INTRODUZIONE ALLA PROGRAMMAZIONE IN KORN SHELL
E.M.

- DAVID KORN, BELL LABS, "THE KORN SHELL COMMAND AND PROGRAMMING LANGUAGE", PRENTICE HALL, 1989

- esecuzione:

- dalla shell (prompt $) → $primo

il file deve avere le protezioni di lettura ed esecuzione.

- oppure, chiamando la shell esplicitamente: $ksh primo

basta il permesso di lettura

- commenti nello script: carattere #

si possono mettere anche righe vuote.

- nome dello script: puo' essere un nome riservato (shell statement o comando unix) → script.ksh

- debug

- debug totale: $ksh –x scriptfile

- debug parziale

nello script : …

```
set –x      # start debug
…
set +x      # stop debug
```

- molti programmatori trovano conveniente usare la seguente scrittura:

usage = "usage: script parameteri"

non viene eseguita, ma puo' essere stampata con $grep usage nome_script

- scritture da script: $print stringa
- letture da script: $read nome $read uno due tre
- caratteri con significato particolare: $\#?[]*{}|();”’
- **quoting** (protezione): operazione mediante la quale viene mascherato il significato speciale dei caratteri.
- **quoting mediante backslash**: quello che segue immediatamente perde il suo significato particolare.

  $rm \$nome
  $rm ‘$’nome

attenzione: il backslash può quotare (proteggere) anche il significato particolare del cr.

```
$print questa \
dovrebbe essere una \
prova
```

questa dovrebbe essere una prova

- **quoting mediante apostrofi** : ‘ “
  - apostrofo singolo: tutto quello che e’ racchiuso tra apostrofi perde il suo significato particolare.

    $print ‘$x’ → scrive $x
    $print ‘ciao ‘a’ tutti’ → scrive ciao ‘a’ tutti
    $print ‘ciao “a” tutti’ → scrive ciao “a” tutti
    $rm ‘$nome’ → se nome ha caratteri speciali

  - apostrofo doppio: quasi tutti i caratteri sono quotati tranne ` " $ \

    $print “$x” → scrive il valore di x
    $print ciao a tutti
    $print “ciao a tutti”
    $print –n “ciao” → inibisce eol
    $print –n “a tutti”
    $print “\nciao \na tutti” → \n introduce una newline
    $print “\t ciao a tutti” → \t e’ un tab

Variabili

- nomi: combinazioni di lettere, numeri e _ ma non possono cominciare con un numero
- lunghezza: nessun limite
- quantita' di variabili: illimitate
- lista delle variabili: $set
- case sensitive

- valore di una variabile

il valore della var nome si recupera con la scrittura $nome

$print "x" → scrive x
$print "$y"
$print "$x" → scrive il valore
$print "y=$x"
$print "$y"

- assegnazione

1 - n=100 (senza spazi ai lati di = !!)
2 - let n=100
3 – let "n = 100"
4 – ((n = 100))
assegnazione di tra due variabili:
x=$n

scrittura di una variabile:
print "$x"
print "$usage="

attenzione agli spazi!
y = 100 ← errore
y = "nome" ← errore

uso improprio del segno $:

y=50 → ok
x=y → assegna il nome y ad x
x=$y → assegna 50 ad x
$x=$y errore

- tipi di dati
- costanti
- stringhe (default)
- interi
- array
- globali (default)
- statement typeset per definire i tipi di variabili
- dimensione array: dipende dalla versione della shell, ma tipicamente 512 o 1024
- array: solo ad una dimensione

DICHIARAZIONE DI STRINGHE

- le stringhe vengono definite non appena viene scritto il nome della variabile

```
lettera="a"
stringa_numerica="12345"
messaggio="chiamami al 3861"
print "ecco qualche stringa"
print "$lettera, $stringa_numerica, $messaggio"
```

DICHIARAZIONE DI VARIABILI INTERE

- integer variabile

```
integer a
a=100
print "a = $a"
```

- inizializzazione di variabili:

integer b=100

- dichiarazione di costanti

typeset -r nome_della_costante=valore

- dichiarazione di array: gli array vengono definiti appena viene assegnato un valore all'array.

esempio di array di stringhe:

a[0]="primo"
a[1]="secondo"
a[10]="decimo"

questo e' un array di stringhe perche' non e' dichiarato diversamente. gli elementi non dichiarati sono stringhe nulle.

esempio di array di interi:

integer b
b[1]=1
b[2]=2

read b[10]

- inizializzazione:

set -a a primo secondo terzo

STAMPA DI UN ARRAY

- valori individuali

```
print "elemento 0 = ${vettore[0]}
indice=2
print "elemento $indice = ${vettore[$indice]}
print
```

- tutto l'array

```
print "array intero = ${vettore[*]}
print
```

- l'uso delle {} va fatto solo quando si preleva un valore. esempio:

```
a[1]="primo"
print ${a[1]}
v[0]=${a[1]}
```

OPERAZIONI MATEMATICHE

- solo numeri interi (32 bit) con segno. non segnala errore se si usano variabili floating point ma esegue operazioni intere!

- operazioni ammesse:

+ - * / % << >> & ^(or esc.) |

- sintassi con doppie parentesi! esempio:

```
integer x
integer y
integer z

((z=x+y))
((z=x-y))
((z=x*y))
print " numero complessivo = $((x+y+z))"
```

- operazioni matematiche anche su variabili di tipo stringa!

- stampa di numeri negativi: opzione -r (altrimenti il segno meno (-) e' considerato una opzione!)

- priorita' delle operazioni:

    1) moltiplicazioni
    2) divisioni
    3) somme e sottrazioni

- numeri in base 2, 8, 16

typeset -i x=123
typeset -i2 y
typeset -i8 z
typeset -i16 h

h=z=y=x (conversione automatica)

- lettura da tastiera:

scrivere 2#1000 o 8#12 o 16#56

- numero di cifre:

typeset -zn variabile (usa n cifre. solo con stringhe)

esempio:
typeset -z4 numero
numero=12345
print $numero (2345)
numero=123
print $numero (0123)

PATTERN MATCHING IN KORN SHELL

- uso di 'wildecards'. dove si possono usare le wildecards?

    1) in alcuni comandi unix
    2) nello statement case
    3) nello statement for
    4) nello statement set
    5) nelle istruzioni di manipolazione stringhe

- wildecards:

1) ? → matching con 1 solo carattere

2) [$c_1c_2...c_n$] → 1 carattere di quelli specificati

    esempio:
    ca[abz] → caa, cab, caz
    ca[a-z][a-z]
    ca[0-9]
    ca[!abz]
    ca[\-\]\!\\] → ca-, ca], ca! ca\

3) * → corrisponde con tutte le stringhe

4) ?(stringa$_1$| stringa$_2$| … | stringa$_n$) → corrisponde con nessuna o con una delle stringhe specificate. esempio:

    care?(ful|less|free) → care, careful, careless, carefree
    care?([a-z]|less) → care, caret, careless
    care?(?|??) → care, carez, careto

5) @(stringa$_1$|stringa$_1$|stringa$_3$) → solo una delle stringhe.

esempio:

care@(a|b|c) → carea, careb, carec

6) *(stringa$_1$|...|stringa$_n$) → qualsiasi stringa inclusa la stringa nulla. esempio:

p*(a|b) → p, pa, pb, pab, paa, ...

7) +(stringa$_1$|...|stringa$_n$) → come *, ma senza la stringa nulla.

8) !(stringa$_1$|...|stringa$_n$) → corrisponde a tutte le stringhe fuorche' quelle specificate esempio:

care!(*.bak|*.out) → tutte le stringhe tranne i nomi che finiscono in .bak o .out

CONDIZIONI E CONFRONTI

- tra numeri → (( ))
- tra stringhe → [[ ]]
- due punti (:) → condizione true
- test su numeri: == != < > <= >=
- test su stringhe: = != > < -z (stringa nulla)
- **statement if-then-else** .

1° esempio :

```
read n1 n2
if ((n1<n2))
then
        print "$n1 minore di $n2"
else
        print "$n2 minore di $n1"
fi
```

2° esempio:

```
read n1 n2
if ((n1<n2))
then
        print "$n1 minore di $n2"
elif ((n1==n2))
then
        print "$n1 uguale a $n2"
else
        print "$n1 maggiore di $n2"
fi
```

CONFRONTO TRA STRINGHE

- attenzione: non mettere spazi!

esempio:

```
if [[$x=$y]]
then
  …
```

- uso del pattern matching

esempio:

```
print –n “scrivi una stringa”
read nome
if [[$nome=c*]]
then
        print “$nome comincia con c”
fi

if [[$nome!=*c]]
then
        print “$nome non finisce con c”
fi

if [[$nome=@[nome1|nome2|…|nomen]]]
then
        print “$nome e’ uno dei nomi dati”
fi
```

- operatori logici and, or: || &&

esempio:

```
if ((x<y)) && ((x<z))
then
        print “$x e’ minore di $y e $z”
fi
```

- **statement case-esac**

  sintassi:

```
case $nome in

        “nome1”)    print “primo caso”
                    print  ;;

        “nome2”)    print “secondo caso”
                    print  ;;

        [a-z][a-z])  print “coppia di caratteri”
                    print  ;;

        str1@[str2|str3)     print “str1str2 oppure str1str3”
                    print  ;;

        str1|str2|str3) print “str1 oppure str2 oppure str3”
                        print;;

        *)          print “caso inatteso”     ;;

esac
```

TEST SUI FILE

- elenco delle opzioni (file mode) :

-a nome → esiste?
-f nome → e' un file regolare?
-d nome → e' un direttorio?
-c nome → e' un file di caratteri?
-b nome → e' un file a blocchi?
-p nome → e' una pipe?
-s nome → e' un socket?
-l nome → e' un link ad un altro oggetto?
-s nome → e' non vuoto?

- elenco delle opzioni (protezioni):

-r nome → posso leggere?
-w nome → posso modificarlo?
-x nome → posso esguirlo?
-o nome → ne sono proprietario?
-g nome → e' il mio gruppo?

- confronto sulle date:

nome1 -nt nome2 → nome1 piu' nuovo di (newer than) nome2
nome1 -ot nome2 → nome1 piu' vecchio di (older than) nome2

nb:
- se un file non esiste, la shell considera il file esistente piu' nuovo
- se nessun file esiste, il risultato e' falso

CICLI IN KORN SHELL

- **statement while**

esempio:

```
integer n=0
while ((n<4))
do
        ((n=n+1))
done
```

- **statement until**

esempio:

```
integer n=0
until((n>4))
do
        ((n=n+1))
done
```

- **statement for**

esempio:

```
for n in 1 2 3 4
do
        print “valore di n = $n”
done

integer ris=5
for n in 10 100 1000
do
        ((ris=ris*n))
done
print “ris=$ris”
```

```
for nome in mario giuseppe vittorio
do
        print “$nome”
done
```

- statements break e continue: analogo al linguaggio c. break esce dal ciclo e continue lo riprende

- altri esempi: parsing di una stringa in parole:

```
for parola in $linea
do
        print “$parola”
done
```

- lista dei file con permesso di lettura

```
for nome in *       ← tutti gli oggetti nella directory
do
        if [[-f $nome]] && [[-r $nome]] && [[-w $nome]]
        then
                print “ il file regolare $nome puo’ essere letto e scritto”
        fi
done
```

PARAMETRI DI LINEA AD UNO SCRIPT

- possono essere:

- argomenti semplici (numeri stringhe pathname)
- opzioni (per es. –x o +x)
- redirezioni (> o <)

- parametri posizionali:

$$	→	pid
$#	→	numero dei parametri
$*	→	stringa formata da tutti i parametri
$@	→	espande gli argomenti
$0	→	nome dello script, della funzione
$1…$9,${10}…	→	parametri

- esempio d'uso di $#

```
usage="usage: $0 parametri"
if (($# > 4))
then
        print "troppi parametri"
elif    (($# == 4))
then
        print "ok"
        for i in $@
        do
                print "$i"
        done
else
        print "$usage"
fi
```

FUNZIONI IN KORN

- variabili globali: dichiarate implicitamente (stringhe) o dichiarate fuori da una funzione

- variabili locali: definite all'interno di una funzione con typedef o integer

- le variabili riservate sono globali

- e' ammessa la ricorsione

- i parametri passati ad una funzione sono recuperati con il meccanismo degli script

- uno script puo' chiamare un altro script: il passaggio parametri e' lo stesso delle funzioni

- ritorno parametri:
- con return: ritorno un valore intero di 8 byte nella variabile $? attenzione: salvare subito $? perche' e' usata da tutti i processi (comandi, script …) per il valore di ritorno
- tramite variabili globali
- tramite file
- con la scrittura a=$(nome_funzione param)

INPUT/OUTPUT

- statement di base:
- read
- print
- exec → apre e chiude streams
- operatori per la redirezione

- read var
- se non e' indicata la variabile, l'input viene messo in **reply**
- prompt: read nome?"scrivi il nome"
- il terminatore dell'ingresso e' dato dalla variabile **ifs**
1. di default, ifs=spazio|tab|return
2. esempio di ridefinizione di ifs: ifs= ".|"
- lettura di un ingresso piu' lungo di una linea → \

esempio:

```
read var
questa\
e'\
una\
stringa
```

- lettura all'interno di un ciclo:

```
while read var          # continua a leggere fino a quando scrivo ^d
do
        ((tot=tot+var))
done
print "totale = $tot"
```

- redirezione dell'input

```
totale:     $script < file
parziale:   while read var
                do
                    ((tot=tot+var))
                done < file
```

- print
- opzioni:
  \a → bell
  \b → sovrascrive
  -r → inibisce il significato del carattere backslash
  -- → come –r

- redirezioni:

```
#parsing di un file
read inp?“input file”
read out?“output file”
while read stringa
do
        for word in $stringa
        do
                print “$word”
        done
done < $inp > $out
```

- exec

apre un file per lettura e scrittura

- apertura per read: exec 8< file# 8 e’ il descrittore del file
- apertura per print: exec 8> file# idem
- chiusura del file: exec 8<&-
- lettura/scrittura: read –u8 var / print –u8 $var
- cattura dell’output ascii di un comando:

```
var=$(comando-unix)
esempio:    var=$(ls –l)
                    var=$(sort filename)
```

- lettura di un text file in una variabile stringa

```
esempio:    var=$(< filename)
```