05a and 2.05b.
15 #  It may not work with earlier versions.
16 #  This demonstration script generates one --intentional--
17 #+ "command not found" error message. See line 394.
18
19 #  The current Bash maintainer, Chet Ramey, has fixed the items noted
20 #+ for an upcoming version of Bash.
21
22
23
24         ###-------------------------------------------###
25         ###  Pipe the output of this script to 'more' ###
26         ###+ else it will scroll off the page.        ###
27         ###                                           ###
28         ###  You may also redirect its output         ###
29         ###+ to a file for examination.               ###
30         ###-------------------------------------------###
31
32
33
34 #  Most of the following points are described at length in
35 #+ the text of the foregoing "Advanced Bash Scripting Guide."
36 #  This demonstration script is mostly just a reorganized presentation.
37 #      -- msz
38
39 # Variables are not typed unless otherwise specified.
40
41 #  Variables are named. Names must contain a non-digit.
42 #  File descriptor names (as in, for example: 2>&1)
43 #+ contain ONLY digits.
44
45 # Parameters and Bash array elements are numbered.
46 # (Parameters are very similar to Bash arrays.)
47
48 # A variable name may be undefined (null reference).
49 unset VarNull
50
51 # A variable name may be defined but empty (null contents).
52 VarEmpty=''                           # Two, adjacent, single quotes.
53
54 # A variable name my be defined and non-empty
55 VarSomething='Literal'
56
57 # A variable may contain:
58 #   * A whole number as a signed 32-bit (or larger) integer
59 #   * A string
60 # A variable may also be an array.
61
62 #  A string may contain embedded blanks and may be treated
63 #+ as if it where a function name with optional arguments.
64
```

```
 65 #  The names of variables and the names of functions
 66 #+ are in different namespaces.
 67
 68
 69 #  A variable may be defined as a Bash array either explicitly or
 70 #+ implicitly by the syntax of the assignment statement.
 71 #  Explicit:
 72 declare -a ArrayVar
 73
 74
 75
 76 # The echo command is a built-in.
 77 echo $VarSomething
 78
 79 # The printf command is a built-in.
 80 # Translate %s as: String-Format
 81 printf %s $VarSomething         # No linebreak specified, none output.
 82 echo                           # Default, only linebreak output.
 83
 84
 85
 86
 87 # The Bash parser word breaks on whitespace.
 88 # Whitespace, or the lack of it is significant.
 89 # (This holds true in general; there are, of course, exceptions.)
 90
 91
 92
 93
 94 # Translate the DOLLAR_SIGN character as: Content-Of.
 95
 96 # Extended-Syntax way of writing Content-Of:
 97 echo ${VarSomething}
 98
 99 #  The ${ ... } Extended-Syntax allows more than just the variable
100 #+ name to be specified.
101 #  In general, $VarSomething can always be written as: ${VarSomething}.
102
103 # Call this script with arguments to see the following in action.
104
105
106
107 #  Outside of double-quotes, the special characters @ and *
108 #+ specify identical behavior.
109 #  May be pronounced as: All-Elements-Of.
110
111 #  Without specification of a name, they refer to the
112 #+ pre-defined parameter Bash-Array.
113
114
115
116 # Glob-Pattern references
117 echo $*                         # All parameters to script or function
118 echo ${*}                       # Same
119
120 # Bash disables filename expansion for Glob-Patterns.
121 # Only character matching is active.
122
123
124 # All-Elements-Of references
125 echo $@                         # Same as above
126 echo ${@}                       # Same as above
```

```
127
128
129
130
131 #  Within double-quotes, the behavior of Glob-Pattern references
132 #+ depends on the setting of IFS (Input Field Separator).
133 #  Within double-quotes, All-Elements-Of references behave the same.
134
135
136 #  Specifying only the name of a variable holding a string refers
137 #+ to all elements (characters) of a string.
138
139
140 #  To specify an element (character) of a string,
141 #+ the Extended-Syntax reference notation (see below) MAY be used.
142
143
144
145
146 #  Specifying only the name of a Bash array references
147 #+ the subscript zero element,
148 #+ NOT the FIRST DEFINED nor the FIRST WITH CONTENTS element.
149
150 #  Additional qualification is needed to reference other elements,
151 #+ which means that the reference MUST be written in Extended-Syntax.
152 #  The general form is: ${name[subscript]}.
153
154 #  The string forms may also be used: ${name:subscript}
155 #+ for Bash-Arrays when referencing the subscript zero element.
156
157
158 # Bash-Arrays are implemented internally as linked lists,
159 #+ not as a fixed area of storage as in some programming languages.
160
161
162 #   Characteristics of Bash arrays (Bash-Arrays):
163 #   ---------------------------------------------
164
165 #   If not otherwise specified, Bash-Array subscripts begin with
166 #+  subscript number zero. Literally: [0]
167 #   This is called zero-based indexing.
168 ###
169 #   If not otherwise specified, Bash-Arrays are subscript packed
170 #+  (sequential subscripts without subscript gaps).
171 ###
172 #   Negative subscripts are not allowed.
173 ###
174 #   Elements of a Bash-Array need not all be of the same type.
175 ###
176 #   Elements of a Bash-Array may be undefined (null reference).
177 #       That is, a Bash-Array my be "subscript sparse."
178 ###
179 #   Elements of a Bash-Array may be defined and empty (null contents).
180 ###
181 #   Elements of a Bash-Array may contain:
182 #     * A whole number as a signed 32-bit (or larger) integer
183 #     * A string
184 #     * A string formated so that it appears to be a function name
185 #     + with optional arguments
186 ###
187 #   Defined elements of a Bash-Array may be undefined (unset).
188 #       That is, a subscript packed Bash-Array may be changed
```

```
#   +   into a subscript sparse Bash-Array.
###
#   Elements may be added to a Bash-Array by defining an element
#+  not previously defined.
###
# For these reasons, I have been calling them "Bash-Arrays".
# I'll return to the generic term "array" from now on.
#     -- msz



#  Demo time -- initialize the previously declared ArrayVar as a
#+ sparse array.
#  (The 'unset ... ' is just documentation here.)

unset ArrayVar[0]                   # Just for the record
ArrayVar[1]=one                     # Unquoted literal
ArrayVar[2]=''                      # Defined, and empty
unset ArrayVar[3]                   # Just for the record
ArrayVar[4]='four'                  # Quoted literal



# Translate the %q format as: Quoted-Respecting-IFS-Rules.
echo
echo '- - Outside of double-quotes - -'
###
printf %q ${ArrayVar[*]}            # Glob-Pattern All-Elements-Of
echo
echo 'echo command:'${ArrayVar[*]}
###
printf %q ${ArrayVar[@]}            # All-Elements-Of
echo
echo 'echo command:'${ArrayVar[@]}

# The use of double-quotes may be translated as: Enable-Substitution.

# There are five cases recognized for the IFS setting.

echo
echo '- - Within double-quotes - Default IFS of space-tab-newline - -'
IFS=$'\x20'$'\x09'$'\x0A'           #  These three bytes,
                                    #+ in exactly this order.


printf %q "${ArrayVar[*]}"          # Glob-Pattern All-Elements-Of
echo
echo 'echo command:'"${ArrayVar[*]}"
###
printf %q "${ArrayVar[@]}"          # All-Elements-Of
echo
echo 'echo command:'"${ArrayVar[@]}"


echo
echo '- - Within double-quotes - First character of IFS is ^ - -'
# Any printing, non-whitespace character should do the same.
IFS='^'$IFS                         # ^ + space tab newline
###
printf %q "${ArrayVar[*]}"          # Glob-Pattern All-Elements-Of
echo
```

```
echo 'echo command:'"${ArrayVar[*]}"
###
printf %q "${ArrayVar[@]}"          # All-Elements-Of
echo
echo 'echo command:'"${ArrayVar[@]}"


echo
echo '- - Within double-quotes - Without whitespace in IFS - -'
IFS='^:%!'
###
printf %q "${ArrayVar[*]}"          # Glob-Pattern All-Elements-Of
echo
echo 'echo command:'"${ArrayVar[*]}"
###
printf %q "${ArrayVar[@]}"          # All-Elements-Of
echo
echo 'echo command:'"${ArrayVar[@]}"


echo
echo '- - Within double-quotes - IFS set and empty - -'
IFS=''
###
printf %q "${ArrayVar[*]}"          # Glob-Pattern All-Elements-Of
echo
echo 'echo command:'"${ArrayVar[*]}"
###
printf %q "${ArrayVar[@]}"          # All-Elements-Of
echo
echo 'echo command:'"${ArrayVar[@]}"


echo
echo '- - Within double-quotes - IFS undefined - -'
unset IFS
###
printf %q "${ArrayVar[*]}"          # Glob-Pattern All-Elements-Of
echo
echo 'echo command:'"${ArrayVar[*]}"
###
printf %q "${ArrayVar[@]}"          # All-Elements-Of
echo
echo 'echo command:'"${ArrayVar[@]}"


# Put IFS back to the default.
# Default is exactly these three bytes.
IFS=$'\x20'$'\x09'$'\x0A'           # In exactly this order.

# Interpretation of the above outputs:
#   A Glob-Pattern is I/O; the setting of IFS matters.
###
#   An All-Elements-Of does not consider IFS settings.
###
#   Note the different output using the echo command and the
#+  quoted format operator of the printf command.


#  Recall:
#   Parameters are similar to arrays and have the similar behaviors.
###
```

```
313 #  The above examples demonstrate the possible variations.
314 #  To retain the shape of a sparse array, additional script
315 #+ programming is required.
316 ###
317 #  The source code of Bash has a routine to output the
318 #+ [subscript]=value   array assignment format.
319 #  As of version 2.05b, that routine is not used,
320 #+ but that might change in future releases.
321
322
323
324 # The length of a string, measured in non-null elements (characters):
325 echo
326 echo '- - Non-quoted references - -'
327 echo 'Non-Null character count: '${#VarSomething}' characters.'
328
329 # test='Lit'$'\x00''eral'           # $'\x00' is a null character.
330 # echo ${#test}                     # See that?
331
332
333
334 #  The length of an array, measured in defined elements,
335 #+ including null content elements.
336 echo
337 echo 'Defined content count: '${#ArrayVar[@]}' elements.'
338 # That is NOT the maximum subscript (4).
339 # That is NOT the range of the subscripts (1 . . 4 inclusive).
340 # It IS the length of the linked list.
341 ###
342 #  Both the maximum subscript and the range of the subscripts may
343 #+ be found with additional script programming.
344
345 # The length of a string, measured in non-null elements (characters):
346 echo
347 echo '- - Quoted, Glob-Pattern references - -'
348 echo 'Non-Null character count: '"${#VarSomething}"' characters.'
349
350 #  The length of an array, measured in defined elements,
351 #+ including null-content elements.
352 echo
353 echo 'Defined element count: '"${#ArrayVar[*]}"' elements.'
354
355 #  Interpretation: Substitution does not effect the ${# ... } operation.
356 #  Suggestion:
357 #  Always use the All-Elements-Of character
358 #+ if that is what is intended (independence from IFS).
359
360
361
362 #  Define a simple function.
363 #  I include an underscore in the name
364 #+ to make it distinctive in the examples below.
365 ###
366 #  Bash separates variable names and function names
367 #+ in different namespaces.
368 #  The Mark-One eyeball isn't that advanced.
369 ###
370 _simple() {
371     echo -n 'SimpleFunc'$@          #  Newlines are swallowed in
372 }                                   #+ result returned in any case.
373
374
```

```
375 # The ( ... ) notation invokes a command or function.
376 # The $( ... ) notation is pronounced: Result-Of.
377
378
379 # Invoke the function _simple
380 echo
381 echo '- - Output of function _simple - -'
382 _simple                             # Try passing arguments.
383 echo
384 # or
385 (_simple)                           # Try passing arguments.
386 echo
387
388 echo '- Is there a variable of that name? -'
389 echo $_simple not defined           # No variable by that name.
390
391 # Invoke the result of function _simple (Error msg intended)
392
393 ###
394 $(_simple)                          # Gives an error message:
395 #                          line 394: SimpleFunc: command not found
396 #                          ---------------------------------------
397
398 echo
399 ###
400
401 #  The first word of the result of function _simple
402 #+ is neither a valid Bash command nor the name of a defined function.
403 ###
404 # This demonstrates that the output of _simple is subject to evaluation.
405 ###
406 # Interpretation:
407 #   A function can be used to generate in-line Bash commands.
408
409
410 # A simple function where the first word of result IS a bash command:
411 ###
412 _print() {
413     echo -n 'printf %q '$@
414 }
415
416 echo '- - Outputs of function _print - -'
417 _print parm1 parm2                  # An Output NOT A Command.
418 echo
419
420 $(_print parm1 parm2)               #  Executes: printf %q parm1 parm2
421                                     #  See above IFS examples for the
422                                     #+ various possibilities.
423 echo
424
425 $(_print $VarSomething)             # The predictable result.
426 echo
427
428
429
430 # Function variables
431 # ------------------
432
433 echo
434 echo '- - Function variables - -'
435 # A variable may represent a signed integer, a string or an array.
436 # A string may be used like a function name with optional arguments.
```

```
437
438 # set -vx                           #  Enable if desired
439 declare -f funcVar                  #+ in namespace of functions
440
441 funcVar=_print                      # Contains name of function.
442 $funcVar parm1                      # Same as _print at this point.
443 echo
444
445 funcVar=$(_print )                  # Contains result of function.
446 $funcVar                            # No input, No output.
447 $funcVar $VarSomething              # The predictable result.
448 echo
449
450 funcVar=$(_print $VarSomething)     #  $VarSomething replaced HERE.
451 $funcVar                            #  The expansion is part of the
452 echo                                #+ variable contents.
453
454 funcVar="$(_print $VarSomething)"   #  $VarSomething replaced HERE.
455 $funcVar                            #  The expansion is part of the
456 echo                                #+ variable contents.
457
458 #  The difference between the unquoted and the double-quoted versions
459 #+ above can be seen in the "protect_literal.sh" example.
460 #  The first case above is processed as two, unquoted, Bash-Words.
461 #  The second case above is processed as one, quoted, Bash-Word.
462
463
464
465
466 # Delayed replacement
467 # -------------------
468
469 echo
470 echo '- - Delayed replacement - -'
471 funcVar="$(_print '$VarSomething')" # No replacement, single Bash-Word.
472 eval $funcVar                       # $VarSomething replaced HERE.
473 echo
474
475 VarSomething='NewThing'
476 eval $funcVar                       # $VarSomething replaced HERE.
477 echo
478
479 # Restore the original setting trashed above.
480 VarSomething=Literal
481
482 #  There are a pair of functions demonstrated in the
483 #+ "protect_literal.sh" and "unprotect_literal.sh" examples.
484 #  These are general purpose functions for delayed replacement literals
485 #+ containing variables.
486
487
488
489
490
491 # REVIEW:
492 # ------
493
494 #  A string can be considered a Classic-Array of elements (characters).
495 #  A string operation applies to all elements (characters) of the string
496 #+ (in concept, anyway).
497 ###
498 #  The notation: ${array_name[@]} represents all elements of the
```

```
#+ Bash-Array: array_name.
###
#  The Extended-Syntax string operations can be applied to all
#+ elements of an array.
###
#  This may be thought of as a For-Each operation on a vector of strings.
###
#  Parameters are similar to an array.
#  The initialization of a parameter array for a script
#+ and a parameter array for a function only differ
#+ in the initialization of ${0}, which never changes its setting.
###
#  Subscript zero of the script's parameter array contains
#+ the name of the script.
###
#  Subscript zero of a function's parameter array DOES NOT contain
#+ the name of the function.
#  The name of the current function is accessed by the $FUNCNAME variable.
###
#  A quick, review list follows (quick, not short).

echo
echo '- - Test (but not change) - -'
echo '- null reference -'
echo -n ${VarNull-'NotSet'}' '          # NotSet
echo ${VarNull}                         # NewLine only
echo -n ${VarNull:-'NotSet'}' '         # NotSet
echo ${VarNull}                         # Newline only

echo '- null contents -'
echo -n ${VarEmpty-'Empty'}' '          # Only the space
echo ${VarEmpty}                        # Newline only
echo -n ${VarEmpty:-'Empty'}' '         # Empty
echo ${VarEmpty}                        # Newline only

echo '- contents -'
echo ${VarSomething-'Content'}          # Literal
echo ${VarSomething:-'Content'}         # Literal

echo '- Sparse Array -'
echo ${ArrayVar[@]-'not set'}

# ASCII-Art time
# State     Y==yes, N==no
#           -       :-
# Unset     Y       Y       ${# ... } == 0
# Empty     N       Y       ${# ... } == 0
# Contents  N       N       ${# ... } > 0

#  Either the first and/or the second part of the tests
#+ may be a command or a function invocation string.
echo
echo '- - Test 1 for undefined - -'
declare -i t
_decT() {
    t=$t-1
}

# Null reference, set: t == -1
t=${#VarNull}                           # Results in zero.
${VarNull- _decT }                      # Function executes, t now -1.
echo $t
```

```
561
562 # Null contents, set: t == 0
563 t=${#VarEmpty}                          # Results in zero.
564 ${VarEmpty- _decT }                     # _decT function NOT executed.
565 echo $t
566
567 # Contents, set: t == number of non-null characters
568 VarSomething='_simple'                  # Set to valid function name.
569 t=${#VarSomething}                      # non-zero length
570 ${VarSomething- _decT }                 # Function _simple executed.
571 echo $t                                 # Note the Append-To action.
572
573 # Exercise: clean up that example.
574 unset t
575 unset _decT
576 VarSomething=Literal
577
578 echo
579 echo '- - Test and Change - -'
580 echo '- Assignment if null reference -'
581 echo -n ${VarNull='NotSet'}' '          # NotSet NotSet
582 echo ${VarNull}
583 unset VarNull
584
585 echo '- Assignment if null reference -'
586 echo -n ${VarNull:='NotSet'}' '         # NotSet NotSet
587 echo ${VarNull}
588 unset VarNull
589
590 echo '- No assignment if null contents -'
591 echo -n ${VarEmpty='Empty'}' '          # Space only
592 echo ${VarEmpty}
593 VarEmpty=''
594
595 echo '- Assignment if null contents -'
596 echo -n ${VarEmpty:='Empty'}' '         # Empty Empty
597 echo ${VarEmpty}
598 VarEmpty=''
599
600 echo '- No change if already has contents -'
601 echo ${VarSomething='Content'}          # Literal
602 echo ${VarSomething:='Content'}         # Literal
603
604
605 # "Subscript sparse" Bash-Arrays
606 ###
607 #  Bash-Arrays are subscript packed, beginning with
608 #+ subscript zero unless otherwise specified.
609 ###
610 #  The initialization of ArrayVar was one way
611 #+ to "otherwise specify".  Here is the other way:
612 ###
613 echo
614 declare -a ArraySparse
615 ArraySparse=( [1]=one [2]='' [4]='four' )
616 # [0]=null reference, [2]=null content, [3]=null reference
617
618 echo '- - Array-Sparse List - -'
619 # Within double-quotes, default IFS, Glob-Pattern
620
621 IFS=$'\x20'$'\x09'$'\x0A'
622 printf %q "${ArraySparse[*]}"
```

```
623 echo
624
625 #  Note that the output does not distinguish between "null content"
626 #+ and "null reference".
627 #  Both print as escaped whitespace.
628 ###
629 #  Note also that the output does NOT contain escaped whitespace
630 #+ for the "null reference(s)" prior to the first defined element.
631 ###
632 # This behavior of 2.04, 2.05a and 2.05b has been reported
633 #+ and may change in a future version of Bash.
634
635 #  To output a sparse array and maintain the [subscript]=value
636 #+ relationship without change requires a bit of programming.
637 #  One possible code fragment:
638 ###
639 # local l=${#ArraySparse[@]}        # Count of defined elements
640 # local f=0                         # Count of found subscripts
641 # local i=0                         # Subscript to test
642 (                                   # Anonymous in-line function
643     for (( l=${#ArraySparse[@]}, f = 0, i = 0 ; f < l ; i++ ))
644     do
645         # 'if defined then...'
646         ${ArraySparse[$i]+ eval echo '\ ['$i']='${ArraySparse[$i]} ; (( f++
)) }
647     done
648 )
649
650 # The reader coming upon the above code fragment cold
651 #+ might want to review "command lists" and "multiple commands on a line"
652 #+ in the text of the foregoing "Advanced Bash Scripting Guide."
653 ###
654 #  Note:
655 #  The "read -a array_name" version of the "read" command
656 #+ begins filling array_name at subscript zero.
657 #  ArraySparse does not define a value at subscript zero.
658 ###
659 #  The user needing to read/write a sparse array to either
660 #+ external storage or a communications socket must invent
661 #+ a read/write code pair suitable for their purpose.
662 ###
663 # Exercise: clean it up.
664
665 unset ArraySparse
666
667 echo
668 echo '- - Conditional alternate (But not change)- -'
669 echo '- No alternate if null reference -'
670 echo -n ${VarNull+'NotSet'}' '
671 echo ${VarNull}
672 unset VarNull
673
674 echo '- No alternate if null reference -'
675 echo -n ${VarNull:+'NotSet'}' '
676 echo ${VarNull}
677 unset VarNull
678
679 echo '- Alternate if null contents -'
680 echo -n ${VarEmpty+'Empty'}' '              # Empty
681 echo ${VarEmpty}
682 VarEmpty=''
683
```

```
684 echo '- No alternate if null contents -'
685 echo -n ${VarEmpty:+'Empty'}' '             # Space only
686 echo ${VarEmpty}
687 VarEmpty=''
688
689 echo '- Alternate if already has contents -'
690
691 # Alternate literal
692 echo -n ${VarSomething+'Content'}' '        # Content Literal
693 echo ${VarSomething}
694
695 # Invoke function
696 echo -n ${VarSomething:+ $(_simple) }' '    # SimpleFunc Literal
697 echo ${VarSomething}
698 echo
699
700 echo '- - Sparse Array - -'
701 echo ${ArrayVar[@]+'Empty'}                 # An array of 'Empty'(ies)
702 echo
703
704 echo '- - Test 2 for undefined - -'
705
706 declare -i t
707 _incT() {
708     t=$t+1
709 }
710
711 #  Note:
712 #  This is the same test used in the sparse array
713 #+ listing code fragment.
714
715 # Null reference, set: t == -1
716 t=${#VarNull}-1                    # Results in minus-one.
717 ${VarNull+ _incT }                 # Does not execute.
718 echo $t' Null reference'
719
720 # Null contents, set: t == 0
721 t=${#VarEmpty}-1                   # Results in minus-one.
722 ${VarEmpty+ _incT }                # Executes.
723 echo $t'  Null content'
724
725 # Contents, set: t == (number of non-null characters)
726 t=${#VarSomething}-1               # non-null length minus-one
727 ${VarSomething+ _incT }            # Executes.
728 echo $t'  Contents'
729
730 # Exercise: clean up that example.
731 unset t
732 unset _incT
733
734 # ${name?err_msg} ${name:?err_msg}
735 #  These follow the same rules but always exit afterwards
736 #+ if an action is specified following the question mark.
737 #  The action following the question mark may be a literal
738 #+ or a function result.
739 ###
740 #  ${name?} ${name:?} are test-only, the return can be tested.
741
742
743
744
745 # Element operations
```

```
# ------------------

echo
echo '- - Trailing sub-element selection - -'

#  Strings, Arrays and Positional parameters

#  Call this script with multiple arguments
#+ to see the parameter selections.

echo '- All -'
echo ${VarSomething:0}              # all non-null characters
echo ${ArrayVar[@]:0}               # all elements with content
echo ${@:0}                         # all parameters with content;
                                    # ignoring parameter[0]

echo
echo '- All after -'
echo ${VarSomething:1}              # all non-null after character[0]
echo ${ArrayVar[@]:1}               # all after element[0] with content
echo ${@:2}                         # all after param[1] with content

echo
echo '- Range after -'
echo ${VarSomething:4:3}            # ral
                                    # Three characters after
                                    # character[3]

echo '- Sparse array gotch -'
echo ${ArrayVar[@]:1:2}     #  four - The only element with content.
                            #  Two elements after (if that many exist).
                            #  the FIRST WITH CONTENTS
                            #+ (the FIRST WITH  CONTENTS is being
                            #+ considered as if it
                            #+ were subscript zero).
#  Executed as if Bash considers ONLY array elements with CONTENT
#  printf %q "${ArrayVar[@]:0:3}"    # Try this one

#  In versions 2.04, 2.05a and 2.05b,
#+ Bash does not handle sparse arrays as expected using this notation.
#
#  The current Bash maintainer, Chet Ramey, has corrected this
#+ for an upcoming version of Bash.


echo '- Non-sparse array -'
echo ${@:2:2}               # Two parameters following parameter[1]

# New victims for string vector examples:
stringZ=abcABC123ABCabc
arrayZ=( abcabc ABCABC 123123 ABCABC abcabc )
sparseZ=( [1]='abcabc' [3]='ABCABC' [4]='' [5]='123123' )

echo
echo ' - - Victim string - -'$stringZ'- - '
echo ' - - Victim array - -'${arrayZ[@]}'- - '
echo ' - - Sparse array - -'${sparseZ[@]}'- - '
echo ' - [0]==null ref, [2]==null ref, [4]==null content - '
echo ' - [1]=abcabc [3]=ABCABC [5]=123123 - '
echo ' - non-null-reference count: '${#sparseZ[@]}' elements'

echo
```

```
808 echo '- - Prefix sub-element removal - -'
809 echo '- - Glob-Pattern match must include the first character. - -'
810 echo '- - Glob-Pattern may be a literal or a function result. - -'
811 echo
812
813
814 # Function returning a simple, Literal, Glob-Pattern
815 _abc() {
816     echo -n 'abc'
817 }
818
819 echo '- Shortest prefix -'
820 echo ${stringZ#123}                 # Unchanged (not a prefix).
821 echo ${stringZ#$(_abc)}             # ABC123ABCabc
822 echo ${arrayZ[@]#abc}               # Applied to each element.
823
824 # Fixed by Chet Ramey for an upcoming version of Bash.
825 # echo ${sparseZ[@]#abc}            # Version-2.05b core dumps.
826
827 # The -it would be nice- First-Subscript-Of
828 # echo ${#sparseZ[@]#*}             # This is NOT valid Bash.
829
830 echo
831 echo '- Longest prefix -'
832 echo ${stringZ##1*3}                # Unchanged (not a prefix)
833 echo ${stringZ##a*C}                # abc
834 echo ${arrayZ[@]##a*c}              # ABCABC 123123 ABCABC
835
836 # Fixed by Chet Ramey for an upcoming version of Bash
837 # echo ${sparseZ[@]##a*c}           # Version-2.05b core dumps.
838
839 echo
840 echo '- - Suffix sub-element removal - -'
841 echo '- - Glob-Pattern match must include the last character. - -'
842 echo '- - Glob-Pattern may be a literal or a function result. - -'
843 echo
844 echo '- Shortest suffix -'
845 echo ${stringZ%1*3}                 # Unchanged (not a suffix).
846 echo ${stringZ%$(_abc)}             # abcABC123ABC
847 echo ${arrayZ[@]%abc}               # Applied to each element.
848
849 # Fixed by Chet Ramey for an upcoming version of Bash.
850 # echo ${sparseZ[@]%abc}            # Version-2.05b core dumps.
851
852 # The -it would be nice- Last-Subscript-Of
853 # echo ${#sparseZ[@]%*}             # This is NOT valid Bash.
854
855 echo
856 echo '- Longest suffix -'
857 echo ${stringZ%%1*3}                # Unchanged (not a suffix)
858 echo ${stringZ%%b*c}                # a
859 echo ${arrayZ[@]%%b*c}              # a ABCABC 123123 ABCABC a
860
861 # Fixed by Chet Ramey for an upcoming version of Bash.
862 # echo ${sparseZ[@]%%b*c}           # Version-2.05b core dumps.
863
864 echo
865 echo '- - Sub-element replacement - -'
866 echo '- - Sub-element at any location in string. - -'
867 echo '- - First specification is a Glob-Pattern - -'
868 echo '- - Glob-Pattern may be a literal or Glob-Pattern function result. - -
'
```

```
869 echo '- - Second specification may be a literal or function result. - -'
870 echo '- - Second specification may be unspecified. Pronounce that'
871 echo '    as: Replace-With-Nothing (Delete) - -'
872 echo
873
874
875
876 # Function returning a simple, Literal, Glob-Pattern
877 _123() {
878     echo -n '123'
879 }
880
881 echo '- Replace first occurrence -'
882 echo ${stringZ/$(_123)/999}         # Changed (123 is a component).
883 echo ${stringZ/ABC/xyz}             # xyzABC123ABCabc
884 echo ${arrayZ[@]/ABC/xyz}           # Applied to each element.
885 echo ${sparseZ[@]/ABC/xyz}          # Works as expected.
886
887 echo
888 echo '- Delete first occurrence -'
889 echo ${stringZ/$(_123)/}
890 echo ${stringZ/ABC/}
891 echo ${arrayZ[@]/ABC/}
892 echo ${sparseZ[@]/ABC/}
893
894 #  The replacement need not be a literal,
895 #+ since the result of a function invocation is allowed.
896 #  This is general to all forms of replacement.
897 echo
898 echo '- Replace first occurrence with Result-Of -'
899 echo ${stringZ/$(_123)/$(_simple)}  # Works as expected.
900 echo ${arrayZ[@]/ca/$(_simple)}     # Applied to each element.
901 echo ${sparseZ[@]/ca/$(_simple)}    # Works as expected.
902
903 echo
904 echo '- Replace all occurrences -'
905 echo ${stringZ//[b2]/X}             # X-out b's and 2's
906 echo ${stringZ//abc/xyz}            # xyzABC123ABCxyz
907 echo ${arrayZ[@]//abc/xyz}          # Applied to each element.
908 echo ${sparseZ[@]//abc/xyz}         # Works as expected.
909
910 echo
911 echo '- Delete all occurrences -'
912 echo ${stringZ//[b2]/}
913 echo ${stringZ//abc/}
914 echo ${arrayZ[@]//abc/}
915 echo ${sparseZ[@]//abc/}
916
917 echo
918 echo '- - Prefix sub-element replacement - -'
919 echo '- - Match must include the first character. - -'
920 echo
921
922 echo '- Replace prefix occurrences -'
923 echo ${stringZ/#[b2]/X}             # Unchanged (neither is a prefix).
924 echo ${stringZ/#$(_abc)/XYZ}        # XYZABC123ABCabc
925 echo ${arrayZ[@]/#abc/XYZ}          # Applied to each element.
926 echo ${sparseZ[@]/#abc/XYZ}         # Works as expected.
927
928 echo
929 echo '- Delete prefix occurrences -'
930 echo ${stringZ/#[b2]/}
```

```
931 echo ${stringZ/#$(_abc)/}
932 echo ${arrayZ[@]/#abc/}
933 echo ${sparseZ[@]/#abc/}
934
935 echo
936 echo '- - Suffix sub-element replacement - -'
937 echo '- - Match must include the last character. - -'
938 echo
939
940 echo '- Replace suffix occurrences -'
941 echo ${stringZ/%[b2]/X}             # Unchanged (neither is a suffix).
942 echo ${stringZ/%$(_abc)/XYZ}        # abcABC123ABCXYZ
943 echo ${arrayZ[@]/%abc/XYZ}          # Applied to each element.
944 echo ${sparseZ[@]/%abc/XYZ}         # Works as expected.
945
946 echo
947 echo '- Delete suffix occurrences -'
948 echo ${stringZ/%[b2]/}
949 echo ${stringZ/%$(_abc)/}
950 echo ${arrayZ[@]/%abc/}
951 echo ${sparseZ[@]/%abc/}
952
953 echo
954 echo '- - Special cases of null Glob-Pattern - -'
955 echo
956
957 echo '- Prefix all -'
958 # null substring pattern means 'prefix'
959 echo ${stringZ/#/NEW}               # NEWabcABC123ABCabc
960 echo ${arrayZ[@]/#/NEW}             # Applied to each element.
961 echo ${sparseZ[@]/#/NEW}            # Applied to null-content also.
962                                     # That seems reasonable.
963
964 echo
965 echo '- Suffix all -'
966 # null substring pattern means 'suffix'
967 echo ${stringZ/%/NEW}               # abcABC123ABCabcNEW
968 echo ${arrayZ[@]/%/NEW}             # Applied to each element.
969 echo ${sparseZ[@]/%/NEW}            # Applied to null-content also.
970                                     # That seems reasonable.
971
972 echo
973 echo '- - Special case For-Each Glob-Pattern - -'
974 echo '- - - - This is a nice-to-have dream - - - -'
975 echo
976
977 _GenFunc() {
978     echo -n ${0}                    # Illustration only.
979     # Actually, that would be an arbitrary computation.
980 }
981
982 # All occurrences, matching the AnyThing pattern.
983 # Currently //*/ does not match null-content nor null-reference.
984 # /#/ and /%/ does match null-content but not null-reference.
985 echo ${sparseZ[@]//*/$(_GenFunc)}
986
987
988 #  A possible syntax would be to make
989 #+ the parameter notation used within this construct mean:
990 #   ${1} - The full element
991 #   ${2} - The prefix, if any, to the matched sub-element
992 #   ${3} - The matched sub-element
```

```
993 #   ${4} - The suffix, if any, to the matched sub-element
994 #
995 # echo ${sparseZ[@]//*/$(_GenFunc ${3})}   # Same as ${1} here.
996 # Perhaps it will be implemented in a future version of Bash.
997
998
999 exit 0
```

Appendice B. Tabelle di riferimento

Le seguenti tabelle rappresentano un utile *riepilogo* di alcuni concetti dello scripting. Nella parte precedente del libro, questi argomenti sono trattati più approfonditamente, con esempi sul loro impiego.

Tabella B-1. Variabili speciali di shell

Variabile	Significato
`$0`	Nome dello script
`$1`	Parametro posizionale nr.1
`$2 - $9`	Parametri posizionali nr.2 - nr.9
`${10}`	Parametro posizionale nr.10
`$#`	Numero dei parametri posizionali
`"$*"`	Tutti i parametri posizionali (come parola singola) *
`"$@"`	Tutti i parametri posizionali (come stringhe separate)
`${#*}`	Numero dei parametri passati allo script da riga di comando
`${#@}`	Numero dei parametri passati allo script da riga di comando
`$?`	Valore di ritorno
`$$`	ID di processo (PID) dello script
`$-`	Opzioni passate allo script (usando *set*)
`$_`	Ultimo argomento del comando precedente
`$!`	ID di processo (PID) dell'ultimo job eseguito in background

***** *È necessario il quoting*, altrimenti viene trattato come "`$@`".

Tabella B-2. Operatori di verifica: confronti binari

Operatore	Significato	-----	Operatore	Significato
Confronto aritmetico			Confronto letterale	
`-eq`	Uguale a		`=`	Uguale a
			`==`	Uguale a
`-ne`	Non uguale a		`!=`	Non uguale a
`-lt`	Minore di		`\<`	Minore di (ASCII) *

Operatore	Significato	-----	Operatore	Significato
`-le`	Minore di o uguale a			
`-gt`	Maggiore di		`\>`	Maggiore di (ASCII) *
`-ge`	Maggiore di o uguale a			
			`-z`	La stringa è vuota
			`-n`	La stringa non è vuota
Confronto aritmetico	tra parentesi doppie ((...))			
`>`	Maggiore di			
`>=`	Maggiore di o uguale a			
`<`	Minore di			
`<=`	Minore di o uguale a			

***** *Se si usa il costrutto doppie parentesi quadre* [[...]] *, allora non è necessario il carattere di escape* \ .

Tabella B-3. Operatori di verifica: file

Operatore	Verifica se	-----	Operatore	Verifica se
`-e`	Il file esiste		`-s`	Il file non è vuoto
`-f`	Il file è un file *regolare*			
`-d`	Il file è una *directory*		`-r`	Il file ha il permesso di *lettura*
`-h`	Il file è un *link simbolico*		`-w`	Il file ha il permesso di *scrittura*
`-L`	Il file è un *link simbolico*		`-x`	Il file ha il permesso di *esecuzione*
`-b`	Il file è un *dispositivo a blocchi*			
`-c`	Il file è un *dispositivo a caratteri*		`-g`	È impostato il bit *sgid*
`-p`	Il file è una *pipe*		`-u`	È impostato il bit *suid*
`-S`	Il file è un socket		`-k`	È impostato lo "sticky bit"
`-t`	Il file è associato a un *terminale*			
`-N`	Il file è stato modificato dall'ultima lettura		`F1 -nt F2`	Il file F1 è *più recente* di F2 *
`-O`	Si è il proprietario del file		`F1 -ot F2`	Il file F1 è *più vecchio* di F2 *
`-G`	L'*id di gruppo* del file è uguale al vostro		`F1 -ef F2`	I file F1 e F2 sono degli *hard link* allo stesso file *
`!`	"NOT" (inverte il senso delle precedenti verifiche)			

***** Operatore *binario* (richiede due operandi).

Tabella B-4. Sostituzione ed espansione di parametro

Espressione	Significato
`${var}`	Valore di *var*, uguale a *$var*
`${var-DEFAULT}`	Se *var* non è impostata, valuta l'espressione al valore di *$DEFAULT* *
`${var:-DEFAULT}`	Se *var* non è impostata o è vuota, valuta l'espressione al valore di *$DEFAULT* *
`${var=DEFAULT}`	Se *var* non è impostata, l'espressione è valutata al valore di *$DEFAULT* *
`${var:=DEFAULT}`	Se *var* non è impostata, l'espressione è valutata al valore di *$DEFAULT* *
`${var+ALTRO}`	Se *var* è impostata, l'espressione è valutata al valore di *$ALTRO*, altrimenti a stringa nulla
`${var:+ALTRO}`	Se *var* è impostata, l'espressione è valutata al valore di *$ALTRO*, altrimenti a stringa nulla
`${var?MSG_ERR}`	Se *var* non è impostata, visualizza *$MSG_ERR* *
`${var:?MSG_ERR}`	Se *var* non è impostata, visualizza *$MSG_ERR* *
`${!varprefix*}`	Verifica tutte le variabili dichiarate precedentemente i cui nomi iniziano con *varprefix*
`${!varprefix@}`	Verifica tutte le variabili dichiarate precedentemente i cui nomi iniziano con *varprefix*

***** Naturalmente se *var è* impostata, l'espressione viene valutata al valore di *$var*.

Tabella B-5. Operazioni su stringhe

Espressione	Significato
`${#stringa}`	Lunghezza di *$stringa*
`${stringa:posizione}`	Estrae una sottostringa da *$stringa* iniziando da *$posizione*
`${stringa:posizione:lunghezza}`	Estrae una sottostringa di *$lunghezza* caratteri da *$stringa* iniziando da *$posizione*
`${stringa#sottostringa}`	Toglie l'occorrenza più breve di *$sottostringa* dalla parte iniziale di *$stringa*
`${stringa##sottostringa}`	Toglie l'occorrenza più lunga di *$sottostringa* dalla parte iniziale di *$stringa*
`${stringa%sottostringa}`	Toglie l'occorrenza più breve di *$sottostringa* dalla parte finale di *$stringa*

Espressione	Significato
`${stringa%%sottostringa}`	Toglie l'occorrenza più lunga di *`$sottostringa`* dalla parte finale di *`$stringa`*
`${stringa/sottostringa/sostituto}`	Sostituisce la prima occorrenza di *`$sottostringa`* con *`$sostituto`*
`${stringa//sottostringa/sostituto}`	Sostituisce *tutte* le occorrenze di *`$sottostringa`* con *`$sostituto`*
`${stringa/#sottostringa/sostituto}`	Se *`$sottostringa`* viene verificata nella parte *iniziale* di *`$stringa`*, allora *`$sottostringa`* viene sostituita con *`$sostituto`*
`${stringa/%sottostringa/sostituto}`	Se *`$sottostringa`* viene verificata nella parte *finale* di *`$stringa`*, allora *`$sottostringa`* viene sostituita con *`$sostituto`*
`expr match "$stringa" '$sottostringa'`	Lunghezza di *`$sottostringa`** verificata nella parte iniziale di *`$stringa`*
`expr "$stringa" : '$sottostringa'`	Lunghezza di *`$sottostringa`** verificata nella parte iniziale di *`$stringa`*
`expr index "$stringa" $sottostringa`	Posizione numerica in *`$stringa`* del primo carattere verificato compreso in *`$sottostringa`*
`expr substr $stringa $posizione $lunghezza`	Estrae *`$lunghezza`* caratteri da *`$stringa`* iniziando da *`$posizione`*
`expr match "$stringa" '\($sottostringa\)'`	Estrae *`$sottostringa`** dalla parte iniziale di *`$stringa`*
`expr "$stringa" : '\($sottostringa\)'`	Estrae *`$sottostringa`** dalla parte iniziale di *`$stringa`*
`expr match "$stringa" '.*\($sottostringa\)'`	Estrae *`$sottostringa`** dalla parte finale di *`$stringa`*
`expr "$stringa" : '.*\($sottostringa\)'`	Estrae *`$sottostringa`** dalla parte finale di *`$stringa`*

***** Dove *`$sottostringa`* è un'*espressione regolare*.

Tabella B-6. Costrutti vari

Espressione	Interpretazione
Parentesi quadre	
`if [ CONDIZIONE ]`	Costrutto di verifica
`if [[ CONDIZIONE ]]`	Costrutto di verifica esteso
`Array[1]=elemento1`	Inizializzazione di array

Espressione	Interpretazione
`[a-z]`	Intervallo di caratteri in un'Espressione Regolare
Parentesi graffe	
`${variabile}`	Sostituzione di parametro
`${!variabile}`	Referenziazione indiretta di variabile
`{ comando1; comando2 }`	Blocco di codice
`{stringa1,stringa2,stringa3,...}`	Espansione multipla
Parentesi	
`( comando1; comando2 )`	Gruppo di comandi eseguiti in una subshell
`Array=(elemento1 elemento2 elemento3)`	Inizializzazione di array
`risultato=$(COMANDO)`	Esegue il comando in una subshell e assegna il risultato alla variabile
`>(COMANDO)`	Sostituzione di processo
`<(COMANDO)`	Sostituzione di processo
Doppie parentesi	
`(( var = 78 ))`	Aritmetica di interi
`var=$(( 20 + 5 ))`	Aritmetica di interi con assegnamento di variabile
Quoting	
`"$variabile"`	Quoting "debole"
`'stringa'`	Quoting "forte"
Apici inversi	
`risultato=`COMANDO``	Esegue il comando in una subshell e assegna il risultato alla variabile

Appendice C. Una breve introduzione a Sed e Awk

Sommario

Questa è una brevissima introduzione alle utility di elaborazione di testo **sed** e **awk**. Qui vengono trattati solamente alcuni comandi di base, ma che saranno sufficienti per comprendere i semplici costrutti sed e awk presenti negli script di shell.

sed: editor non interattivo di file di testo

awk: linguaggio per l'elaborazione di modelli orientato ai campi, con una sintassi simile a quella del C

Nonostante le loro differenze, le due utility condividono una sintassi d'invocazione simile, entrambe fanno uso delle Espressioni Regolari, entrambe leggono l'input, in modo predefinito, dallo `stdin` ed entrambe inviano i risultati allo `stdout`. Sono strumenti UNIX ben collaudati e funzionano bene insieme. L'output dell'una può essere collegato, per mezzo di una pipe, all'altra e le loro capacità combinate danno agli script di shell parte della potenza di Perl.


Un'importante differenza tra le due utility è che mentre gli script di shell possono passare facilmente degli argomenti a sed, è più complicato fare la stessa cosa con awk (vedi Esempio 34-3 e Esempio 9-22).

C.1. Sed

Sed è un editor di riga non interattivo. Riceve un testo come input, o dallo `stdin` o da un file, esegue alcune operazioni sulle righe specificate, una alla volta, quindi invia il risultato allo `stdout` o in un file. Negli script di shell, sed è, di solito, una delle molte componenti di una pipe.

Sed determina le righe dell'input, su cui deve operare, tramite un *indirizzo* che gli è stato passato. [1] Questo indirizzo può essere rappresentato sia da un numero di riga sia da una verifica d'occorrenza. Ad esempio, *`3d`* indica a sed di cancellare la terza riga dell'input, mentre *`/windows/d`* segnala a sed che si vogliono cancellare tutte le righe dell'input contenenti l'occorrenza "windows".

Di tutte le operazioni a disposizione di sed, vengono focalizzate, in primo luogo, le tre più comunemente usate. Esse sono **p**rint (visualizza allo `stdout`), **d**elete (cancella) e **s**ubstitute (sostituisce).

Tabella C-1. Operatori sed di base

Operatore	Nome	Effetto
`[indirizzo]/p`	print	Visualizza [l'indirizzo specificato]
`[indirizzo]/d`	delete	Cancella [l'indirizzo specificato]
`s/modello1/modello2/`	substitute	Sostituisce in ogni riga la prima occorrenza della stringa modello1 con la stringa modello2
`[indirizzo]/s/modello1/modello2/`	substitute	Sostituisce, in tutte le righe specificate in *`indirizzo`*, la prima occorrenza della stringa modello1 con la stringa modello2
`[indirizzo]/y/modello1/modello2/`	transform	sostituisce tutti i caratteri della stringa modello1 con i corrispondenti caratteri della stringa modello2, in tutte le righe specificate da

Operatore	Nome	Effetto
		indirizzo (equivalente di **tr**)
g	global	Agisce su *tutte* le verifiche d'occorrenza di ogni riga di input controllata

Se l'operatore g (*global*) non è accodato al comando substitute, la sostituzione agisce solo sulla prima verifica d'occorrenza di ogni riga.

Sia da riga di comando che in uno script di shell, un'operazione sed può richiedere il quoting e alcune opzioni.

```
  1 sed -e '/^$/d' $nomefile
  2 #  L'opzione -e indica che la stringa successiva deve essere interpretata
come
  3 #+ un'istruzione di editing.
  4 #  (se a "sed" viene passata un'unica istruzione, "-e" è facoltativo.)
  5 #  Il quoting "forte" ('') protegge i caratteri speciali delle ER, presenti
  6 #+ nell'istruzione, dalla reinterpretazione da parte dello script.
  7 # (Questo riserva solo a sed l'espansione delle ER.)
  8 #
  9 # Agisce sul testo del file $nomefile.
```

In certi casi, un comando di editing **sed** non funziona in presenza degli apici singoli.

```
  1 nomefile=file1.txt
  2 modello=INIZIO
  3
  4   sed "/^$modello/d" "$nomefile"  # Funziona come indicato.
  5 # sed '/^$modello/d' "$nomefile"    dà risultati imprevisti.
  6 #        In questo esempio, il quoting forte (' ... '),
  7 #+       impedisce a "$modello" di espandersi a "INIZIO".
```

Sed utilizza l'opzione -e per indicare che la stringa che segue è un'istruzione, o una serie di istruzioni. Se la stringa contiene una singola istruzione, allora questa opzione può essere omessa.

```
  1 sed -n '/xzy/p' $nomefile
  2 #  L'opzione -n indica a sed di visualizzare solo quelle righe che
verificano
  3 #+ il modello.
  4 #  Altrimenti verrebbero visualizzate tutte le righe dell'input.
  5 #  L'opzione -e, in questo caso, non sarebbe necessaria perché vi è una sola
  6 #+ istruzione di editing.
```

Tabella C-2. Esempi di operatori sed

Notazione	Effetto
8d	Cancella l'ottava riga dell'input.
/^$/d	Cancella tutte le righe vuote.
1,/^$/d	Cancella dall'inizio dell'input fino alla prima riga vuota compresa.
/Jones/p	Visualizza solo le righe in cui è presente "Jones" (con l'opzione -n).
s/Windows/Linux/	Sostituisce con "Linux" la prima occorrenza di "Windows" trovata in ogni

Notazione	Effetto
	riga dell'input.
`s/BSOD/stabilità/g`	Sostituisce con "stabilità" tutte le occorrenze di "BSOD" trovate in ogni riga dell'input.
`s/ *$//`	Cancella tutti gli spazi che si trovano alla fine di ogni riga.
`s/00*/0/g`	Riduce ogni sequenza consecutiva di zeri ad un unico zero.
`/GUI/d`	Cancella tutte le righe in cui è presente "GUI".
`s/GUI//g`	Cancella tutte le occorrenze di "GUI", lasciando inalterata la parte restante di ciascuna riga.

Sostituire una stringa con un'altra di lunghezza zero (nulla) equivale a cancellare quella stringa nella riga di input. Questo lascia intatta la parte restante della riga. L'espressione **`s/GUI//`** applicata alla riga

```
Le parti più importanti di ogni applicazione sono le sue GUI e gli effetti sonori
```

dà come risultato

```
Le parti più importanti di ogni applicazione sono le sue e gli effetti sonori
```

La barra inversa costringe il comando di sostituzione **sed** a continuare sulla riga successiva. L'effetto è quello di usare il carattere di *a capo* alla fine della prima riga come *stringa di sostituzione*.

```
1 s/^  */\
2 /g
```

In questo modo, tutti gli spazi che si trovano all'inizio della riga vengono sostituiti con un carattere di a capo. Il risultato finale è la sostituzione di tutte le indentazioni dei paragrafi con righe vuote poste tra gli stessi paragrafi.

Un indirizzo seguito da una, o più, operazioni può richiedere l'impiego della parentesi graffa aperta e chiusa, con un uso appropriato dei caratteri di a capo.

```
1 /[0-9A-Za-z]/,/^$/{
2 /^$/d
3 }
```

Questo cancella solo la prima di ogni serie di righe vuote. Potrebbe essere utile per effettuare la spaziatura singola di un file di testo mantenendo, però, la/e riga/he vuota/e tra i paragrafi.

La via più rapida per effettuare una spaziatura doppia di un file di testo è **`sed G nomefile`**.

Per esempi che illustrano l'uso di sed negli script di shell, vedi:

1. Esempio 34-1
2. Esempio 34-2

3. Esempio 12-3
4. Esempio A-3
5. Esempio 12-15
6. Esempio 12-23
7. Esempio A-13
8. Esempio A-18
9. Esempio 12-27
10. Esempio 10-9
11. Esempio 12-39
12. Esempio A-2
13. Esempio 12-13
14. Esempio 12-11
15. Esempio A-11
16. Esempio 17-12

Per una spiegazione più ampia di sed, si controllino i relativi riferimenti in *Bibliografia*.

Note

[1] Se non viene specificato alcun indirizzo, sed, in modo predefinito, considera *tutte* le righe

C.2. Awk

Awk è un linguaggio completo per l'elaborazione di testo, con una sintassi che ricorda quella del **C**. Sebbene possegga un'ampia serie di funzionalità e di operatori, qui ne verranno analizzati solo un paio - quelli più utili allo scripting di shell.

Awk suddivide ogni riga dell'input che gli è stato passato in *campi*. Normalmente, un campo è una stringa di caratteri consecutivi separati da spazi, anche se esistono opzioni per modificare il delimitatore. Awk, quindi, analizza e agisce su ciascun singolo campo. Questo lo rende ideale per trattare file di testo strutturati -- in particolare le tabelle -- e dati organizzati in spezzoni logici, come righe e colonne.

Negli script di shell, i segmenti di codice awk vengono racchiusi da apici singoli (quoting forte) e da parentesi graffe.

```
1 awk '{print $3}' $nomefile
2 # Visualizza allo stdout il campo nr.3 del file $nomefile.
3
4 awk '{print $1 $5 $6}' $nomefile
5 # Visualizza i campi nr.1, 5 e 6 del file $nomefile.
```

Si è appena visto il comando **print** di awk in azione. L'altra sola funzionalità di awk di cui è necessaria la spiegazione sono le variabili. Awk le tratta in modo simile a come sono gestite negli script di shell, anche se con una maggiore flessibilità.

```
1 { totale += ${numero_colonna} }
```

In questo modo si aggiunge il valore di *numero_colonna* al totale di "totale". Infine, per visualizzare "totale", vi è il comando di blocco di codice **END**, da eseguire dopo che lo script ha elaborato completamente il proprio input.

```
1 END { print totale }
```

Corrispondente ad **END**, vi è **BEGIN**, per il blocco di codice che deve essere eseguito prima che awk inizi l'elaborazione del suo input.

L'esempio seguente illustra come **awk** permetta di incrementare il numero di strumenti di veridica di testo a disposizione dello scripting di shell.

Esempio C-1. Conteggio delle occorrenze di lettera

```
 1 #! /bin/sh
 2 # letter-count.sh: Conta le occorrenze di lettere in un file di testo.
 3 #
 4 # Script di nyal (nyal@voila.fr).
 5 # Usato con il permesso dell'autore.
 6 # Ricommentato dall'autore di questo libro.
 7
 8
 9 INIT_TAB_AWK=""
10 # Parametro per inizializzare lo script awk.
11 conteggio=0
12 FILE_INDICATO=$1
13
14 E_ERR_PARAM=65
15
16 utilizzo ()
17 {
18     echo "Utilizzo: letter-count.sh file lettere" 2>&1
19     # Per esempio:   ./letter-count.sh nomefile.txt a b c
20     exit $E_ERR_PARAM # Parametri passati allo script insufficienti.
21 }
22
23 if [ ! -f "$1" ] ; then
24     echo "$1: File inesistente." 2>&1
25     utilizzo              # Visualizza il messaggio di utilizzo ed esce.
26 fi
27
28 if [ -z "$2" ] ; then
29     echo "$2: Non è stata specificata nessuna lettera." 2>&1
30     utilizzo
31 fi
32
33 shift                     # Le lettere sono state specificate.
34 for lettera in `echo $@`   # Per ognuna . . .
35   do
36   INIT_TAB_AWK="$INIT_TAB_AWK tab_search[${conteggio}] = \"$lettera\";\
37    final_tab[${conteggio}] = 0; "
38   # Passato come parametro al successivo script awk.
39   conteggio=`expr $conteggio + 1`
40 done
41
42 # DEBUGGING:
43 # echo $INIT_TAB_AWK;
44
45 cat $FILE_INDICATO |
```

```
 46 # Il file viene collegato, per mezzo di una pipe, al seguente script awk.
 47
 48 # ------------------------------------------------------------------------
------
 49 awk -v tab_search=0 -v final_tab=0 -v tab=0 -v nb_letter=0 -v chara=0 -v
chara2=0 \
 50 "BEGIN { $INIT_TAB_AWK } \
 51 { split(\$0, tab, \"\"); \
 52 for (chara in tab) \
 53 { for (chara2 in tab_search) \
 54 { if (tab_search[chara2] == tab[chara]) { final_tab[chara2]++ } } } } \
 55 END { for (chara in final_tab) \
 56 { print tab_search[chara] \" => \" final_tab[chara] } }"
 57 # ------------------------------------------------------------------------
------
 58 #  Niente di così complicato, solo . . .
 59 #+ cicli for, costrutti if e un paio di funzioni specializzate.
 60
 61 exit $?
```

Per dimostrazioni più semplici dell'uso di awk negli script di shell, vedi:

1. Esempio 11-11
2. Esempio 16-8
3. Esempio 12-27
4. Esempio 34-3
5. Esempio 9-22
6. Esempio 11-17
7. Esempio 28-2
8. Esempio 28-3
9. Esempio 10-3
10. Esempio 12-50
11. Esempio 9-27
12. Esempio 12-4
13. Esempio 9-12
14. Esempio 34-13
15. Esempio 10-8

Questo è tutto, per quanto riguarda awk, ma vi sono moltissime altre cose da imparare. Si vedano i relativi riferimenti in *Bibliografia*.

Appendice D. Codici di Exit con significati speciali

Tabella D-1. Codici di Exit "riservati"

Numero di codice Exit	Significato	Esempio	Commenti
1	indica errori generici	let "var1 = 1/0"	errori vari, come "divisione per zero"
2	uso scorretto dei builtin di		Si vede raramente, sostituito

Numero di codice Exit	Significato	Esempio	Commenti
	shell, secondo la documentazione Bash		solitamente dal codice di exit 1
126	il comando invocato non può essere eseguito		problemi di permessi o il comando non è un eseguibile
127	"command not found"		possibili problemi con $PATH o errore di digitazione
128	argomento di exit non valido	exit 3.14159	**exit** richiede come argomento solo un intero compreso nell'intervallo 0 - 255 (vedi nota a piè di pagina)
128+n	segnale di errore fatale "n"	**kill -9** $PPID dello script	**$?** restituisce 137 (128 + 9)
130	script terminato con Control-C		Control-C invia il segnale di errore fatale 2 (130 = 128 + 2, vedi sopra)
255*	exit status fuori intervallo	exit -1	**exit** richiede come argomento solo un intero compreso nell'intervallo 0 - 255

Secondo la tabella, i codici di exit 1 - 2, 126 - 165 e 255 [1] hanno significati speciali e, quindi, si dovrebbe evitare di usarli come parametri di exit definiti dall'utente. Terminare uno script con **exit 127** sicuramente provoca della confusione nella fase di risoluzione dei problemi (si tratta dell'errore "command not found" oppure è quello definito dall'utente?). Comunque, molti script usano **exit 1** come codice di uscita di errore generico. Dal momento che il codice di exit 1 può indicare molti differenti errori, questo, se da una parte non aggiunge ulteriore ambiguità, dall'altra non è neanche molto significativo.

Vi è stato un tentativo per sistematizzare i numeri degli exit status (vedi `/usr/include/sysexits.h`), ma questo fu fatto solo per i programmatori di C e C++. Uno standard simile sarebbe stato appropriato anche per lo scripting. L'autore di questo documento propone di limitare i codici di exit definiti dall'utente all'intervallo 64 - 113 (in aggiunta a 0, per indicare successo), per conformarsi allo standard del C/C++. In questo modo, si possono assegnare 50 validi codici rendendo, di fatto, più immediata la soluzione dei problemi degli script.

Tutti i codici di exit definiti dall'utente presenti negli esempi che accompagnano questo documento sono conformi a questo standard, tranne nei casi in cui vi siano state delle circostanze tali da non permettere l'applicazione di questa regola, come in Esempio 9-2.

Eseguendo, al termine di uno script, $? da riga di comando, si otterranno risultati coerenti con la precedente tabella solo con Bash o *sh*. L'esecuzione di C-shell o *tcsh* potrebbe, in alcuni casi, fornire valori diversi.

Note

[1] Valori di exit al di fuori dell'intervallo possono dar luogo a numeri di exit imprevedibili. Un valore di exit maggiore di 255 restituisce un codice di exit in modulo 256. Per esempio, **exit 3809** dà come codice di exit 225 (3809 % 256 = 225).

Appendice E. Una dettagliata introduzione all'I/O e alla redirezione I/O

scritta da Stephane Chazelas e rivista dall'autore del documento

Un comando si aspetta che siano disponibili i primi tre descrittori di file. Il primo, *fd 0* (lo standard input, `stdin`), è utilizzato per la lettura. Gli altri due (*fd 1*, `stdout` e *fd 2*, `stderr`) per la scrittura.

Ad ogni comando sono associati uno `stdin`, uno `stdout` e uno `stderr`. **`ls 2>&1`** trasforma temporaneamente lo `stderr` del comando **ls** in un'unica "risorsa", lo `stdout` della shell.

Per convenzione, un comando legge il proprio input da fd 0 (`stdin`), visualizza l'output in fd 1 (`stdout`) e i messaggi d'errore in fd 2 (`stderr`). Se uno di questi descrittori di file non è aperto, si possono riscontrare dei problemi:

```
bash$ cat /etc/passwd >&-
cat: standard output: Bad file descriptor
```

Ad esempio, quando viene posto in esecuzione **xterm**, come prima cosa questo inizializza se stesso. Prima di mettere in esecuzione la shell dell'utente, **xterm** apre per tre volte il dispositivo di terminale (/dev/pts/<n> o qualcosa di analogo).

A questo punto Bash eredita questi tre descrittori di file, a loro volta ereditati da ogni comando (processo figlio) messo in esecuzione da Bash, tranne nel caso in cui il comando viene rediretto. Redirezione vuol dire riassegnare uno dei descrittori di file a un altro file (o ad una pipe, o ad altro che lo consenta). I descrittori di file possono essere riassegnati localmente (per un comando, un gruppo di comandi, una subshell, if o case, cicli for o while...), oppure globalmente, per l'intera shell (usando exec).

`ls > /dev/null` esegue **ls** con il suo fd 1 connesso a `/dev/null`.

```
bash$ lsof -a -p $$ -d0,1,2
COMMAND PID     USER   FD   TYPE DEVICE SIZE NODE NAME
 bash    363 bozo        0u   CHR  136,1         3 /dev/pts/1
 bash    363 bozo        1u   CHR  136,1         3 /dev/pts/1
 bash    363 bozo        2u   CHR  136,1         3 /dev/pts/1


bash$ exec 2> /dev/null
bash$ lsof -a -p $$ -d0,1,2
COMMAND PID     USER   FD   TYPE DEVICE SIZE NODE NAME
 bash    371 bozo        0u   CHR  136,1         3 /dev/pts/1
 bash    371 bozo        1u   CHR  136,1         3 /dev/pts/1
 bash    371 bozo        2w   CHR    1,3       120 /dev/null


bash$ bash -c 'lsof -a -p $$ -d0,1,2' | cat
COMMAND PID USER   FD   TYPE DEVICE SIZE NODE NAME
 lsof    379 root    0u   CHR  136,1         3 /dev/pts/1
 lsof    379 root    1w  FIFO    0,0      7118 pipe
 lsof    379 root    2u   CHR  136,1         3 /dev/pts/1
```

```
bash$ echo "$(bash -c 'lsof -a -p $$ -d0,1,2' 2>&1)"
COMMAND PID USER   FD   TYPE DEVICE SIZE NODE NAME
 lsof    426 root    0u   CHR  136,1         3 /dev/pts/1
 lsof    426 root    1w  FIFO    0,0      7520 pipe
 lsof    426 root    2w  FIFO    0,0      7520 pipe
```

Questo funziona per tipi differenti di redirezione.

Esercizio: Si analizzi lo script seguente.

```
 1 #! /usr/bin/env bash
 2 mkfifo /tmp/fifo1 /tmp/fifo2
 3 while read a; do echo "FIFO1: $a"; done < /tmp/fifo1 &
 4 exec 7> /tmp/fifo1
 5 exec 8> >(while read a; do echo "FD8: $a, to fd7"; done >&7)
 6 exec 3>&1
 7 (
 8  (
 9   (
10    while read a; do echo "FIFO2: $a"; done < /tmp/fifo2 | tee /dev/stderr \
11                     | tee /dev/fd/4 | tee /dev/fd/5 | tee /dev/fd/6 >&7 &
12    exec 3> /tmp/fifo2
13
14    echo 1st, allo stdout
15    sleep 1
16    echo 2nd, allo stderr >&2
17    sleep 1
18    echo 3rd, a fd 3 >&3
19    sleep 1
20    echo 4th, a fd 4 >&4
21    sleep 1
22    echo 5th, to fd 5 >&5
23    sleep 1
24    echo 6th, tramite una pipe | sed 's/.*/PIPE: &, to fd 5/' >&5
25    sleep 1
26    echo 7th, a fd 6 >&6
27    sleep 1
28    echo 8th, a fd 7 >&7
29    sleep 1
30    echo 9th, a fd 8 >&8
31
32   ) 4>&1 >&3 3>&- | while read a; do echo "FD4: $a"; done 1>&3 5>&- 6>&-
33  ) 5>&1 >&3 | while read a; do echo "FD5: $a"; done 1>&3 6>&-
34 ) 6>&1 >&3 | while read a; do echo "FD6: $a"; done 3>&-
35
36
37 rm -f /tmp/fifo1 /tmp/fifo2
38
39
40 # Per ogni comando e subshell, indicate il fd in uso e a cosa punta.
41
42 exit 0
```

Appendice F. Opzioni standard da riga di comando

Con il passar del tempo si è andato evolvendo uno standard non ben definito riguardante i significati delle opzioni da riga di comando. Le utility GNU, comunque, si comformano ad uno "standard" più rigoroso rispetto alle vecchie utility UNIX.

Tradizionalmente, le opzioni UNIX sono formate da un trattino seguito da una o più lettere minuscole. Le utility GNU hanno aggiunto il doppio trattino seguito da una parola, semplice o composta.

Le due forme maggiormente accettate sono:

- `-h`

  `--help`

  Aiuto: fornisce informazioni sull'utilizzo ed esce.

- `-v`

  `--version`

  Versione: mostra la versione del programma ed esce.

Altre opzioni d'uso corrente sono:

- `-a`

  `--all`

  Tutto: mostra *tutte* le informazioni o agisce su *tutti* gli argomenti.

- `-l`

  `--list`

  Lista: elenca semplicemente i file o gli argomenti, senza intraprendere nessun'altra azione.

- `-o`

  Visualizza (output) il nome del file

- `-q`

  `--quiet`

  Quiete: sopprime lo `stdout`.

- `-r`

`-R`

`--recursive`

Ricorsivo: agisce ricorsivamente (sul contenuto della directory).

- `-v`

  `--verbose`

  Dettaglio: visualizza informazioni aggiuntive allo `stdout` o allo `stderr`.

- `-z`

  `--compress`

  Comprime: applica la compressione (di solito gzip).

Tuttavia:

- In **tar** e **gawk**:

  `-f`

  `--file`

  File: seguita dal nome del file.

- In **cp**, **mv**, **rm**:

  `-f`

  `--force`

  Forza: forza la sovrascrittura del/i file di riferimento.

Molte delle utility UNIX e Linux, però, si discostano da questo "modello". Diventa quindi pericoloso *presumere* che una data opzione si comporti secondo lo standard descritto. In caso di dubbio, si controlli la pagina di manuale del comando in questione.

Una tabella completa delle opzioni raccomandate per le utility GNU è disponibile al sito http://www.gnu.org/prep/standards_19.html.

Appendice G. Importanti directory di sistema

Gli amministratori di sistema, e coloro che scrivono script riguardanti la sua amministrazione, dovrebbero avere un'intima familiarità: con le directory che seguono.

- `/bin`

  Binari (eseguibili). Programmi e utility fondamentali per il sistema (come **bash**).

- `/usr/bin` [1]

  Altri binari di sistema.

- `/usr/local/bin`

  Binari diversi specifici di una macchina particolare.

- `/sbin`

  Binari di sistema. Programmi e utility fondamentali (come **fsck**).

- `/usr/sbin`

  Altri programmi e utility per l'amministrazione del sistema.

- `/etc`

  Et cetera. Script per la configurazione generale del sistema.

- `/etc/rc.d`

  Script di boot, su Red Hat e distribuzioni Linux derivate.

- `/usr/share/doc`

  Documentazione riguardante i pacchetti installati.

- `/usr/man`

  Pagine di manuale.

- `/tmp`

  File di sistema temporanei.

- `/sys`

  Directory dei processi. Contiene informazioni e statistiche sui processi in esecuzione. Si tratta di una nuova directory aggiunta a Linux con i kernel della serie 2.6.X.

- `/var`

  File di sistema *variabili* (modificabili). È la directory "blocco per appunti" generale per i dati prodotti durante il funzionamento di una macchina Linux/Unix.

- `/var/log`

  File dei log di sistema.

- `/var/spool/mail`

Cartella per la posta utente.

- `/lib`

  Librerie di sistema.

- `/usr/lib`

  Altre librerie.

- `/boot`

  Directory inerente al *boot* del sistema. Il kernel, i collegamenti ai moduli, il system map e il boot manager risiedono qui.

   Modificare i file di questa directory può rendere impossibile il riavvio del sistema.

Note

[1] Alcuni dei primi sistemi UNIX possedevano un disco fisso di capacità limitata, ma veloce (contenente `/`, la partizione root) e un secondo disco di dimenzioni molto maggiori, ma più lento (contenente `/usr` e le altre partizioni). I programmi e le utility usate più di frequente risiedevano, quindi, sul disco più piccolo, ma veloce , in `/bin`, mentre tutte le altre su quello più lento, in `/usr/bin`.

Questo spiega, inoltre, la suddivisione tra `/sbin` e `/usr/sbin`, `/lib` e `/usr/lib`, etc

Appendice H. Localizzazione

La localizzazione è una funzionalità di Bash non documentata.

Uno script di shell localizzato visualizza il testo dell'output nella lingua che è stata definita, nel sistema, come locale. Un utente Linux di Berlino, Germania, preferirebbe gli output degli script in tedesco, mentre suo cugino di Berlin, Maryland, li vorrebbe in inglese.

Per creare uno script localizzato, che visualizzi nella lingua dell'utente tutti i messaggi (messaggi d'errore, prompt, ecc,), si usi lo schema seguente.

```
 1 #!/bin/bash
 2 # localized.sh
 3 #  Script di Stephane Chazelas,
 4 #+ modificato da Bruno Haible, corretto da Alfredo Pironti.
 5
 6 .gettext.sh
 7
 8 E_CDERROR=65
 9
10 error()
11 {
12   printf "$@" >&2
```

```
13   exit $E_CDERROR
14 }
15
16 cd $var || error "`eval_gettext \"Can\'t cd to \$var.\"`"
17 read -p "`gettext \"Enter the value: \"`" var
18 # ...
19
20
21 #  ----------------------------------------------------------------------
22 #  Commento di Alfredo Pironti:
23
24 #  Lo script è stato modificato in modo da non utilizzare la sintassi
25 #+ $"..." in favore di "`gettext \"...\"`".
26 #  Questo è corretto, ma con il nuovo programma localized.sh, i
27 #+ comandi "bash -D nomefile" e "bash --dump-po-string nomefile"
28 #+ non produrrebbero alcun risultato
29 #+ (perchè quei comandi cercherebbero solo le stringhe $"..."!
30 #  Il SOLO modo per estrarre le stringhe dal nuovo file è quello di usare
31 #+ il programma 'xgettext'. Il programma xgettext, però, ha dei problemi.
32
33 # Va notato, inoltre, quest'altro problema di 'xgettext'.
34 #
35 # Il comando:
36 #    gettext -s "I like Bash"
37 # estrae la stringa correttamente, mentre . . .
38 #    xgettext -s "I like Bash"
39 # . . . fallisce!
40 #  'xgettext' restituisce "-s" perchè
41 #+ il comando estrae semplicemente solo
42 #+ il primo argomento che incontra dopo la parola 'gettext'.
43
44
45 #  Caratteri di escape:
46 #
47 #  Per localizzare la frase
48 #     echo -e "Hello\tworld!"
49 #+ si deve usare
50 #     echo -e "`gettext \"Hello\\tworld\"`"
51 #  Il "doppio carattere di escape" prima della `t' è necessario per
52 #+ consentire a 'gettext' di cercare la stringa: 'Hello\tworld'
53 #  In questo modo gettext interpreta una `\' letteralmente)
54 #+ restituendo una stringa del tipo "Bonjour\tmonde",
55 #+ così che il comando 'echo' possa visualizzare il messaggio correttamente.
56 #
57 #  Non si deve usare
58 #     echo "`gettext -e \"Hello\tworld\"`"
59 #+ a causa del problema di xgettext spiegato prima.
60
61
62
63 # Proviamo a localizzare la seguente riga di codice:
64 #     echo "-h display help and exit"
65 #
66 # Come prima soluzione, si potrebbe fare così:
67 #     echo "`gettext \"-h display help and exit\"`"
68 #  In questo modo 'xgettext' funziona correttamente,
69 #+ mentre il programma 'gettext' avrebbe interpretato "-h" come un'opzione!
70 #
71 # Una possibile soluzione sarebbe
72 #     echo "`gettext -- \"-h display help and exit\"`"
73 #  Così 'gettext' funziona,
74 #+ mentre 'xgettext' avrebbe estratto "--", come già spiegato.
```

```
 75 #
 76 # Un espediente per localizzare la stringa è
 77 #     echo -e "`gettext \"\\0-h display help and exit\"`"
 78 #  Abbiamo aggiunto un \0 (NULL) all'inizio della frase.
 79 #  In questo modo 'gettext' funziona correttamente, come 'xgettext.'
 80 #  Inoltre, il carattere NULL non modifica il comportamento
 81 #+ del comando 'echo'.
 82 #  ------------------------------------------------------------------------
```

```
bash$ bash -D localized.sh
"Can't cd to %s."
"Enter the value: "
```

Così viene elencato tutto il testo localizzato. (L'opzione `-D` elenca le stringhe tra doppi apici, precedute da $, senza eseguire lo script.)

```
bash$ bash --dump-po-strings localized.sh
#: a:6
msgid "Can't cd to %s."
msgstr ""
#: a:7
msgid "Enter the value: "
msgstr ""
```

L'opzione di Bash `--dump-po-strings` assomiglia all'opzione `-D`, ma usa il formato gettext "po".

 Bruno Haible precisa:

> A partire da gettext-0.12.2, si raccomanda l'uso di **xgettext -o - localized.sh** invece di **bash --dump-po-strings localized.sh**, perchè **xgettext**:
>
> 1. interpreta i comandi gettext e eval_gettext (mentre bash --dump-po-strings interpreta solamente la sua deprecata sintassi $"...")
>
> 2. può togliere i commenti che il programmatore ha inserito per dare informazioni al traduttore.
>
> Questo codice di shell, quindi, non è più specifico di Bash: funziona allo stesso modo anche con Bash 1.x e altre implementazioni /bin/sh.

Ora è necessario creare un file `linguaggio.po`, per ogni lingua in cui si vuole vengano tradotti i messaggi degli script, specificando *`msgstr`*. Alfredo Pironti ha fornito l'esempio seguente:

fr.po:

```
  1 #: a:6
  2 msgid "Can't cd to $var."
  3 msgstr "Impossible de se positionner dans le répertoire $var."
  4 #: a:7
  5 msgid "Enter the value: "
  6 msgstr "Entrez la valeur : "
  7
  8 #  Le stringhe vengono fornite per mezzo di variabili, non con la sintassi
%s,
  9 #+ come nei programmi in C.
 10 #+ È una funzionalità bellissima se il programmatore ha l'accortezza
```

```
11 #+ di usare nomi di variabili che ne esplicitano il contenuto!
```

Quindi si esegue msgfmt.

```
msgfmt -o localized.sh.mo fr.po
```

Il risultante file `localized.sh.mo` va collocato nella directory `/usr/local/share/locale/fr/LC_MESSAGES` e, all'inizio dello script, si inseriscono le righe:

```
1 TEXTDOMAINDIR=/usr/local/share/locale
2 TEXTDOMAIN=localized.sh
```

Se un utente di un sistema francese dovesse eseguire lo script, otterrebbe i messaggi nella sua lingua.

Nelle versioni più vecchie di Bash, o in altre shell, la localizzazione richiede l'uso di gettext con l'opzione `-s`. In questo caso, lo script viene così modificato:

```
 1 #!/bin/bash
 2 # localized.sh
 3
 4 E_CDERROR=65
 5
 6 error() {
 7   local format=$1
 8   shift
 9   printf "$(gettext -s "$format")" "$@" >&2
10   exit $E_CDERROR
11 }
12 cd $var || error "Can't cd to %s." "$var"
13 read -p "$(gettext -s "Enter the value: ")" var
14 # ...
```

Le variabili `TEXTDOMAIN` e `TEXTDOMAINDIR` devono essere impostate ed esportate. Ciò andrebbe fatto all'interno dello script stesso.

Appendice scritta da Stephane Chazelas, con modifiche suggerite da Alfredo Pironti e da Bruno Haible, manutentore di gettext GNU

Appendice I. Cronologia dei comandi

La shell Bash dispone di strumenti da riga di comando per gestire e manipolare la *cronologia dei comandi* dell'utente. Si tratta, innanzi tutto, di una comodità, un mezzo per evitare la continua ridigitazione di comandi.

Comandi di cronologia di Bash:

1. **history**
2. **fc**

```
bash$ history
   1  mount /mnt/cdrom
   2  cd /mnt/cdrom
   3  ls
    ...
```

Le variabili interne associate a questi comandi sono:

1. $HISTCMD
2. $HISTCONTROL
3. $HISTIGNORE
4. $HISTFILE
5. $HISTFILESIZE
6. $HISTSIZE
7. $HISTTIMEFORMAT (Bash, ver. 3.0 o successive)
8. !!
9. !$
10. !#
11. !N
12. !-N
13. !STRING
14. !?STRING?
15. ^STRING^string^

Purtroppo, questi strumenti non possono essere usati negli script di Bash.

```
  1 #!/bin/bash
  2 # history.sh
  3 # Tentativo di usare il comando 'history' in uno script.
  4
  5 history
  6
  7 # Lo script non produce alcun output.
  8 # Inseriti negli script, i comandi di cronologia non funzionano.
```

```
bash$ ./history.sh
(nessun output)
```

Sul sito Advancing in the Bash Shell è presente un'ottima introduzione all'uso dei comandi di cronologia in Bash.

Appendice J. Un esempio di file `.bashrc`

Il file `~/.bashrc` determina il comportamento delle shell interattive. Un attento esame di questo file porta ad una migliore comprensione di Bash.

Emmanuel Rouat ha fornito il seguente, e molto elaborato, file `.bashrc`, scritto per un sistema Linux. Egli gradirebbe, anche, commenti ed opinioni da parte dei lettori.

Lo si studi attentamente, sapendo che si è liberi di riutilizzarne frammenti di codice e funzioni nei propri file `.bashrc` o anche negli script.

Esempio J-1. Esempio di file `.bashrc`

```
 1 #=============================================================
 2 #
 3 # PERSONAL $HOME/.bashrc FILE for bash-2.05 (or later)
 4 #
 5 # This file is read (normally) by interactive shells only.
 6 # Here is the place to define your aliases, functions and
 7 # other interactive features like your prompt.
 8 #
 9 # This file was designed (originally) for Solaris.
10 # --> Modified for Linux.
11 # This bashrc file is a bit overcrowded - remember it is just
12 # just an example. Tailor it to your needs
13 #
14 #=============================================================
15
16 # --> Comments added by HOWTO author.
17
18 #-----------------------------------
19 # Source global definitions (if any)
20 #-----------------------------------
21
22 if [ -f /etc/bashrc ]; then
23         . /etc/bashrc   # --> Read /etc/bashrc, if present.
24 fi
25
26 #--------------------------------------------------------------
27 # Automatic setting of $DISPLAY (if not set already)
28 # This works for linux and solaris - your mileage may vary....
29 #--------------------------------------------------------------
30
31 if [ -z ${DISPLAY:=""} ]; then
32     DISPLAY=$(who am i)
33     DISPLAY=${DISPLAY%%\!*}
34     if [ -n "$DISPLAY" ]; then
35         export DISPLAY=$DISPLAY:0.0
36     else
37         export DISPLAY=":0.0"  # fallback
38     fi
39 fi
40
41 #---------------
42 # Some settings
43 #---------------
44
45 set -o notify
46 set -o noclobber
47 set -o ignoreeof
48 set -o nounset
49 #set -o xtrace          # useful for debuging
50
51 shopt -s cdspell
52 shopt -s cdable_vars
53 shopt -s checkhash
54 shopt -s checkwinsize
55 shopt -s mailwarn
56 shopt -s sourcepath
57 shopt -s no_empty_cmd_completion
58 shopt -s histappend histreedit
59 shopt -s extglob        # useful for programmable completion
60
```

```
 61 #----------------------
 62 # Greeting, motd etc...
 63 #----------------------
 64
 65 # Define some colors first:
 66 red='\e[0;31m'
 67 RED='\e[1;31m'
 68 blue='\e[0;34m'
 69 BLUE='\e[1;34m'
 70 cyan='\e[0;36m'
 71 CYAN='\e[1;36m'
 72 NC='\e[0m'              # No Color
 73 # --> Nice. Has the same effect as using "ansi.sys" in DOS.
 74
 75 # Looks best on a black background.....
 76 echo -e "${CYAN}This is BASH ${RED}${BASH_VERSION%.*}${CYAN} - DISPLAY on
${RED}$DISPLAY${NC}\n"
 77 date
 78 if [ -x /usr/games/fortune ]; then
 79     /usr/games/fortune -s     # makes our day a bit more fun.... :-)
 80 fi
 81
 82 function _exit()        # function to run upon exit of shell
 83 {
 84     echo -e "${RED}Hasta la vista, baby${NC}"
 85 }
 86 trap _exit 0
 87
 88 #---------------
 89 # Shell prompt
 90 #---------------
 91
 92 function fastprompt()
 93 {
 94     unset PROMPT_COMMAND
 95     case $TERM in
 96         *term | rxvt )
 97             PS1="[\h] \W > \[\033]0;[\u@\h] \w\007\]" ;;
 98         *)
 99             PS1="[\h] \W > " ;;
100     esac
101 }
102
103 function powerprompt()
104 {
105     _powerprompt()
106     {
107         LOAD=$(uptime|sed -e "s/.*: \([^,]*\).*/\1/" -e "s/ //g")
108         TIME=$(date +%H:%M)
109     }
110
111     PROMPT_COMMAND=_powerprompt
112     case $TERM in
113         *term | rxvt  )
114             PS1="${cyan}[\$TIME \$LOAD]$NC\n[\h \#] \W > \[\033]0;[\u@\h]
\w\007\]" ;;
115         linux )
116             PS1="${cyan}[\$TIME - \$LOAD]$NC\n[\h \#] \w > " ;;
117         * )
118             PS1="[\$TIME - \$LOAD]\n[\h \#] \w > " ;;
119     esac
120 }
```

```
121
122 powerprompt     # this is the default prompt - might be slow
123                 # If too slow, use fastprompt instead....
124
125 #================================================================
126 #
127 # ALIASES AND FUNCTIONS
128 #
129 # Arguably, some functions defined here are quite big
130 # (ie 'lowercase') but my workstation has 512Meg of RAM, so .....
131 # If you want to make this file smaller, these functions can
132 # be converted into scripts.
133 #
134 # Many functions were taken (almost) straight from the bash-2.04
135 # examples.
136 #
137 #================================================================
138
139 #--------------------
140 # Personnal Aliases
141 #--------------------
142
143 alias rm='rm -i'
144 alias cp='cp -i'
145 alias mv='mv -i'
146 # -> Prevents accidentally clobbering files.
147
148 alias h='history'
149 alias j='jobs -l'
150 alias r='rlogin'
151 alias which='type -all'
152 alias ..='cd ..'
153 alias path='echo -e ${PATH//:/\\n}'
154 alias print='/usr/bin/lp -o nobanner -d $LPDEST'   # Assumes LPDEST is
defined
155 alias pjet='enscript -h -G -fCourier9 -d $LPDEST'  # Pretty-print using
enscript
156 alias background='xv -root -quit -max -rmode 5' # put a picture in the
background
157 alias vi='vim'
158 alias du='du -h'
159 alias df='df -kh'
160
161 # The 'ls' family (this assumes you use the GNU ls)
162 alias ls='ls -hF --color'  # add colors for filetype recognition
163 alias lx='ls -lXB'         # sort by extension
164 alias lk='ls -lSr'         # sort by size
165 alias la='ls -Al'          # show hidden files
166 alias lr='ls -lR'          # recursice ls
167 alias lt='ls -ltr'         # sort by date
168 alias lm='ls -al |more'           # pipe through 'more'
169 alias tree='tree -Cs'             # nice alternative to 'ls'
170
171
172 # tailoring 'less'
173 alias more='less'
174 export PAGER=less
175 export LESSCHARSET='latin1'
176 export LESSOPEN='|/usr/bin/lesspipe.sh %s 2>&-' # Use this if lesspipe.sh
exists
177 export LESS='-i -N -w  -z-4 -g -e -M -X -F -R -P%t?f%f \
178 :stdin .?pb%pb\%:?lbLine %lb:?bbByte %bb:-...'
```

```
179
180 # spelling typos - highly personnal :-)
181 alias xs='cd'
182 alias vf='cd'
183 alias moer='more'
184 alias moew='more'
185 alias kk='ll'
186
187 #---------------
188 # a few fun ones
189 #---------------
190
191 function xtitle ()
192 {
193     case $TERM in
194         *term | rxvt)
195             echo -n -e "\033]0;$*\007" ;;
196         *)  ;;
197     esac
198 }
199
200 # aliases...
201 alias top='xtitle Processes on $HOST && top'
202 alias make='xtitle Making $(basename $PWD) ; make'
203 alias ncftp="xtitle ncFTP ; ncftp"
204
205 # .. and functions
206 function man ()
207 {
208     xtitle The $(basename $1|tr -d .[:digit:]) manual
209     man -a "$*"
210 }
211
212 function ll(){ ls -l "$@"| egrep "^d" ; ls -lXB "$@" 2>&-| egrep -v
"^d|total "; }
213 function xemacs() { { command xemacs -private $* 2>&- & } && disown ;}
214 function te()  # wrapper around xemacs/gnuserv
215 {
216     if [ "$(gnuclient -batch -eval t 2>&-)" == "t" ]; then
217         gnuclient -q "$@";
218     else
219         ( xemacs "$@" & );
220     fi
221 }
222
223 #----------------------------------
224 # File & strings related functions:
225 #----------------------------------
226
227 function ff() { find . -name '*'$1'*' ; }                # find a file
228 function fe() { find . -name '*'$1'*' -exec $2 {} \; ; }  # find a file and
run $2 on it
229 function fstr() # find a string in a set of files
230 {
231     if [ "$#" -gt 2 ]; then
232         echo "Usage: fstr \"pattern\" [files] "
233         return;
234     fi
235     SMSO=$(tput smso)
236     RMSO=$(tput rmso)
237     find . -type f -name "${2:-*}" -print | xargs grep -sin "$1" | \
238 sed "s/$1/$SMSO$1$RMSO/gI"
```

```
}

function cuttail() # cut last n lines in file, 10 by default
{
    nlines=${2:-10}
    sed -n -e :a -e "1,${nlines}!{P;N;D;};N;ba" $1
}

function lowercase()  # move filenames to lowercase
{
    for file ; do
        filename=${file##*/}
        case "$filename" in
        */*) dirname==${file%/*} ;;
        *) dirname=.;;
        esac
        nf=$(echo $filename | tr A-Z a-z)
        newname="${dirname}/${nf}"
        if [ "$nf" != "$filename" ]; then
            mv "$file" "$newname"
            echo "lowercase: $file --> $newname"
        else
            echo "lowercase: $file not changed."
        fi
    done
}

function swap()         # swap 2 filenames around
{
    local TMPFILE=tmp.$$
    mv $1 $TMPFILE
    mv $2 $1
    mv $TMPFILE $2
}

#-----------------------------------
# Process/system related functions:
#-----------------------------------

function my_ps() { ps $@ -u $USER -o pid,%cpu,%mem,bsdtime,command ; }
function pp() { my_ps f | awk '!/awk/ && $0~var' var=${1:-".*"} ; }

# This function is roughly the same as 'killall' on linux
# but has no equivalent (that I know of) on Solaris
function killps()   # kill by process name
{
    local pid pname sig="-TERM"   # default signal
    if [ "$#" -lt 1 ] || [ "$#" -gt 2 ]; then
    echo "Usage: killps [-SIGNAL] pattern"
    return;
    fi
    if [ $# = 2 ]; then sig=$1 ; fi
    for pid in $(my_ps| awk '!/awk/ && $0~pat { print $1 }' pat=${!#} ) ; do
    pname=$(my_ps | awk '$1~var { print $5 }' var=$pid )
    if ask "Kill process $pid <$pname> with signal $sig?"
        then kill $sig $pid
    fi
    done
}

function my_ip() # get IP adresses
{
```

```
301     MY_IP=$(/sbin/ifconfig ppp0 | awk '/inet/ { print $2 } ' | sed -e
s/addr://)
302     MY_ISP=$(/sbin/ifconfig ppp0 | awk '/P-t-P/ { print $3 } ' | sed -e s/P-
t-P://)
303 }
304
305 function ii()   # get current host related info
306 {
307     echo -e "\nYou are logged on ${RED}$HOST"
308     echo -e "\nAdditionnal information:$NC " ; uname -a
309     echo -e "\n${RED}Users logged on:$NC " ; w -h
310     echo -e "\n${RED}Current date :$NC " ; date
311     echo -e "\n${RED}Machine stats :$NC " ; uptime
312     echo -e "\n${RED}Memory stats :$NC " ; free
313     my_ip 2>&- ;
314     echo -e "\n${RED}Local IP Address :$NC" ; echo ${MY_IP:-"Not connected"}
315     echo -e "\n${RED}ISP Address :$NC" ; echo ${MY_ISP:-"Not connected"}
316     echo
317 }
318
319
320 # Misc utilities:
321
322 function repeat()       # repeat n times command
323 {
324     local i max
325     max=$1; shift;
326     for ((i=1; i <= max ; i++)); do  # --> C-like syntax
327         eval "$@";
328     done
329 }
330
331
332 function ask()
333 {
334     echo -n "$@" '[y/n] ' ; read ans
335     case "$ans" in
336         y*|Y*) return 0 ;;
337         *) return 1 ;;
338     esac
339 }
340
341 #============================================================================
342 #
343 # PROGRAMMABLE COMPLETION - ONLY SINCE BASH-2.04
344 # (Most are taken from the bash 2.05 documentation)
345 # You will in fact need bash-2.05 for some features
346 #
347 #============================================================================
348
349 if [ "${BASH_VERSION%.*}" \< "2.05" ]; then
350     echo "You will need to upgrade to version 2.05 for programmable
completion"
351     return
352 fi
353
354 shopt -s extglob        # necessary
355 set +o nounset           # otherwise some completions will fail
356
357 complete -A hostname   rsh rcp telnet rlogin r ftp ping disk
358 complete -A command    nohup exec eval trace gdb
359 complete -A command    command type which
```

```
360 complete -A export     printenv
361 complete -A variable   export local readonly unset
362 complete -A enabled    builtin
363 complete -A alias      alias unalias
364 complete -A function   function
365 complete -A user       su mail finger
366
367 complete -A helptopic  help     # currently same as builtins
368 complete -A shopt      shopt
369 complete -A stopped -P '%' bg
370 complete -A job -P '%'     fg jobs disown
371
372 complete -A directory  mkdir rmdir
373 complete -A directory   -o default cd
374
375 complete -f -d -X '*.gz'  gzip
376 complete -f -d -X '*.bz2' bzip2
377 complete -f -o default -X '!*.gz'  gunzip
378 complete -f -o default -X '!*.bz2' bunzip2
379 complete -f -o default -X '!*.pl'  perl perl5
380 complete -f -o default -X '!*.ps'  gs ghostview ps2pdf ps2ascii
381 complete -f -o default -X '!*.dvi' dvips dvipdf xdvi dviselect dvitype
382 complete -f -o default -X '!*.pdf' acroread pdf2ps
383 complete -f -o default -X '!*.+(pdf|ps)' gv
384 complete -f -o default -X '!*.texi*' makeinfo texi2dvi texi2html texi2pdf
385 complete -f -o default -X '!*.tex' tex latex slitex
386 complete -f -o default -X '!*.lyx' lyx
387 complete -f -o default -X '!*.+(jpg|gif|xpm|png|bmp)' xv gimp
388 complete -f -o default -X '!*.mp3' mpg123
389 complete -f -o default -X '!*.ogg' ogg123
390
391
392 # This is a 'universal' completion function - it works when commands have
393 # a so-called 'long options' mode , ie: 'ls --all' instead of 'ls -a'
394 _universal_func ()
395 {
396     case "$2" in
397     -*)     ;;
398     *)      return ;;
399     esac
400
401     case "$1" in
402     \~*)    eval cmd=$1 ;;
403     *)      cmd="$1" ;;
404     esac
405     COMPREPLY=( $("$cmd" --help | sed  -e '/--/!d' -e 's/.*--\([^ ]*\).*/--
\1/'| \
406 grep ^"$2" |sort -u) )
407 }
408 complete  -o default -F _universal_func ldd wget bash id info
409
410
411 _make_targets ()
412 {
413     local mdef makef gcmd cur prev i
414
415     COMPREPLY=()
416     cur=${COMP_WORDS[COMP_CWORD]}
417     prev=${COMP_WORDS[COMP_CWORD-1]}
418
419     # if prev argument is -f, return possible filename completions.
420     # we could be a little smarter here and return matches against
```

```
421     # `makefile Makefile *.mk', whatever exists
422     case "$prev" in
423         -*f)    COMPREPLY=( $(compgen -f $cur ) ); return 0;;
424     esac
425
426     # if we want an option, return the possible posix options
427     case "$cur" in
428         -)      COMPREPLY=(-e -f -i -k -n -p -q -r -S -s -t); return 0;;
429     esac
430
431     # make reads `makefile' before `Makefile'
432     if [ -f makefile ]; then
433         mdef=makefile
434     elif [ -f Makefile ]; then
435         mdef=Makefile
436     else
437         mdef=*.mk               # local convention
438     fi
439
440     # before we scan for targets, see if a makefile name was specified
441     # with -f
442     for (( i=0; i < ${#COMP_WORDS[@]}; i++ )); do
443         if [[ ${COMP_WORDS[i]} == -*f ]]; then
444             eval makef=${COMP_WORDS[i+1]}       # eval for tilde expansion
445             break
446         fi
447     done
448
449         [ -z "$makef" ] && makef=$mdef
450
451     # if we have a partial word to complete, restrict completions to
452     # matches of that word
453     if [ -n "$2" ]; then gcmd='grep "^$2"' ; else gcmd=cat ; fi
454
455     # if we don't want to use *.mk, we can take out the cat and use
456     # test -f $makef and input redirection
457     COMPREPLY=( $(cat $makef 2>/dev/null | awk 'BEGIN {FS=":"} /^[^.#
][^=]*:/ {print $1}' | tr -s ' ' '\012' | sort -u | eval $gcmd ) )
458 }
459
460 complete -F _make_targets -X '+($*|*.[cho])' make gmake pmake
461
462 _configure_func ()
463 {
464     case "$2" in
465         -*)     ;;
466         *)      return ;;
467     esac
468
469     case "$1" in
470         \~*)    eval cmd=$1 ;;
471         *)      cmd="$1" ;;
472     esac
473
474     COMPREPLY=( $("$cmd" --help | awk '{if ($1 ~ /--.*/) print $1}' | grep
^"$2" | sort -u) )
475 }
476
477 complete -F _configure_func configure
478
479 # cvs(1) completion
480 _cvs ()
```

```
481 {
482     local cur prev
483     COMPREPLY=()
484     cur=${COMP_WORDS[COMP_CWORD]}
485     prev=${COMP_WORDS[COMP_CWORD-1]}
486
487     if [ $COMP_CWORD -eq 1 ] || [ "${prev:0:1}" = "-" ]; then
488     COMPREPLY=( $( compgen -W 'add admin checkout commit diff \
489     export history import log rdiff release remove rtag status \
490     tag update' $cur ))
491     else
492     COMPREPLY=( $( compgen -f $cur ))
493     fi
494     return 0
495 }
496 complete -F _cvs cvs
497
498
499 _killall ()
500 {
501     local cur prev
502     COMPREPLY=()
503     cur=${COMP_WORDS[COMP_CWORD]}
504
505     # get a list of processes (the first sed evaluation
506     # takes care of swapped out processes, the second
507     # takes care of getting the basename of the process)
508     COMPREPLY=( $( /usr/bin/ps -u $USER -o comm  | \
509         sed -e '1,1d' -e 's#[]\[]##g' -e 's#^.*/##'| \
510         awk '{if ($0 ~ /^'$cur'/) print $0}' ))
511
512     return 0
513 }
514
515 complete -F _killall killall killps
516
517 # Local Variables:
518 # mode:shell-script
519 # sh-shell:bash
520 # End:
```

Appendice K. Conversione dei file batch di DOS in script di shell

Un certo numero di programmatori ha imparato lo scripting su PC dove era installato il sistema operativo DOS. Anche il frammentario linguaggio dei file batch di DOS consente di scrivere delle applicazioni e degli script piuttosto potenti, anche se questo richiede un ampio impiego di espedienti e stratagemmi. Talvolta, la necessità spinge a convertire vecchi file batch di DOS in script di shell UNIX. Questa operazione, generalmente, non è difficile, dal momento che gli operatori dei file batch DOS sono in numero inferiore rispetto agli analoghi operatori dello scripting di shell.

Tabella K-1. Parole chiave / variabili / operatori dei file batch e loro equivalenti di shell

Operatore di File Batch	Corrispondente di scripting di shell	Significato
%	$	prefisso dei parametri da riga di comando
/	-	prefisso per le opzione di un comando
\	/	separatore di percorso
==	=	(uguale a) verifica di confronto di stringhe
!==!	!=	(non uguale a) verifica di confronto di stringhe
|	|	pipe
@	set +v	non visualizza il comando corrente
*	*	"carattere jolly" per nomi di file
>	>	redirezione di file (sovrascrittura)
>>	>>	redirezione di file (accodamento)
<	<	redirezione dello `stdin`
%VAR%	$VAR	variabile d'ambiente
REM	#	commento
NOT	!	nega la verifica successiva
NUL	/dev/null	"buco nero" dove seppellire l'output dei comandi
ECHO	echo	visualizzazione (molte più opzioni in Bash)
ECHO.	echo	visualizza una riga vuota
ECHO OFF	set +v	non visualizza il/i comando/i successivo/i
FOR %%VAR IN (LISTA) DO	for var in [lista]; do	ciclo "for"
:ETICHETTA	nessuno (non necessario)	etichetta
GOTO	nessuno (usa una funzione)	salta ad un altra parte dello script
PAUSE	sleep	pausa o intervallo di attesa
CHOICE	case o select	menu di scelta
IF	if	condizione if
IF EXIST *NOMEFILE*	if [-e nomefile]	verifica l'esistenza del file
IF !%N==!	if [-z "$N"]	verifica se il parametro "N" non è presente
CALL	source o . (operatore punto)	"include" un altro script
COMMAND /C	source o . (operatore punto)	"include" un altro script (uguale a CALL)
SET	export	imposta una variabile d'ambiente
SHIFT	shift	scorrimento a sinistra dell'elenco degli argomenti da riga di comando
SGN	-lt o -gt	segno (di intero)
ERRORLEVEL	$?	exit status
CON	stdin	"console" (stdin)
PRN	/dev/lp0	dispositivo di stampa (generico)
LPT1	/dev/lp0	primo dispositivo di stampa

Operatore di File Batch	Corrispondente di scripting di shell	Significato
COM1	/dev/ttyS0	prima porta seriale

Ovviamente, i file batch contengono, di solito, comandi DOS. Per una corretta conversione, anche questi devono essere sostituiti con i loro equivalenti UNIX.

Tabella K-2. Comandi DOS e loro equivalenti UNIX

Comando DOS	Corrispettivo UNIX	Effetto
ASSIGN	ln	collega file o directory
ATTRIB	chmod	cambia i permessi del file
CD	cd	cambia directory
CHDIR	cd	cambia directory
CLS	clear	pulisce lo schermo
COMP	diff, comm, cmp	confronta i file
COPY	cp	copia i file
Ctl-C	Ctl-C	interruzione (segnale)
Ctl-Z	Ctl-D	EOF (end-of-file)
DEL	rm	cancella il/i file
DELTREE	rm -rf	cancella ricorsivamente una directory
DIR	ls -l	elenca una directory
ERASE	rm	cancella il/i file
EXIT	exit	esce dal processo corrente
FC	comm, cmp	confronta i file
FIND	grep	ricerca le stringhe nei file
MD	mkdir	crea una directory
MKDIR	mkdir	crea una directory
MORE	more	filtro per l'impaginazione del testo del file
MOVE	mv	spostamento
PATH	$PATH	percorso degli eseguibili
REN	mv	rinomina (sposta)
RENAME	mv	rinomina (sposta)
RD	rmdir	cancella una directory
RMDIR	rmdir	cancella una directory
SORT	sort	ordina il file
TIME	date	visualizza l'ora di sistema
TYPE	cat	visualizza il file allo stdout
XCOPY	cp	copia (estesa) di file

In pratica, tutti gli operatori e i comandi di shell, e UNIX, possiedono molte più

Comando DOS	Corrispettivo UNIX	Effetto

opzioni e funzionalità rispetto ai loro equivalenti DOS e dei file batch. Inoltre, molti file batch di DOS si basano su utility ausiliarie, come **ask.com**, una farraginosa controparte di read.

DOS supporta una serie molto limitata e incompatibile di caratteri jolly per l'espansione dei nomi dei file, riconoscendo solo i caratteri * e ?.

Convertire un file batch di DOS in uno script di shell è, solitamente, semplice ed il risultato, molte volte, è più leggibile dell'originale.

Esempio K-1. VIEWDATA.BAT: file batch DOS

```
 1 REM VIEWDATA
 2
 3 REM ISPIRATO DA UN ESEMPIO PRESENTE IN "DOS POWERTOOLS"
 4 REM                                    DI PAUL SOMERSON
 5
 6
 7 @ECHO OFF
 8
 9 IF !%1==! GOTO VIEWDATA
10 REM  SE NON CI SONO ARGOMENTI DA RIGA DI COMANDO...
11 FIND "%1" C:\BOZO\BOOKLIST.TXT
12 GOTO EXIT0
13 REM  VISUALIZZA LA RIGA DELLA STRINGA VERIFICATA, QUINDI ESCE.
14
15 :VIEWDATA
16 TYPE C:\BOZO\BOOKLIST.TXT | MORE
17 REM  VISUALIZZA L'INTERO FILE, 1 PAGINA ALLA VOLTA.
18
19 :EXIT0
```

La conversione dello script rappresenta un miglioramento.

Esempio K-2. viewdata.sh: Script di shell risultante dalla conversione di VIEWDATA.BAT

```
 1 #!/bin/bash
 2 # Conversione di VIEWDATA.BAT in script di shell.
 3
 4 FILEDATI=/home/bozo/datafiles/book-collection.data
 5 NRARG=1
 6
 7 # @ECHO OFF       In questo caso il comando è inutile.
 8
 9 if [ $# -lt "$NRARG" ]    # IF !%1==! GOTO VIEWDATA
10 then
11   less $FILEDATI          # TYPE C:\MYDIR\BOOKLIST.TXT | MORE
12 else
13   grep "$1" $FILEDATI     # FIND "%1" C:\MYDIR\BOOKLIST.TXT
14 fi
15
16 exit 0                   # :EXIT0
17
18 # Non sono necessari GOTO, etichette, giochi di specchi e imbrogli.
19 # Il risultato della conversione è uno script breve, dolce e pulito,
20 # il che non può dirsi dell'originale.
```

In Shell Scripts on the PC, sul sito di Ted Davis, è presente un'ampia serie di manuali sull'arte, ormai fuori moda, della programmazione di file batch. È immaginabile che alcune delle sue ingegnose tecniche possano aver rilevanza anche per gli script di shell.

Appendice L. Esercizi

Sommario

L.1. Analisi di script

Si esamini lo script seguente. Lo si esegua e, quindi, si spieghi quello che fa. Si commenti lo script e lo si riscriva in modo che risulti più compatto ed elegante.

```
 1 #!/bin/bash
 2
 3 MAX=10000
 4
 5
 6   for((nr=1; nr<$MAX; nr++))
 7   do
 8
 9     let "t1 = nr % 5"
10     if [ "$t1" -ne 3 ]
11     then
12       continue
13     fi
14
15     let "t2 = nr % 7"
16     if [ "$t2" -ne 4 ]
17     then
18       continue
19     fi
20
21     let "t3 = nr % 9"
22     if [ "$t3" -ne 5 ]
23     then
24       continue
25     fi
26
27   break   # Cosa succede se si commenta questa riga? E perché?
28
29   done
30
31   echo "Numero = $nr"
32
33
34 exit 0
```

Un lettore ha inviato il seguente frammento di codice.

```
 1 while read RIGA
```

```
2 do
3   echo $RIGA
4 done < `tail -f /var/log/messages`
```

Il suo desiderio era quello di scrivere uno script che visualizzasse le modifiche del file di log di sistema `/var/log/messages`. Sfortunatamente, il precedente codice si blocca e non fa niente di utile. Perché? Si risolva il problema in modo che funzioni correttamente (Suggerimento: invece di redirigere lo `stdin` del ciclo, si provi con una pipe.)

Si analizzi l'Esempio A-11 e lo si riorganizzi in uno stile più semplice e logico. Si veda quante delle variabili in esso presenti possono essere eliminate e lo si ottimizzi per aumentarne la velocità d'esecuzione.

Si modifichi lo script in modo che accetti, come input, un qualsiasi file di testo ASCII per la sua "generazione" iniziale. Lo script dovrà leggere i primi caratteri `$ROW*$COL` ed impostare le occorrenze delle vocali come celle "vive". Suggerimento: ci si accerti di aver trasformato tutti gli spazi presenti nel file di input in caratteri di sottolineatura

L.2. Scrivere script

Per ciascuno dei compiti sotto elencati, si scriva uno script che svolga correttamente quanto richiesto.

Facili

Elenco della directory home

> Si esegua un elenco ricorsivo della directory home dell'utente e si salvino le informazioni in un file. Questo file va compresso e lo script deve visualizzare un messaggio che invita l'utente ad inserire un dischetto e a premere, successivamente, il tasto **INVIO**. Alla fine il file risulterà essere registrato sul floppy.

Modifica dei cicli for in cicli while e until

> Si sostituiscano i *cicli for* presenti in Esempio 10-1 con *cicli while*. Suggerimento: si registrino i dati in un array, quindi si passino in rassegna gli elementi dell'array stesso.
>
> Essendo "il più" già fatto, ora si convertano i cicli dell'esempio in *cicli until*.

Modifica dell'interlinea di un file di testo

> Si scriva uno script che legga ciascuna riga del file indicato e la visualizzi allo `stdout`, ma seguita da una riga bianca aggiuntiva. Questo produrrà, come risultato, un file con *interlinea doppia*.
>
> Si aggiunga il codice necessario affinché venga effettuato un controllo sui necessari argomenti che devono essere passati allo script da riga di comando (il nome di un file) e per verificare che il file esista.

Una volta certi che lo script funzioni correttamente, lo si modifichi in modo da ottenere una *interlinea tripla* del file indicato.

Infine, si scriva uno script che rimuova tutte le righe vuote dal file indicato in modo che il testo risulti composto con *interlinea singola*.

Elenco inverso

Si scriva uno script che si autovisualizzi allo `stdout`, ma in *senso inverso* (prima l'ultima riga, poi la penultima, ecc.).

Decompressione automatica di file

Dato come input un elenco di file, questo script interrogherà ciascun file (verificando l'output del comando file) per controllare quale tipo di compressione è stata ad esso applicata. Lo script, quindi, dovrà invocare automaticamente l'appropriato comando di decompressione (**gunzip**, **bunzip2**, **unzip**, **uncompress** o altro). Se, tra i file indicati, ve ne dovessero essere di non compressi, lo script dovrà visualizzare un messaggio d'avvertimento e non effettuare, su tali file, nessun'altra azione.

ID unico di sistema

Si generi un numero identificativo "unico", di sei cifre esadecimali, per il vostro computer. *Non* si usi l'inadeguato comando hostid. Suggerimento: **md5sum** **`/etc/passwd`** e quindi si scelgano le prime sei cifre dell'output.

Backup

Si archivino come "tarball" (file `*.tar.gz`) tutti i file presenti nella vostra directory home (`/home/vostro-nome`) che sono stati modificati nelle ultime 24 ore. Suggerimento: si usi find.

Numeri primi

Si visualizzino (allo stdout) tutti i numeri primi compresi tra 60000 e 63000. L'output dovrebbe essere elegantemente ordinato in colonne (suggerimento: si usi printf).

Numeri della lotteria

Un tipo di lotteria prevede l'estrazione di cinque numeri diversi nell'intervallo 1 - 50. Si scriva uno script che generi cinque numeri pseudocasuali compresi in quell'intervallo, *senza duplicazioni*. Lo script dovrà dare la possibilità di scelta tra la visualizzazione dei numeri allo `stdout` o il loro salvataggio in un file, con data e ora in cui quella particolare serie numerica è stata generata.

Intermedi

Gestione dello spazio su disco

Si elenchino, uno alla volta, tutti i file della directory `/home/nomeutente` di dimensioni maggiori di 100K. Ad ogni file elencato si dovrà dare all'utente la possibilità di scelta tra la

sua cancellazione o la sua compressione, dopo di che verrà visualizzato il file successivo. Si registrino in un file di log i nomi di tutti i file cancellati nonché la data e l'ora della cancellazione.

Cancellazione sicura

Si scriva, in forma di script, il comando per la cancellazione di "sicurezza" `srm.sh`. I file, passati allo script come argomenti da riga di comando, non verranno cancellati, ma, se non già compressi, dovranno esserlo tramite gzip (per la verifica si usi file), e quindi spostati nella directory `/home/nomeutente/trash`. Al momento dell'invocazione, lo script controllerà nella directory "trash" i file in essa presenti da più di 48 ore e li cancellerà.

Scambiare soldi

Qual'è la via più efficiente per scambiare $ 1.68, usando il minor numero di monete correntemente in circolazione (fino a 25 cents)? 6 quarti di dollaro, 1 dime (moneta da dieci centesimi di dollaro), un nickel (moneta da 5 centesimi) e tre monete da un centesimo.

Dato come input, da riga di comando, un importo arbitrario espresso in dollari e centesimi ($*.??), si calcoli come scambiarlo utilizzando il minor numero di monete. Se il vostro paese non sono gli Stati Uniti, si può utilizzare la valuta locale. Lo script dovrà verificare l'input e, quindi, trasformarlo in multipli dell'unità monetaria più piccola (centesimi o altro). Suggerimento: si dia un'occhiata a Esempio 23-7.

Equazioni quadratiche

Si risolva un'equazione "quadratica" nella forma *Ax^2 + Bx + C = 0*. Lo script dovrà avere come argomenti i coefficienti **A**, **B** e **C**, e restituire il risultato con quattro cifre decimali.

Suggerimento: si colleghino i coefficienti, con una pipe, a bc, utilizzando la ben nota formula *x = (-B +/- sqrt(B^2 - 4AC)) / 2A*.

Somma di numeri di corrispondenza

Si calcoli la somma di tutti i numeri di cinque cifre (compresi nell'intervallo 10000 - 99999) che contengono *esattamente due*, e solo due, cifre della serie seguente: { 4, 5, 6 }. All'interno dello stesso numero, queste possono ripetersi, nel qual caso la stessa cifra non può apparire più di due volte.

Alcuni esempi di numeri di corrispondenza che soddisfano il criterio più sopra enunciato sono 42057, 74638 e 89515.

Numeri fortunati

Un "numero fortunato" è quello in cui la somma, per addizioni successive, delle cifre che lo compongono dà come risultato 7. Per esempio, 62431 è un "numero fortunato" (6 + 2 + 4 + 3 + 1 = 16, 1 + 6 = 7). Si ricerchino tutti i "numeri fortunati" compresi tra 1000 e 10000.

Ordinare alfabeticamente una stringa

Si pongano in ordine alfabetico (in ordine ASCII) le lettere di una stringa arbitraria passata da riga di comando.

Verifica

Si verifichi il file `/etc/passwd` e se ne visualizzi il contenuto in un preciso, e facilmente comprensibile, formato tabellare.

Registrare le connessioni

Si scorra il file `/var/log/messages` e si crei un file ben ordinato contenete le connessioni effettuate da un dato utente con la relativa ora. Lo script deve essere eseguito da root. (Suggerimento: si ricerchi la stringa "LOGIN.")

Visualizzazione intelligente di un file dati

Alcuni programmi per database e fogli di calcolo sono soliti salvare i propri file usando, *come separatore di campo*, la virgola (CSV - comma-separated values)). Spesso, altre applicazioni hanno la necessità di accedere a questi file.

Dato un file con queste caratteristiche, nella forma:

```
1 Jones,Bill,235 S. Williams St.,Denver,CO,80221,(303) 244-7989
2 Smith,Tom,404 Polk Ave.,Los Angeles,CA,90003,(213) 879-5612
3 ...
```

si riorganizzino i dati e li si visualizzi allo `stdout` in colonne intestate e correttamente distanziate.

Giustificazione

Dato come input un testo ASCII, fornito o dallo `stdin` o da un file, si agisca sulla spaziatura delle parole di ogni riga, con giustificazione a destra, in modo che la stessa corrisponda alle dimensioni specificate dall'utente, inviando, successivamente, il risultato allo `stdout`.

Mailing List

Usando il comando mail, si scriva uno script che gestisca una semplice mailing list. Lo script dovrà spedire automaticamente, per e-mail, un'informativa mensile della società, il cui testo viene preso dal file indicato, a tutti gli indirizzi presenti nella mailing list, che lo script ricaverà da un altro file specificato.

Creare password

Si generino password di 8 caratteri (compresi negli intervalli [0-9], [A-Z], [a-z]) pseudocasuali. Ogni password dovrà contenere almeno due cifre.

Difficili

Verifica delle password

Si scriva uno script che controlli e convalidi le password. Lo scopo è quello di segnalare le password "deboli" o che possono essere facilmente indovinate.

Allo script deve essere passata una password di prova, come parametro da riga di comando. Per essere considerata valida, una password deve avere i seguenti requisiti minimi:

- Lunghezza minima di 8 caratteri
- Deve contenere almeno un carattere numerico
- Deve contenere almeno uno dei seguenti caratteri non alfabetici: @, #, $, %, &, *, +, -, =

Facoltativi:

- Eseguire un controllo di dizionario su tutte le sequenze di almeno quattro caratteri alfabetici consecutivi presenti nella password in verifica. Questo per eliminare quelle password contenenti "parole" che si possono trovare in un normale dizionario.
- Permettere allo script di controllare tutte le password presenti sul sistema. Possano risiedere, o meno, nel file `/etc/passwd`.

Questo esercizio richiede la perfetta padronanza delle Espressioni Regolari.

Log degli accessi ai file

Si crei un file di log degli accessi ai file presenti in `/etc` avvenuti nel corso della giornata. Le informazioni devono comprendere: il nome del file, il nome dell'utente, l'ora di accesso. Si dovrà anche contrassegnare quel/quei file che hanno subito delle modifiche. Questi dati dovranno essere registrati nel file di log in record ben ordinati.

Controllo dei processi

Lo script deve controllare in continuazione tutti i processi in esecuzione e annotare quanti processi figli sono stati generati da ciascun processo genitore. Se un processo genera più di cinque processi figli, allora lo script deve spedire una e-mail all'amministratore di sistema (o a root) con tutte le informazioni di maggior importanza, tra cui l'ora, il PID del processo genitore, i PID dei processi figli, etc. Lo script deve anche scrivere un rapporto in un file di log ogni dieci minuti.

Togliere i commenti

Si tolgano tutti i commenti da uno script di shell il cui nome andrà specificato da riga di comando. Si faccia attenzione a non cancellare la "riga #!".

Conversione HTML

Si converta un dato file di testo nel formato HTML. Questo script non interattivo dovrà inserire automaticamente tutti gli appropriati tag HTML nel file specificato come argomento.

Togliere i tag HTML

Si tolgano tutti i tag da un file HTML specificato, quindi lo si ricomponga in righe di dimensione compresa tra i 60 e i 75 caratteri. Si reimpostino appropriatamente i paragrafi e le spaziature dei blocchi di testo, e si convertano le tabelle HTML in quelle approssimativamente corrispondenti del formato testo.

Conversione XML

Si converta un file XML sia nel formato HTML che in formato testo.

Caccia agli spammer

Si scriva uno script che analizzi una e-mail di spam eseguendo una ricerca DNS sugli indirizzi IP presenti nell'intestazione del messaggio, per identificare i vari host così come l'ISP d'origine. Lo script dovrà reindirizzare il messaggio di spam inalterato agli ISP responsabili. Naturalmente, sarà necessario togliere *l'indirizzo IP del proprio provider* per non finire col lamentarsi con se stessi.

Se necessario, si usino gli appropriati comandi per l'analisi di rete.

Per farsi qualche idea in merito, si veda Esempio 12-34 e Esempio A-26.

Creare pagine di manuale

Si scriva uno script che automatizzi il processo di creazione delle *pagine di manuale*.

Dato un file di testo contenente informazioni da impaginare in una *pagina di manuale*, lo script dovrà leggere il file, quindi invocare gli appropriati comandi groff per visualizzare la risultante *pagina di manuale* allo `stdout`. Il file di testo deve essere strutturato in blocchi di informazioni secondo l'intestazione standard di una *pagina di manuale*, es. "NAME," "SYNOPSIS," "DESCRIPTION," ecc.

Vedi Esempio A-1.

Codice Morse

Si converta un file di testo in codice Morse. Ogni carattere del testo verrà rappresentato dal corrispondente carattere dell'alfabeto Morse, formato da punti e linee (si usi il trattino di sottolineatura), separato l'uno dall'altro da spazi. Per esempio, "script" ===> "... _._. ._. .. .__. _".

Editor esadecimale

Si esegua una visualizzazione in esadecimale di un file binario specificato come argomento. L'output dovrà avere i campi ben ordinati in forma tabellare, con il primo campo che indica l'indirizzo di memoria, ciascuno dei successivi otto campi un numero esadecimale di 4 byte e l'ultimo campo l'equivalente ASCII dei precedenti otto campi.

Simulare uno scorrimento di registro

Ispirandosi all'Esempio 26-14, si scriva uno script che simuli uno shift di registro a 64 bit, in forma di array. Si implementino le funzioni per il *caricamento* del registro, per lo

scorrimento a sinistra e per lo *scorrimento a destra*. Infine, si scriva una funzione che interpreti il contenuto del registro come caratteri ASCII di otto per otto bit.

Determinante

Si risolva una determinante 4 x 4.

Parole nascoste

Si scriva un generatore di puzzle di "parole", vale a dire uno script che celi 10 parole fornite come input in una matrice 10 x 10 di lettere casuali. Le parole possono essere inserite orizzontalmente, verticalmente o diagonalmente.

Facoltativo: si scriva uno script che *risolva* tali puzzle di parole. Per non rendere la soluzione troppo difficile, si ricerchino solo le parole orizzontali e verticali. (Suggerimento: ogni riga e colonna va considerata come un'unica stringa in cui trovare le sottostringhe.)

Anagrammare

Si trovino gli anagrammi di un input di quattro lettere. Ad esempio, gli anagrammi di *word* sono: *do or rod row word*. Come elenco di riferimento si può utilizzare `/usr/share/dict/linux.words`.

"**Parole concatenate**"

Una "catena di parole" è formata da una sequenza di parole in cui ognuna differisce da quella che la precede per una sola lettera.

Per esempio, una "catena" che va da *mark* a *vase*:

```
1 mark --> park --> part --> past --> vast --> vase
```

Si scriva uno script che risolva una "catena di parole" . Date quella iniziale e quella finale, lo script dovrà trovare ed elencare tutte quelle intermedie a formare la "catena". Si faccia attenzione a che *tutte* le parole della serie siano "valide."

Indice di comprensione

L'"indice di comprensione" di un brano di un testo, indica la difficoltà di lettura dello stesso per mezzo di un numero che corrisponde, approssimativamente, al livello di scolarizzazione. Ad esempio, un brano con indice 12 dovrebbe essere capito da tutti coloro che hanno avuto dodici anni di scolarizzazione.

La versione Gunning dell'indice di comprensione usa il seguente algoritmo.

1. Si sceglie un brano, di almeno cento parole, da un testo.
2. Si conta il numero delle frasi (la parte di frase che è stata troncata perché alla fine del brano, si calcola come se fosse intera).
3. Si calcola il numero medio di parole per frase.

   MEDIA_PAROLE = PAROLE_TOTALI / NUMERO_FRASI

4. Si conta il numero delle parole "difficili" presenti nel brano -- quelle formate da almeno tre sillabe. Questa quantità viene divisa per il totale delle parole, per ottenere la proporzione di parole difficili.

   PRO_PAROLE_DIFFICILI = PAROLE_LUNGHE / TOTALE_PAROLE

5. L'indice di comprensione Gunning risulta dalla somma delle due precedenti grandezze moltiplicata per 0.4, con arrotondamento all'intero più prossimo.

   INDICE_GUNNING = int (0.4 * (MEDIA_PAROLE + PRO_PAROLE_DIFFICILI))

Il passaggio nr. 4 è la parte di gran lunga più difficile dell'esercizio. Esistono diversi algoritmi per il conteggio delle sillabe di una parola. Una formula empirica approssimativa potrebbe prendere in considerazione il numero di lettere che compongono la parola e il modo in cui le consonanti e le vocali si alternano.

L'interpretazione restrittiva dell'indice di comprensione Gunning non considera come parole "difficili" le parole composte e i nomi propri, ma questo avrebbe reso lo script enormemente complesso.

Calcolo del PI Greco usando il metodo dell'Ago di Buffon

Il matematico francese del XVIII secolo De Buffon se n'è uscito con un esperimento insolito. Ha fatto cadere ripetutamente un ago di lunghezza "n" su un pavimento di legno, formato da assi lunghe e strette disposte parallelamente. Le linee di giunzione delle assi del pavimento, che sono tutte della stessa larghezza, si trovano, l'una dall'altra, alla distanza fissa "d". Ha annotato il numero totale delle cadute dell'ago nonché il numero di volte in cui lo stesso andava ad intersecare le giunzioni delle assi del pavimento. Il rapporto tra queste due grandezze è risultato essere un multiplo frazionario del PI Greco.

Prendendo come spunto l'Esempio 12-41, si scriva uno script che esegua una simulazione Monte Carlo dell'Ago di Buffon. Per semplificare le cose, si imposti la lunghezza dell'ago uguale alla distanza tra le giunzioni, $n = d$.

Suggerimento: in verità bisogna tenere in considerazione due variabili critiche: la distanza dal centro dell'ago alla giunzione ad esso più vicina e l'angolo formato dall'ago con quella giunzione. Per l'esecuzione dei calcoli si dovrà usare bc.

Cifrario Playfair

Si implementi in uno script il cifrario Playfair (Wheatstone).

Il cifrario Playfair codifica un testo mediante la sostituzione dei "digrammi" (gruppi di due lettere). Per consuetudine si dovrebbe usare, per la cifratura e la decodifica, una *chiave a matrice quadrata* di 5 x 5 lettere, poste in un certo ordine.

```
1    C O D E S
2    A B F G H
3    I K L M N
4    P Q R T U
5    V W X Y Z
```

```
  6
  7 Ogni lettera alfabetica appare una sola volta, con la "I" che
rappresenta
  8 anche la "J". La parola chiave "CODES", scelta arbitrariamente, viene
  9 per prima e, di seguito, tutte le lettere dell'alfabeto, ordinate da
sinistra
 10 a destra, saltando quelle che formano la parola chiave.
 11
 12 Per la cifratura, si suddivide il messaggio in chiaro in digrammi
 13 (gruppi di 2 lettere). Se un gruppo risulta formato da due lettere
 14 uguali, si cancella la seconda e si forma un nuovo gruppo. Se per
 15 l'ultimo digramma rimane una sola lettera, come seconda si usa un
 16 carattere "nullo", di solito una "X"
 17
 18 QUESTO È UN MESSAGGIO TOP SECRET
 19
 20 QU ES TO EU NM ES SA GI OT OP SE CR ET
 21
 22 Per ogni digramma vi sono tre possibilità.
 23 ------------------------------------------
 24 1) Entrambe le lettere si trovano su una stessa riga della chiave a
 25    matrice quadrata. Ciascuna lettera va sostituita con quella che si
 26    trova immediatamente alla sua destra. Se la lettera da sostituire
 27    è l'ultima della riga, si userà la prima della stessa riga.
 28
 29 oppure
 30
 31 2) Entrambe le lettere si trovano su una stessa colonna della chiave a
 32    matrice quadrata. Ciascuna lettera va sostituita con quella che si
 33    trova immediatamente al di sotto. Se la lettera da sostituire è
 34    l'ultima della colonna, si userà la prima della stessa colonna.
 35
 36 oppure
 37
 38 3) Entrambe le lettere formano gli angoli di un rettangolo all'interno
 39    della chiave a matrice quadrata. Ciascuna lettera viene sostituita
 40    con quella che si trova all'angolo opposto, ma sulla stessa riga.
 41
 42
 43 Il digramma "QU" ricade nel caso nr. 1.
 44 P Q R T U   (Riga che contiene sia "Q" che "U")
 45
 46 Q --> R
 47 U --> P     (si è tornati ad inizio riga)
 48
 49
 50 Il digramma "GI" ricade nel caso nr. 3.
 51 A B F G     (Rettangolo avente "G" e "I" agli angoli)
 52 I K L M
 53
 54 G --> A
 55 I --> M
 56

57 ================================================================
==
 58
 59 Per la decodifica del testo cifrato bisogna invertire, nei casi nr. 1
 60 e nr. 2, la procedura (per la sostituzione ci si sposta nella
 61 direzione opposta). Mentre nulla cambia per quanto riguarda il caso
 62 nr. 3.
 63
```

```
64
65 Il lavoro, ormai classico, di Helen Fouche Gaines, "Elementary
66 Cryptanalysis" (1939), fornisce un resoconto veramente dettagliato
67 sul Cifrario Playfair e sui relativi metodi di soluzione.
```

Lo script dovrà essere composto da tre sezioni principali

I. Generazione della "chiave a matrice quadrata", basata su una parola scelta dall'utente.
II. Cifratura del messaggio "in chiaro".
III. Decodifica del testo cifrato.

Lo script dovrà fare un uso intensivo di array e funzioni.

--

Si è pregati di non inviare all'autore le soluzioni degli esercizi. Vi sono modi migliori per impressionarlo con le proprie abilità, come segnalargli errori e fornirgli suggerimenti per migliorare il libro.

Appendice M. Cronologia delle revisioni

```
      Questo documento è apparso per la prima volta, come HOWTO, nella tarda
primavera
      del 2000.  Da allora ha subito numerosi aggiornamenti e revisioni.
      Non sarebbe stato possibile realizzare questo libro senza la
collaborazione della
      comunità Linux e, in modo particolare, del Linux Documentation Project.
```

Tabella M-1. Cronologia delle revisioni

Release	Data	Commenti
0.1	14 giugno 2000	Release iniziale.
0.2	30 ottobre 2000	Correzioni, aggiunta di materiale addizionale e script d'esempio.
0.3	12 febbraio 2001	Aggiornamento importante.
0.4	08 luglio 2001	Ulteriori correzioni, molto più materiale e script - una revisione completa ed un ampliamento del libro.
0.5	03 settembre 2001	Altro importante aggiornamento. Correzioni, aggiunta di materiale, riorganizzazione di capitoli e sezioni.
1.0	14 ottobre 2001	Correzioni, riorganizzazione, aggiunta di materiale. Stable release.
1.1	06 gennaio 2002	Correzioni, aggiunti materiale e script.
1.2	31 marzo 2002	Correzioni, aggiunta di materiale e script.
1.3	02 giugno	'TANGERINE' release: Piccole correzioni, molto più materiale e aggiunta

Release	Data	Commenti
	2002	di script.
1.4	16 giugno 2002	'MANGO' release: Correzione di diversi errori tipografici, altro materiale e ulteriori script.
1.5	13 luglio 2002	'PAPAYA' release: Alcune correzioni, molto più materiale e aggiunta di altri script.
1.6	29 settembre 2002	'POMEGRANATE' release: qualche correzione, altro materiale, ancora altri script aggiunti.
1.7	05 gennaio 2003	'COCONUT' release: un paio di correzioni, più materiale, ulteriori script.
1.8	10 maggio 2003	'BREADFRUIT' release: diverse correzioni, altro materiale e script.
1.9	21 giugno 2003	'PERSIMMON' release: correzioni e materiale aggiuntivo.
2.0	24 agosto 2003	'GOOSEBERRY' release: ampio aggiornamento.
2.1	14 settembre 2003	'HUCKLEBERRY' release: correzioni ed altro materiale.
2.2	31 ottobre 2003	'CRANBERRY' release: aggiornamento rilevante.
2.3	03 gennaio 2004	'STRAWBERRY' release: correzioni e aggiunta di materiale.
2.4	25 gennaio 2004	MUSKMELON release: correzioni.
2.5	15 febbbraio 2004	STARFRUIT release: correzioni e aggiunta di materiale.
2.6	15 marzo 2004	SALAL release: aggiornamento secondario.
2.7	18 aprile 2004	MULBERRY release: aggiornamento secondario.
2.8	11 luglio 2004	ELDERBERRY release: aggiornamento secondario.
3.0	03 ottobre 2004	LOGANBERRY release: aggiornamento rilevante.
3.1	14 novembre 2004	BAYBERRY release: correzioni.

Appendice N. Siti per il download

L'ultimo aggiornamento del presente documento, sotto forma di archivio "tarball" comprendente sia il sorgente SGML che il formato HTML, può essere scaricato dal sito dell'autore.

Il mirror principale per questo libro è il Linux Documentation Project, che mantiene anche molte altre guide e HOWTO.

La *ABS Guide* è reperibile anche presso Sunsite/Metalab/ibiblio.org.

Un altro sito ancora è the morethan.org ftp site.

Appendice O. Ancora da fare

- Un'indagine estesa sulle incompatibilità tra Bash e la shell Bourne classica.
- Come il precedente, ma per la shell Korn (ksh).
- Un'introduzione alla programmazione CGI con Bash.

Ecco un semplice script CGI da cui si potrebbe partire.

Esempio O-1. Visualizzare l'ambiente di un server

```
  1 #!/bin/bash
  2 # Per il vostro sito potrebbe essere necessario modificare il
percorso.
  3 # (Su alcuni server ISP, Bash potrebbe non trovarsi nella directory
solita.)
  4 # Altre directory: /usr/bin o /usr/local/bin
  5 # Può anche essere usato senza l'intestazione.
  6
  7 # test-cgi.sh
  8 # di Michael Zick
  9 # Usato con il permesso dell'autore
 10
 11
 12 # Disabilita il globbing dei nomi dei file.
 13 set -f
 14
 15 # Informa il browser di ciò che deve aspettarsi.
 16 echo Content-type: text/plain
 17 echo
 18
 19 echo CGI/1.0 rapporto dello script di verifica:
 20 echo
 21
 22 echo impostazioni d'ambiente:
 23 set
 24 echo
 25
 26 echo bash dove si trova?
 27 whereis bash
 28 echo
 29
 30
 31 echo chi siamo?
 32 echo ${BASH_VERSINFO[*]}
 33 echo
 34
 35 echo argc è $#. argv è "$*".
 36 echo
 37
 38 # Variabili d'ambiente attese da CGI/1.0.
 39
 40 echo SERVER_SOFTWARE = $SERVER_SOFTWARE
 41 echo SERVER_NAME = $SERVER_NAME
 42 echo GATEWAY_INTERFACE = $GATEWAY_INTERFACE
 43 echo SERVER_PROTOCOL = $SERVER_PROTOCOL
 44 echo SERVER_PORT = $SERVER_PORT
 45 echo REQUEST_METHOD = $REQUEST_METHOD
 46 echo HTTP_ACCEPT = "$HTTP_ACCEPT"
 47 echo PATH_INFO = "$PATH_INFO"
```

```
 48 echo PATH_TRANSLATED = "$PATH_TRANSLATED"
 49 echo SCRIPT_NAME = "$SCRIPT_NAME"
 50 echo QUERY_STRING = "$QUERY_STRING"
 51 echo REMOTE_HOST = $REMOTE_HOST
 52 echo REMOTE_ADDR = $REMOTE_ADDR
 53 echo REMOTE_USER = $REMOTE_USER
 54 echo AUTH_TYPE = $AUTH_TYPE
 55 echo CONTENT_TYPE = $CONTENT_TYPE
 56 echo CONTENT_LENGTH = $CONTENT_LENGTH
 57
 58 exit 0
 59
 60 # Here document contenente informazioni sull'utilizzo.
 61 :<<-'_test_CGI_'
 62
 63 1) Inserite lo script nella vostra directory
http://nome.dominio/cgi-bin.
 64 2) Quindi aprite http://nome.dominio/cgi-bin/test-cgi.sh.
 65
 66 _test_CGI_
```

Qualche volontario?