5. La programmazione nella *bash shell*

Programmazione della shell

In questo e nei paragrafi successivi vengono descritte le modalità per scrivere **shell script**, cioè testi contenenti comandi della shell che diventano a loro volta comandi eseguibili. In particolare si fa riferimento alla programmazione nella shell standard di Linux, la **Bash shell**.

Bash è l'acronimo di *Bourne again shell* e sta a indicare che questa shell deriva direttamente, con qualche variante, dalla **shell di Bourne** che è sempre stata, fin dalle prime versioni di Unix, la shell standard del sistema operativo.
Le differenze, rispetto alle altre shell disponibili in Linux, riguardano essenzialmente il formato delle strutture di controllo utilizzate nella programmazione degli script.
La Bash shell è un interprete di linguaggio di comandi rappresentato con il nome **bash**, che si trova nella directory */bin*. Il suo *pathname* è quindi **/bin/bash**.

Per scrivere il testo con il codice degli script si può usare un editor qualsiasi di Linux: *vi*, *Emacs* oppure *KWrite*.
In particolare l'editor **KWrite**, che è già stato presentato nel capitolo precedente, visualizza con colori diversi le parole chiave del linguaggio di programmazione, le variabili, gli identificatori e i messaggi del programmatore e le righe di commenti; offre inoltre la possibilità di espandere e contrarre le strutture di controllo e i sottoprogrammi e di mostrare i numeri di riga. Questo consente di tenere sotto controllo con maggiore facilità la struttura complessiva dello script e facilità il lavoro di debug in caso di errori, soprattutto quando i programmi sono complessi.
Queste facilitazioni per il programmatore sono attivate al primo salvataggio del testo, ma possono essere impostate anche dal menu **Strumenti** di *KWrite*, con le sottoscelte successive **Evidenziazione**, **Script**, **Bash**.

Illustriamo, con un primo semplice esempio, che cosa significa programmare nella shell di Linux.
Scriviamo un testo di nome *calendario* contenente le due seguenti righe:

```
echo '********   Calendario Anno ********'
cal 2010
```

Il comando **echo** visualizza sullo schermo il messaggio delimitato dagli apici.
Il comando **cal**, come già visto in precedenza, visualizza il calendario del 2010.

Scriviamo ora il comando

```
bash calendario
```

Il testo *calendario* diventa uno **script** per la shell, cioè una sequenza di comandi da eseguire: l'output consiste nella visualizzazione della riga di intestazione seguita dal calendario del 2010.

Lo script può essere reso eseguibile cambiandone i permessi con il comando

```
chmod +x calendario
```

oppure, usando la notazione ottale,

```
chmod 755 calendario
```

Occorre anche aggiungere una nuova riga all'inizio del file, contenente

```
#!/bin/bash
```

Questa deve essere obbligatoriamente la prima riga dello script e indica alla shell il nome dell'interprete di comandi che si vuole utilizzare.
Righe successive nello script aventi il carattere **#** nella prima posizione identificano le **righe di commento**.
Dopo aver inserito la riga con l'indicazione dell'interprete dei comandi, l'utente può chiedere l'esecuzione precedente semplicemente scrivendo sulla linea comandi il nome dello script:

```
calendario
```

Per eseguire lo script, la directory corrente, indicata con il punto (directory .), deve essere aggiunta nella definizione della variabile **PATH** (che contiene l'elenco delle directory dove trovare i comandi) nel file di configurazione locale *.bashrc*, se esiste, oppure nel file di configurazione generale */etc/profile.*
Dopo questa modifica, per rendere operativo il nuovo *path* occorre dare il comando **rehash**, oppure, più semplicemente, basta sconnettersi ed effettuare una nuova connessione al *login*.

L'aggiunta della directory corrente nella ricerca dei percorsi può essere fatta anche direttamente dalla linea comandi nel seguente modo:

```
export PATH="$PATH:."
```

Il comando **export** stabilisce i valori per le variabili che vengono esportate, nel senso che vengono trasferite all'ambiente, in modo da renderle visibili ai programmi eseguiti successivamente all'interno della shell.

Occorre osservare che, per ragioni di protezione e sicurezza del sistema, l'aggiunta della directory corrente, indicata con il punto, all'interno del *path* deve essere fatta solo in situazioni di prova o di apprendimento; è opportuno comunque che sia all'interno del file locale di configurazione per l'utente e in ogni caso in coda ai percorsi già esistenti nella variabile PATH.

Se non si stabilisce questa specificazione sulla directory corrente, il comando può essere comunque eseguito facendolo precedere dall'indicazione della directory corrente stessa:

```
./calendario
```

L'attività svolta, pur essendo molto semplice, ha un'importanza rilevante: infatti l'utente ha in pratica creato un nuovo comando di nome *calendario* che può essere invocato per la sua esecuzione come accade per i comandi della shell, quali *ls*, *cat* o *rm*. In sostanza è stato ampliato l'insieme dei comandi che l'interprete è in grado di riconoscere.

Lo script *calendario* può essere successivamente modificato introducendo la possibilità di esecuzione non solo per l'anno 2010, ma per un anno qualsiasi, fornito dall'utente come argomento del comando.

Modifichiamo il testo dello script come segue:

```
#!/bin/bash
echo '********   Calendario Anno ********'
cal $1
```

Abbiamo inserito al posto di 2010 una variabile che funziona da parametro (si chiama **variabile posizionale**) e che verrà sostituita dal valore fornito dall'utente come argomento del comando. Per esempio, con la riga di comando

```
./calendario 2012
```

viene mandato sul video lo stesso output precedente, ma riferito all'anno 2012; al momento dell'esecuzione, la shell ha sostituito il parametro di $1 nello script con il valore 2012.

Come ulteriore conferma del fatto che *calendario* funziona come un qualsiasi comando di Linux, si provi a eseguire la seguente pipeline che produce l'output su carta anziché sul video:

```
./calendario 2012 | lpr
```

oppure la pipeline

```
./calendario 2012 | more
```

per visualizzare l'output del comando sul video con 24 righe per volta.

È possibile utilizzare una specie di debugger per gli script lanciando l'esecuzione con il comando

```
bash -v nome_script
```

(*v* sta per *verbose*); si ottiene come aiuto una traccia delle azioni attraverso la visualizzazione delle righe di comando una ad una durante l'esecuzione: questo consente di individuare la riga sulla quale l'interprete trova un eventuale errore.

Le variabili nella shell

Le variabili sono usate dalla shell per memorizzare e modificare stringhe di caratteri o numeri. Una variabile della shell inizia con una lettera e può contenere lettere, cifre e carattere _ (*underscore*).
Le variabili possono essere quelle definite dall'utente oppure le variabili predefinite (o di sistema) di cui si parlerà nel paragrafo successivo.
Il comando **set** visualizza le variabili inizializzate e i rispettivi valori attuali nella sessione di lavoro dell'utente nella shell.
L'assegnazione del valore a una variabile utente viene effettuato scrivendo nella riga comandi il nome della variabile e il valore assegnato, separati dal segno =, senza spazi prima o dopo il segno =.

variabile=valore

(l'assegnazione viene fatta a sinistra).

Per esempio:

```
a=3
```

crea una nuova variabile di nome *a* alla quale viene assegnato il valore 3.
Alle variabili si possono assegnare anche valori non numerici, per esempio:

```
citta=Roma
```

Se la variabile deve assumere un valore stringa formato da più parole, esse devono essere racchiuse tra apici, per esempio:

```
nome='Bianchi Vittorio'
```

Come già visto in precedenza, si possono creare nuove variabili e assegnare ad esse valori iniziali attraverso il comando **export**. Con questa modalità, la variabile definita può essere esportata, nel senso che può essere usata non soltanto dalla shell attiva, ma anche dai programmi che vengono lanciati in esecuzione a partire dalla shell stessa.

Se si vuole soltanto dichiarare una variabile senza assegnare ad essa alcun valore iniziale, si può usare il comando **declare**.
Per esempio:

```
declare media
```

Il valore di una variabile è visualizzato con il comando **echo**; la variabile è preceduta dal simbolo $ che indica alla shell di sostituire il nome della variabile con il suo valore.
Con il comando

```
echo $a
```

sul video appare come risposta il numero 3.
La definizione di una variabile viene annullata con il comando **unset**; per esempio la sequenza di comandi

```
unset a
echo $a
```

non produce alcun valore come risposta al secondo comando.

Si possono definire anche variabili di tipo **array**, racchiudendo le componenti tra parentesi tonde e separandole con uno spazio:

```
a=(1 2 18 13 15)
```

Dopo questa assegnazione gli elementi dell'array acquisiscono un ordine intrinseco con il valore dell'indice a partire da 0: a[0] = 1, a[1] = 2, a[2]=18, ecc.

Si può anche assegnare o cambiare il valore a una singola componente, il cui indice viene indicato tra parentesi quadre:

```
a[1]=7
```

La dimensione dell'array può essere modificata aggiungendo nuovi elementi all'array, per esempio:

```
a[5]=67
```

Per visualizzare il valore di una componente dell'array si usa la seguente notazione

```
echo ${a[2]}
```

per ottenere sul video il numero 18.

Usando la tastiera italiana, le parentesi graffe si scrivono con le seguenti combinazioni di tasti:

{	AltGr + 7
}	AltGr + 0

Per visualizzare tutte le componenti dell'array si deve scrivere il comando

```
echo ${a[*]}
```

Se nell'istruzione di visualizzazione si indica il nome dell'array, senza indice, si ottiene il valore della prima componente; in altre parole, il comando

```
echo $a
```

è equivalente a

```
echo ${a[0]}
```

Il comando

```
echo ${#a[*]}
```

restituisce il numero delle componenti dell'array *a*; nel caso dell'esempio iniziale, il numero 5.

Un array può essere eliminato con il comando *unset*, come già visto per le variabili di tipo semplice:

```
unset a
```

Se si specifica l'indice, viene eliminato una singola componente dell'array: per esempio, il comando

```
unset a[3]
```

elimina la quarta componente dell'array (indice = 3).

Le variabili di sistema

La shell possiede alcune variabili speciali predefinite; esse sono indicate con i simboli

```
$# $* $? $$
```

L'utente, come già visto nell'esempio del primo paragrafo, può fornire alla shell uno o più argomenti insieme al comando da eseguire. Questi argomenti vengono associati all'interno dello script ai parametri $1, $2, $3, ecc. come **variabili posizionali**, nel senso che il primo valore fornito è associato alla variabile $1, il secondo a $2 e così via.
La variabile $0 rappresenta il nome della shell oppure il nome dello shell script in esecuzione. Per esempio, con il comando

```
./calendario 2012
```

$0 vale *calendario*
$1 vale 2012

La variabile speciale **$#** rappresenta il numero dei parametri posizionali passati come argomenti allo script.
Con **$*** si rappresenta la variabile contenente tutti gli argomenti (a eccezione di $0).
Con i caratteri **$?**, seguiti dal nome del comando, si rappresenta lo stato di uscita dall'ultimo programma eseguito dalla shell: se il valore è 0 il comando si è concluso correttamente; valori diversi indicano situazioni di errore.
Con **$$** infine si rappresenta il numero identificativo (PID) del processo della shell, processo padre del comando in esecuzione.

Altre variabili predefinite sono le **variabili di sistema** che rappresentano i valori di inizializzazione dell'ambiente di lavoro per l'utente: per questo motivo ad esse viene assegnato un valore all'interno dei file di configurazione che sono eseguiti al momento della connessione.

Tutti i valori delle variabili definite possono essere visualizzati con il comando **set**.

Le variabili di sistema più comunemente usate, con il rispettivo significato, sono:

path l'elenco delle directory che contengono i comandi;

home il pathname della home directory dell'utente;

mail la directory dove si trovano le caselle postali per i messaggi di posta elettronica;

PS1 il prompt primario, cioè la sequenza di caratteri visualizzata all'inizio della riga comandi (per la Bash shell il carattere $);

PS2 il prompt secondario, cioè la sequenza di caratteri visualizzata sulle righe successive nei comandi che richiedono altri inserimenti da parte dell'utente (normalmente il carattere >);

PWD contiene il nome della *working directory*;

term il tipo di terminale su cui si sta lavorando;

shell il nome della shell assegnata all'utente (*/bin/bash* per la Bash shell, */bin/csh* per la C-shell);

HOSTTYPE Il nome del tipo di elaboratore utilizzato;

MACHTYPE architettura e sistema operativo utilizzato.

Si faccia attenzione al fatto che le variabili di sistema sono scritte con i caratteri in maiuscolo.
L'utente può visualizzare il contenuto di queste variabili con il comando *echo*; per esempio:

```
echo $PATH
```

visualizza le directory che contengono i comandi e i programmi che l'utente può utilizzare.

L'output di dati e messaggi

Nei paragrafi precedenti sono già stati presentati alcuni esempi del comando **echo**, che consente di visualizzare dati e messaggi sull'unità standard di output (video).
Per esempio:

```
echo $a
```

visualizza il valore della variabile *a*;

```
echo Linux
```

visualizza la stringa *Linux*.

Se la costante da visualizzare è di tipo stringa formata da più parole, essa deve essere racchiusa tra una coppia di apici che la delimitano. Per esempio:

```
echo 'Linux 8.1'
```

Si deve fare attenzione al differente significato che hanno i delimitatori con singoli apici e con apici doppi.
Precisamente, il messaggio delimitato da apici singoli produce in output la sequenza di caratteri del messaggio, mentre gli apici doppi producono un'**espansione** (o *sostituzione*) delle variabili (utente o di sistema) con i loro valori.
Per esempio, se *a* è una variabile definita dall'utente per rappresentare l'area di una figura geometrica che assume il valore 36, il comando

```
echo 'area figura = $a'
```

(con gli apici semplici) produce in output la stringa

```
area figura = $a
```

Invece il comando

```
echo "area figura = $a"
```

(con gli apici doppi) visualizza in output il valore dell'area

```
area figura = 36
```

La sequenza di caratteri *$a* è stata sostituita dal valore della variabile *a*.
Come è evidente, tutto ciò consente di presentare i dati di output all'interno degli script della shell in modo efficace, aggiungendo messaggi di spiegazione dei risultati.
Gli output possono essere ottenuti con una formattazione attraverso il comando **printf**, che ha una sintassi simile all'analogo comando del linguaggio C.

Un'altra considerazione riguarda l'utilizzo del carattere ****(*backslash*). Abbiamo già visto, nell'unità di apprendimento precedente a proposito del comando *grep*, che questo carattere viene usato per rappresentare il carattere **escape** e serve a mantenere il significato letterale del carattere successivo, cioè a evitare che venga interpretato come **metacarattere**.
Per esempio, se *prezzo* è una variabile dichiarata dall'utente e possiede il valore 18, volendo visualizzare il prezzo di un libro e il suo titolo racchiuso tra virgolette, si deve scrivere il seguente comando:

```
echo "Il libro \"La Divina Commedia\" costa $prezzo euro"
```

Il carattere di *escape* davanti alle virgolette del titolo consente di mantenere il significato letterale ai doppi apici. L'output è:

```
Il libro "La Divina Commedia" costa 18 euro
```

Particolari combinazioni del carattere \ con altri caratteri, dette **sequenze di escape**, assumono significati speciali quando vengono usate con il comando *echo*. Esse risultano particolarmente utili per formattare i dati e i messaggi da visualizzare.

**** inserisce la barra obliqua inversa (\)
\a inserisce il carattere BEL (avvisatore acustico)
\b inserisce il carattere BS (*backspace*)
\c impedisce il ritorno a capo per i messaggi di due *echo* successive
\f inserisce il carattere FF (*formfeed*)
\n inserisce il carattere LF (*linefeed*)
\r inserisce il carattere CR (*carriage return*)
\t inserisce una tabulazione normale (HT)
\v inserisce una tabulazione verticale (VT)
\nnn inserisce il carattere corrispondente al codice *nnn* espresso in ottale.

Per rendere attive le sequenze di *escape*, il comando *echo* deve essere usato con l'opzione **–e**.
Per esempio, lo shell script seguente

```
#!/bin/bash
echo-e "prima riga \v12 \v20 \nseconda riga"
```

produce il seguente output:

```
prima riga
                12
                    20
seconda riga
```

Il carattere \ può anche essere utilizzato per indicare la **continuazione di un messaggio** nella riga successiva: il carattere va posto alla fine della prima riga e deve essere l'ultimo carattere della riga.
Per esempio, il seguente script

```
#!/bin/bash
echo  "Torino Milano Venezia Bologna \
Firenze Roma Napoli Palermo Cagliari"
```

produce in output la seguente riga:

```
Torino Milano Venezia Bologna Firenze Roma Napoli Palermo Cagliari
```

Le substitution

Con il termine **substitution** si intende l'operazione con la quale la shell sostituisce o espande certi simboli con altri simboli o valori ad essi associati.
La shell consente cinque tipi di substitution.

• variable substitution

La shell mantiene un elenco di variabili alle quali è associata una lista di zero o più valori: la *variable substitution* permette di sostituire alle variabili il corrispondente valore mediante il carattere speciale $.
Questo tipo di sostituzione è già stato illustrato con molti esempi nei paragrafi precedenti.

• history substitution

La *bash shell* conserva all'interno della home directory dell'utente, in un file speciale denominato **.bash_history**, l'elenco dei comandi più recenti scritti dall'utente sulla linea dei comandi. Il numero massimo di comandi che sono conservati è scritto nella variabile di sistema **HISTSIZE**: il suo valore di default è 1000. Tuttavia questo valore può essere modificato dall'utente, assegnando un nuovo valore alla variabile di sistema *HISTSIZE*: per esempio:

```
HISTSIZE=30
```

ordina alla shell di mantenere in memoria le 30 righe dei comandi più recenti forniti dall'utente. Questa assegnazione può essere resa permanente inserendola nel file di inizializzazione dell'ambiente di lavoro (*.bashrc*) che viene letto al momento della connessione.
Il comando **history** elenca gli ultimi comandi inseriti preceduti da un numero d'ordine con tante linee quanto è il valore assegnato alla variabile *HISTSIZE*.
Si possono poi usare speciali metacaratteri per rieseguire righe di comandi formulate in precedenza senza riscriverle sulla tastiera; questo è particolarmente utile per righe contenenti comandi con sintassi complesse o lunghe pipeline.
Per esempio se sulla linea comandi si scrive

```
!n
```

si può eseguire di nuovo un comando precedente sostituendo a *n* dopo il punto esclamativo il numero del comando nell'elenco ottenuto con il comando *history*.

Nelle versioni più recenti dei sistemi operativi il comando *history* si realizza in pratica con l'uso dei tasti freccia sulla tastiera, con i quali è possibile rivedere, modificare e riattivare i comandi scritti in precedenza.

• alias substitution

Il comando **alias** della shell consente di assegnare un nuovo nome più semplice da ricordare e da scrivere a righe di comandi di uso frequente o con sintassi complessa.
La sintassi del comando è:

```
alias nome=comando
```

dove *nome* indica la stringa di definizione dell'alias e *comando* indica il comando o la linea di comandi che è rappresentata con l'alias.
Per esempio:

```
alias h=history
```

permette all'utente di attivare il comando *history* scrivendo la sola lettera *h*.
La shell opera il meccanismo di *alias substitution* sostituendo alla lettera *h* il comando *history*, quando la lettera *h* è scritta come comando accanto al prompt.

Se il comando è complesso oppure è costituito da una *pipeline*, esso va racchiuso tra apici. Per esempio:

```
alias contaf='ls -l | wc -l'
```

definisce un nome alias *contaf* per la *pipeline* che restituisce il numero dei file presenti nella directory corrente.

Le definizioni di alias possono essere rimosse con il comando **unalias**; per esempio:

```
unalias contaf
```

elimina la precedente definizione dell'alias.
Le definizioni degli alias restano valide fino al logout dell'utente. Se si ritiene che alcune definizioni di alias siano particolarmente utili nel lavoro con Linux, occorre inserire i relativi comandi *alias* nei file di inizializzazione dell'ambiente di lavoro per l'utente (*.bashrc*) oppure nel file di inizializzazione generale */etc/profile.*
Il comando *alias* senza argomenti visualizza le definizioni di tutti gli alias attualmente disponibili nella shell:

```
alias
```

• filename substitution

Con questo tipo di substitution si intende l'operazione con la quale la shell utilizza alcuni metacaratteri per espandere i *pathname* di file e directory.
I caratteri con le relative espansioni sono:

***** (asterisco) una sequenza qualsiasi di zero o più caratteri; per esempio:

```
ls -l a*
```

lista i nomi dei file di nome *a* o che iniziano con la lettera *a* (minuscola).

? (punto interrogativo) un singolo carattere qualsiasi; per esempio:

```
ls ? G?
```

lista i nomi dei file lunghi un solo carattere oppure lunghi due caratteri con la lettera *G* (maiuscola) in prima posizione.

[] (parentesi quadre) un insieme di caratteri ognuno dei quali può comparire come singolo carattere in quella posizione; per esempio:

```
cat arch[123]
```

visualizza il contenuto dei file *arch1*, *arch2* e *arch3*.

– (trattino all'interno delle parentesi quadre) un range di valori compresi tra quelli specificati; per esempio:

```
cat arch[1-5]
```

equivale a

```
cat arch[12345]
```

~ (*tilde*) il *pathname* della home directory dell'utente; è particolarmente utile quando ci si trova in una working directory diversa dalla home directory e si vuole fare riferimento ai file della propria home directory; per esempio:

```
cp ~/arch1 .
```

copia il file *arch1*, che si trova nella home directory, all'interno della working directory corrente (indicata con il punto).

Il carattere ~, nella tastiera italiana, si ottiene con la combinazione di tasti AltGr + ì (in alto a destra, vicino al tasto *backspace*).

• command substitution

La shell sostituisce a un comando o a una pipeline il valore da essi restituito; il comando o la pipeline vanno delimitati con una coppia di apici rovesciati ` `.

Per esempio:

```
adesso=`date | cut -c12-20`
```

(senza spazi prima e dopo il segno uguale), assegna l'ora corrente di sistema alla variabile *adesso*: l'output della pipeline tra apici rovesciati corrisponde all'ora attuale che, attraverso la *command substitution*, diventa il valore della variabile *adesso*.

Il carattere ` (apice rovesciato) si ottiene nella tastiera italiana con la combinazione di testi AltGr + ' (apice semplice).

Con le due righe di comandi

```
spazio=`ls -l arch1.dat | cut -c29-32`
echo $spazio
```

si ottiene sul video il valore assegnato alla variabile *spazio*, cioè il numero di byte occupati dal file *arch1.dat;* la prima riga rappresenta un esempio di *command substitution*, la seconda un esempio di *variable substitution*.

Come già detto nel capitolo precedente, la posizione dei caratteri da estrarre può variare a seconda delle directory del filesystem.

La *command substitution* può essere anche rappresentata utilizzando il simbolo $ seguito da una coppia di parentesi tonde che racchiudono il comando o la pipeline. Per esempio, la sequenza di comandi precedente per calcolare il numero di byte occupati da un file si può riscrivere in questo modo:

```
spazio=$(ls -l arch1.dat | cut -c29-32)
echo $spazio
```

Questa notazione è derivata dalla *Korn shell* di Unix.

Una notazione analoga è utilizzata per rappresentare l'**espansione di espressioni aritmetiche**. In questi casi occorre usare le doppie parentesi tonde.

Per esempio:

```
echo $((120+40))
```

visualizza il valore 160;

```
echo $((a+b))
```

visualizza il valore della somma delle variabili *a* e *b*.

Le doppie parentesi tonde possono essere sostituite in forma abbreviata dalle parentesi quadre:

```
echo $[a+b]
```

MATERIALI ON LINE

Gli operatori aritmetici

Per eseguire un'espressione di calcolo, memorizzando il risultato in una variabile, si usa il comando **let.** Con la seguente sequenza di comandi inserita in uno script:

```
#!/bin/bash
a=1
let b=a+1
echo $b
```

si ottiene come output il numero 2.
Il seguente shell script calcola la somma di due numeri:

```
#!/bin/bash
a=3
b=5
let som=a+b
echo $som
```

Nella versione seguente la somma è calcolata su due numeri forniti da tastiera come argomenti dello script; si devono introdurre nello script due variabili posizionali:

```
#!/bin/bash
let som=$1+$2
echo $som
```

Gli **operatori aritmetici** che si possono usare nella shell sono:

–op	inverte il segno dell'operando
op1 + op2	somma i due operandi
op1 – op2	sottrae dal primo il secondo operando
op1 * op2	moltiplica i due operandi
op1 / op2	divide il primo operando per il secondo
op1 % op2	restituisce il modulo, cioè il resto della divisione tra il primo e il secondo operando
op1 += op2	equivale a op1 = op1 + op2
op1 –= op2	equivale a op1 = op1 – op2
op1 /= op2	equivale a op1 = op1 / op2
op1 *= op2	equivale a op1 = op1 * op2
op1 %= op2	equivale a op1 = op1 % op2.

Per esempio, il comando

```
let b+=7
```

significa *somma 7 al contenuto di b e memorizza il risultato in b.*
Analogamente

```
let b-=7
```

significa *sottrai 7 dal contenuto di b e memorizza il risultato in b.*

Espressioni condizionali

Nella Bash shell si possono costruire condizioni che controllano lo stato dei file, dette anche **primitive di test sui file.**

Sono costituite da una lettera preceduta dal trattino. La struttura generale della condizione è

```
-l nomefile
```

dove la lettera *l* va sostituita con una delle seguenti, corrispondenti ai principali controlli:

r permesso in lettura
w permesso in scrittura
x permesso di esecuzione
e esistenza del file
o proprietà sul file
z dimensione zero per il file
s dimensione maggiore di zero per il file
f file ordinario
d directory.

La condizione

```
-e arch1
```

vale vero (uguale a 1) se il file *arch1* esiste; vale falso (uguale a 0) se il file non esiste.

Altre condizioni riguardano il confronto tra due file:

file1 **-nt** file2 vale vero se il primo file ha la data di modifica più recente
file1 **-ot** file2 vale vero se il primo file ha la data di modifica più vecchia
file1 **-et** file2 vale vero se i due nomi di file corrispondono allo stesso file fisico (cioè allo stesso *inode)*.

La shell possiede anche operatori per costruire espressioni condizionali riguardanti le **stringhe**:

-z stringa vale vero se la lunghezza della stringa è zero
-n stringa vale vero se la lunghezza della stringa è diversa da zero
stringa1 **=** stringa2 stringhe uguali
stringa1 **!=** stringa2 stringhe diverse
stringa1 **<** stringa2 la prima stringa precede la seconda nell'ordine lessicografico
stringa1 **>** stringa2 la prima stringa segue la seconda nell'ordine lessicografico.

Gli **operatori di confronto tra numeri** consentono di costruire espressioni condizionali per operandi di tipo numerico:

op1 **-eq** op2 vale vero se gli operandi sono uguali (equivalente a **=)**
op1 **-ne** op2 vale vero se gli operandi sono diversi (equivalente a **!=**)
op1 **-lt** op2 vale vero se il primo operando è minore del secondo (equivalente a **<**)
op1 **-le** op2 vale vero se il primo operando è minore o uguale al secondo (equivalente a **<=**)
op1 **-gt** op2 vale vero se il primo operando è maggiore del secondo (equivalente a **>**)
op1 **-ge** op2 vale vero se il primo operando è maggiore o uguale al secondo (equivalente a **>=**).

MATERIALI ON LINE

Gli **operatori booleani** seguenti possono essere usati per costruire espressioni condizionali complesse:

L'operatore NOT si indica con **!**
! espressione — rappresenta la negazione logica dell'espressione

L'operatore AND si indica con **-a**
espressione1 -a espressione2 — produce un'espressione vera se entrambe le espressioni hanno valore logico vero

L'operatore OR si indica con **-o**
espressione1 -o espressione2 — produce un'espressione vera se almeno una delle due espressioni ha valore logico vero.

Le strutture di selezione e di ripetizione

La struttura di **selezione** è realizzata per mezzo dei comandi **if**, **then**, **else** e **fi**. La sintassi più semplice è la seguente:

```
if [ condizione ]
 then
   comando
fi
```

La struttura si chiude con la parola *fi* che è la parola *if* scritta alla rovescia.
L'espressione condizionale deve essere scritta lasciando uno spazio dopo la parentesi quadra aperta e uno spazio prima della parentesi quadra chiusa.
L'uso delle parentesi quadre per rappresentare un'espressione condizionale è una forma abbreviata del comando **test**. In sostanza la precedente struttura di selezione si potrebbe anche scrivere in questo modo:

```
if test condizione
 then
   comando
fi
```

Per esempio, usando la primitiva di test sul file vista nel paragrafo precedente, si può scrivere:

```
#!/bin/bash
if [ -e arch1 ]
then
  echo 'esiste'
fi
```

Si faccia attenzione agli spazi dopo la parentesi quadra aperta e prima della parentesi quadra chiusa.
La struttura di selezione a due vie si rappresenta con

```
if [ condizione ]
 then
   comando1
 else
    comando2
fi
```

Nel formato completo la sintassi si presenta in questo modo:

```
if [ condizione ]
 then
    comandi1
elif [ condizione2 ]
   then
    comandi2
   else
    comandi3
fi
```

La parola **elif** rappresenta la selezione annidata *else if*. I comandi possono essere uno o più di uno.

PROGETTO 1

Lo script seguente accetta come argomento le parole lista, per ottenere la lista dei file, e data, per ottenere la data odierna.

```
#!/bin/bash
# scelte
if [ "$1" = 'lista' ]
  then
    ls -l
fi
if [ "$1" = 'data' ]
  then
    date
fi
```

La seconda riga di commento dello script, che inizia con #, è usata per dare un titolo allo script. La variabile posizionale $1 è racchiusa tra virgolette perché, nel caso in cui lo script venisse eseguito senza parametri, si avrebbe un errore sintattico.

PROGETTO 2

Sono accettati come argomenti dello script due numeri di cui si vuole calcolare il quoziente controllando che il secondo numero sia diverso da zero.

```
#!/bin/bash
# divisione
if [ "$2" = 0 ]
  then
    echo 'divisione impossibile'
  else
    let quoz=$1/$2
    echo "quoziente = $quoz"
fi
```

Si noti che, nel secondo comando *echo*, *$quoz* (valore di *quoz*) è contenuto all'interno delle virgolette che delimitano il messaggio di output. Il primo *echo* utilizza invece gli apici. Come già detto gli apici singoli delimitano una sequenza di caratteri, i doppi apici consentono anche l'espansione di variabili e comandi.

Lo script può essere ulteriormente migliorato con un controllo sul numero dei parametri che l'utente ha introdotto con il comando; per far questo basta controllare il valore della variabile di sistema *$#* che contiene il numero dei parametri passati a un comando:

```
#!/bin/bash
# divisione
if [ $# -ne 2 ]
  then
    echo 'troppo pochi parametri'
elif [ "$2" = 0 ]
    then
      echo 'divisione impossibile'
    else
      let quoz=$1/$2
      echo "quoziente = $quoz"
fi
```

PROGETTO 3

Cambiare il nome di un file accettando come argomenti il vecchio nome e il nuovo e controllando che il primo esista e che il secondo non esista già.

Lo script serve a mostrare l'uso delle primitive di test sui file.

```
#!/bin/bash
# cambio nome
if [ $# -ne 2 ]
  then
       echo 'troppo pochi parametri'
elif [ -e $2 ]
  then
    echo "il secondo file esiste gia'"
elif [ -e $1 ]
    then
      mv $1 $2
    else
      echo 'il primo file non esiste'
fi
```

La prima istruzione *echo* usa le virgolette perché tra i caratteri da visualizzare c'è anche il carattere apice.

PROGETTO 4

Controllare se un numero è interno o esterno a un intervallo e se è diverso da 2.

Il progetto mostra l'uso degli operatori di confronto tra numeri e degli operatori booleani AND e OR. Quest'ultimi sono indicati rispettivamente con **–a** e **–o**.

```
#!/bin/bash
#operatori
if [ $# -ne 1 ]
```

```
  then
       echo 'numero parametri errato'
  else
if  [ $1 -ge 2 -a $1 -le 10 ]
  then
    echo "compreso nell'intervallo"
fi
if   [ $1 -lt 1 -o $1 -gt 100 ]
  then
    echo "esterno all'intervallo"
fi
if  [ $1 -ne 2 ]
  then
    echo 'diverso da 2'
fi
fi
```

Lo script controlla il valore numerico introdotto come parametro, verificando se è interno o esterno a due intervalli di valori (da 2 a 10 compresi gli estremi e da 1 a 100 esclusi gli estremi) e successivamente se è diverso da 2.

La struttura di **ripetizione** è realizzata con il comando **for**.
La sintassi generale della struttura è la seguente:

```
for variabile in lista_di_valori
do
   comandi
done
```

La struttura *for* assegna alla *variabile* in successione tutti i valori elencati nella *lista_ di_ valori* e per ogni assegnamento esegue i *comandi* scritti tra *do* e *done*.
La lista di valori, come accade molto frequentemente, può essere costituita da una *command substitution*.

PROGETTO 5

Calcolare e visualizzare l'occupazione totale in byte di tutti i file della directory corrente che hanno il nome che inizia con i caratteri arch (arch1, arch2, ecc.).

```
#!/bin/bash
# occupazione
somma=0
for spazio in `ls -l arch* | cut -c34-42`
do
   let somma+=spazio
done
echo "spazio totale = $somma"
```

Si ricordi di controllare la posizione corretta dei byte dei file all'interno della riga di output del comando *ls -l*, per determinare i parametri del comando *cut*.

La lista dei valori può anche essere costituita dai valori passati come parametri al comando.

PROGETTO 6

Calcolare il doppio dei numeri forniti come parametri del comando.

```
#!/bin/bash
# doppio
for numero in $*
do
   let numero*=2
   echo $numero
done
```

I numeri devono essere scritti accanto al comando separati da uno spazio.

ALTRE STRUTTURE DI CONTROLLO

Il costrutto di **selezione multipla** si scrive in *Bash shell* usando la struttura **case … esac** nel modo seguente (*esac* è la parola *case* scritta alla rovescia):

```
#!/bin/bash
case selettore in
valore1 | valore2 | valore3) comandi1 ;;
valore4) comandi4 ;;
valore5) comandi5 ;;
*) comandi6 ;;
esac
```

La variabile *selettore* che segue la parola chiave *case* è confrontata in sequenza con i valori scritti dopo *case*, che possono contenere i metacaratteri *, ? e [ ] con i significati già visti. Se la variabile *selettore* coincide con uno dei valori elencati, si esegue il comando (o i comandi) scritto dopo la parentesi tonda. La lista dei comandi da eseguire termina con un doppio punto e virgola.
Se nessuno dei valori elencati coincide con la stringa, sono eseguiti i comandi scritti dopo l'asterisco *, se questa opzione è presente. Al termine il controllo passa al comando successivo a *esac.*
Se ci sono più valori del selettore che devono produrre la stessa esecuzione di comandi, essi sono scritti sulla stessa riga separati dalla barra verticale **|.**

PROGETTO 7

Effettuare l'operazione aritmetica richiesta dall'utente su due numeri inseriti come argomenti dello script insieme al segno dell'operazione (nell'ordine devono essere inseriti il primo operando, il segno dell'operazione e il secondo operando, separati da uno spazio).

```
#!/bin/bash
# operazioni
case $2 in
'+')  let risult=$1+$3
```

```
      echo $risult ;;
'-')  let risult=$1-$3
      echo $risult ;;
'x')  let risult=$1*$3
      echo $risult ;;
':')  let risult=$1/$3
      echo $risult ;;
*) echo "La sintassi corretta e':
   <comando> <oper1> <segno> <oper2>"
   echo "I segni delle operazioni sono + - x :" ;;
esac
```

Nel caso in cui l'utente non inserisca alcun parametro o un numero di parametri errato, oppure utilizzi segni di operazione diversi da quelli previsti nello script, viene visualizzato un messaggio che descrive la sintassi corretta da utilizzare.
Per evitare confusione con i metacaratteri della shell, abbiamo scelto di indicare i segni delle operazioni di moltiplicazione e di divisione rispettivamente con x e :.
Lo script può essere migliorato inserendo il controllo sul secondo operando che deve essere diverso da zero nel caso della divisione, come già mostrato nel Progetto 2.

Le strutture di **ripetizione** con controllo della condizione nella *Bash shell* sono realizzate per mezzo dei costrutti **while** e **until** nelle forme seguenti.

- **while**

La sintassi generale della struttura ha questo formato:

```
while [ condizione ]
do
   comandi
done
```

La struttura consente di ripetere i comandi tra *do* e *done* mentre la *condizione* specificata dopo *while* mantiene il valore logico vero (valore diverso da zero). La condizione deve essere racchiusa tra parentesi quadre, nella stessa forma abbreviata del comando *test*, già vista per la struttura *if*.
Lo script seguente realizza una ripetizione senza fine che riproduce sul video la data corrente:

```
#!/bin/bash
# non finisce mai
a=0
while [ $a = 0 ]
do
  date
done
```

L'esecuzione deve essere interrotta con i tasti CTRL + c.

Si noti che nella condizione scritta dopo *while* è stata indicata la notazione *$a* per indicare il valore della variabile *a*.

PROGETTO 8

Sommare i primi 10 numeri naturali.

```
#!/bin/bash
# somma numeri
som=0
cont=1
while [ $cont -le  10 ]
do
   let som+=$cont
   let cont+=1
done
echo $som
```

• until

La sintassi generale della struttura è la seguente:

```
until [ condizione ]
do
   comandi
done
```

La struttura è del tutto analoga alla struttura *while*: la differenza consiste nel fatto che la ripetizione viene eseguita finché la condizione scritta vicino a *until* diventa vera, in altre parole i comandi compresi tra *do* e *done* sono ripetuti mentre la condizione si mantiene falsa. Questo significa che, data una struttura *while*, si può costruire la struttura *until* equivalente ad essa, scrivendo vicino a *until* la negazione logica della condizione scritta vicina a *while*.

PROGETTO 9

Sommare i primi 10 numeri naturali.

Il precedente script che calcola la somma dei primi 10 numeri naturali può essere riscritto in forma del tutto equivalente utilizzando la struttura *until:*

```
#!/bin/bash
# somma numeri
som=0
cont=1
until [ $cont -gt  10 ]
do
   let som+=$cont
   let cont+=1
done
echo $som
```

All'interno delle strutture *while* e *until* può essere usato il comando **break** che causa l'interruzione del ciclo e provoca il trasferimento del controllo al primo comando che segue l'istruzione *done* della ripetizione.

Esempi di shell script

PROGETTO 10

Creare con l'editor un file contenente per ogni riga due campi, codice articolo e quantità venduta, separati dal carattere di tabulazione: ci possono essere vendite diverse anche per lo stesso articolo. Costruire lo shell script che accetta come argomenti il codice articolo e il nome del file e restituisce la media della quantità venduta dell'articolo richiesto.

```
#!/bin/bash
# vendite
somma=0
cont=0
declare media
for qta in `grep $1 $2 | cut -f2`
do
  let somma+=qta
  let cont+=1
done
let media=somma/cont
echo "media articolo $1 = $media"
```

Per l'esecuzione occorre fornire, oltre al nome dello script, anche due argomenti: il codice articolo prefissato e il nome del file che contiene i dati; questi argomenti verranno associati rispettivamente a $1 e $2 (variabili posizionali).

Si può osservare che la procedura è composta dalle seguenti parti:

- assegna a *somma* e a *cont* il valore iniziale 0

```
somma=0
cont=0
```

- definisce la variabile *media* senza attribuirle un valore

```
declare media
```

- struttura di ripetizione sulle righe del file

```
for qta ......
do
   ......
   ......
done
```

I valori della variabile *qta* sono acquisiti come risultato della *command substitution* della pipeline:

```
`grep $1 $2 | cut -f2`
```

MATERIALI ON LINE

Il codice associato a $1 viene usato come valore di ricerca nelle righe del file associato a $2: sulle righe così selezionate viene fatta un'operazione di proiezione (*cut*) sul secondo campo (*f2*).

- somma mano a mano le quantità e conta il numero di record selezionati

```
let somma+=qta
let cont+=1
```

- calcola la media e visualizza il risultato

```
let media=somma/cont
echo "media articolo $1 = $media "
```

Lo script può essere migliorato aggiungendo:

- un controllo sul numero dei parametri inseriti dall'utente, come visto negli esempi di paragrafi precedenti;
- un controllo sul valore della variabile *cont* all'uscita dal ciclo *for*: nel caso in cui il suo valore sia uguale a zero non viene calcolata la media perché significa che non ci sono vendite per il codice articolo richiesto.

PROGETTO 11

Costruire uno shell script che accetta come argomento una delle seguenti lettere: D, W, L, P e restituisce rispettivamente: la data di oggi, l'elenco degli utenti collegati, la lista dei file e la directory corrente. Controllare anche che l'utente inserisca un argomento e in caso negativo suggerire le scelte possibili.

```
#!/bin/bash
# scelte di comandi
case $1 in
'D') date;;
'W') who;;
'L') ls -l|more;;
'P') pwd;;
*)
   echo 'D = data odierna'
   echo 'W = utenti collegati'
   echo 'L = lista dei file'
   echo 'P = directory corrente';;
esac
```

PROGETTO 12

Dopo aver creato con l'editor un file di nome dizion contenente per ogni riga un termine in italiano e la sua traduzione in inglese, separati dal carattere di tabulazione, si costruisca uno shell script che accetta come argomento del comando la parola in italiano e restituisce la sua traduzione in inglese oppure un messaggio di termine non trovato. Controllare anche che l'utente inserisca la parola da tradurre e in caso negativo suggerire la sintassi corretta.

```
#!/bin/bash
# traduttore
if [ $# -ne 1 ]
then
  echo "La sintassi corretta: <comando> <termine>"
else
  traduzione=`grep $1 dizion | cut -f2`
  if [ -z $traduzione ]
  then
     echo "Il termine non e' presente nel dizionario"
  else
     echo "La traduzione di $1 = $traduzione"
  fi
fi
```

PROGETTO 13

Dato un file di nome articoli, contenente per ogni riga 4 colonne con codice reparto, descrizione, prezzo, quantità, si vuole calcolare, per un reparto il cui codice viene fornito dall'utente insieme al comando:
- la giacenza di magazzino (somma delle quantità) dei prodotti di quel reparto
- il prezzo medio dei 10 prezzi più alti tra i prodotti di quel reparto.
Controllare anche che l'utente usi la sintassi del comando in modo corretto.

Per controllare che l'utente inserisca il codice, occorre inserire all'inizio dello script, una struttura del tipo

```
if [ $# -ne 1 ]
  then ....
```

Il codice del reparto fornito dall'utente viene conservato nella variabile posizionale $1 e quindi per estrarre da ciascuna riga il valore della quantità si deve usare una ripetizione con:

```
for qta in `grep "^$1" articoli | cut -f4`
```

Il carattere ^ indica che la ricerca del codice di reparto deve essere effettuata a partire dal primo carattere di ogni riga. L'opzione *-f4* del comando *cut* consente di estrarre il quarto campo di ogni riga che contiene la quantità.
Al secondo punto del problema, per calcolare la media occorre prima effettuare la somma dei prezzi dei prodotti con i 10 prezzi più alti di quel reparto; la struttura da utilizzare è la seguente:

```
for prezzo in `grep "^$1" articoli | sort -nr +3| head -10 | cut -f3`
```

MATERIALI ON LINE

È opportuno inserire anche un controllo sul valore di *cont* all'uscita dal secondo ciclo *for*, per effettuare il calcolo della media solo se il valore di *cont* è maggiore di 0.
Lo script completo è il seguente:

```
#!/bin/bash
# articoli e reparti
if [ $# -ne 1 ]
then
   echo 'La sintassi corretta: <comando> <reparto>'
else
#giacenza del reparto
somma=0
for qta in `grep "^$1" articoli | cut -f4`
do
  let somma+=qta
done
echo "giacenza dei reparto $1 = $somma"
#media dei 10 prezzi piu' alti
somma=0
cont=0
declare media
for prezzo in `grep "^$1" articoli|sort -nr +3|head -10|cut -f3`
do
  let somma+=prezzo
  let cont+=1
done
if [$cont -gt 0]
then
  let media=somma/cont
  echo "media dei 10 prezzi piu' alti del reparto $1 = $media "
fi
fi
```

PROGETTO 14

Dato il file carichi contenente per ogni riga i seguenti campi: giorno, mese (3 lettere: gen, feb, ...), anno, codice, pezzi, costruire uno script per calcolare e visualizzare il totale dei pezzi caricati in magazzino in un mese per un codice articolo; il mese e il codice vengono forniti come argomenti dello script al momento dell'esecuzione.

L'utente deve fornire due argomenti quando richiede l'esecuzione dello script; è quindi opportuno inserire all'inizio dello script un controllo con eventuale messaggio per l'utente nel caso in cui il numero dei parametri forniti non sia corretto:

```
if [ $# -ne 2 ]
 then ......
```

La selezione deve poi essere fatta su ciascuna riga per mese e per codice articolo con una struttura di questo tipo:

```
for pezzi in `grep $1 carichi | grep $2 | cut -f5`
```

La pipeline opera correttamente la selezione delle righe nell'ipotesi che i mesi delle date siano indicate in lettere e che il codice dell'articolo sia alfanumerico.

Lo script completo è il seguente:

```
#!/bin/bash
#carichi di magazzino
if [ $# -ne 2 ]
then
 echo 'La sintassi corretta: <comando> <mese> <codice>'
else
 somma=0
 for pezzi in `grep $1 carichi | grep $2 | cut -f5`
 do
  let somma+=pezzi
 done
 echo "totale pezzi nel mese $1 per il codice $2 = $somma"
fi
```

PROGETTO 15

Dato un file riviste contenente per ogni riga due colonne, titolo e copie vendute, costruire un comando per fornire in output il numero copie di un titolo fornito dall'utente come argomento, oppure un messaggio di non trovato.

Lo script deve contenere il controllo sul parametro fornito dall'utente, come negli esercizi precedenti. Si calcoli, poi, con una *command substitution*, il numero delle copie per il titolo richiesto:

```
copie=`grep $1 riviste | cut -f2`
if [ -z $copie ]
then
   echo ....
else
   echo ....
fi
```

Il messaggio visualizzato cambia a seconda che il valore di copie ottenuto dalla pipeline sia uguale o diverso dalla stringa vuota:

```
[ -z $copie ]
```

Lo script che risolve il problema è il seguente:

```
#!/bin/bash
# edicola
if [ $# -ne 1 ]
then
   echo 'La sintassi corretta: <comando> <titolo>'
else
   copie=`grep $1 riviste | cut -f2`
   if [ -z $copie ]
    then
      echo 'Rivista non trovata'
    else
      echo "Numero di copie vendute per $1 = $copie"
  fi
fi
```

LE FUNZIONI

La **funzione** è una lista di comandi che costituisce un sottoprogramma dello shell script e che svolge un determinato compito. La funzione offre la possibilità di organizzare un insieme di comandi che può essere eseguito più volte.
La funzione è identificata da un nome, con il quale può essere richiamata dallo script, così come si fa con i comandi interni di Linux.
La struttura generale di una **function** è:

```
function nome_funzione  () {
     comandi
}
```

I comandi che formano il corpo della funzione sono raggruppati all'interno di una coppia di parentesi graffe.
La specificazione della parola *function* è facoltativa.
Ci possono essere variabili che sono utilizzate solo all'interno della funzione, cioè variabili locali: esse devono essere dichiarate all'interno della *function* e precedute dalla parola chiave **local**.

```
local variabile=valore
```

È possibile sfruttare le caratteristiche delle funzioni per eseguire uno stesso gruppo di comandi più volte, ma con valori diversi per un parametro. In questo caso occorre introdurre all'interno della *function* i **parametri posizionali** (le variabili che iniziano con il carattere $), proprio come si è fatto in precedenza con gli script. In questo caso la chiamata della *function* deve contenere anche i valori da passare ai parametri posizionali.
Il comando **return** consente la terminazione anticipata della *function*.
È possibile dichiarare nuove funzioni rendendole disponibili alla shell attraverso il comando **export**, come accade per le variabili. Se le funzioni devono diventare permanenti, è possibile anche inserire la loro dichiarazione all'interno dei file di inizializzazione che vengono letti al momento della connessione.
L'utente può controllare le funzioni attualmente disponibili attraverso il comando **set**.

PROGETTO 16

Costruire uno script che calcola lo spazio occupato da due categorie di file: i file di testo aventi estensione .txt nel nome e i file immagine aventi estensione .png.

Lo script riutilizza in parte il codice usato in precedenza, come esempio per illustrare la struttura *for*, per calcolare lo spazio occupato da un insieme di file. La *function* contiene una variabile locale e un parametro posizionale.

```
#!/bin/bash
#funzione per calcolare lo spazio occupato
function occupa () {
local somma=0
for spazio in `ls -l *.$1 | cut -c29-32`
do
   let somma+=spazio
```

```
done
echo "spazio totale = $somma"
}
#main
echo 'occupazione file testi'
occupa txt
echo 'occupazione file immagini'
occupa png
```

La posizione dei caratteri da estrarre con il comando *cut* può variare a seconda della directory corrente.

La shell consente la **chiamata ricorsiva** delle *function*, cioè è possibile chiamare l'esecuzione di una *function* dall'interno della funzione stessa.

PROGETTO 17

Scrivere lo script per calcolare il fattoriale di 3 usando una funzione ricorsiva.

```
#!/bin/bash
#fattoriale
function fact() {
    if [ $1 = 0 ]
     then
        echo 1
     else
        echo $(( $1 * $( fact $(( $1 - 1 )) ) ))
    fi
}
if [ $# -ne 1 ]
  then
     echo 'manca il numero'
  else
     fact $1
fi
```

In questo progetto, per una maggiore comprensione, è stata usata la notazione con il carattere $ seguito dalle parentesi tonde per rappresentare l'espansione dei calcoli aritmetici e la *command substitution*, come mostrato alla fine del paragrafo 5.

Si osservi con attenzione che lo script usa due volte la variabile posizionale $1:
- nell'ultima riga dello script la variabile $1 indica il numero fornito dall'utente insieme al comando;
- all'interno della *function* la variabile $1 rappresenta il parametro passato con la chiamata della funzione.

ESERCIZI

1 Costruire un comando che consente all'utente di aggiungere in coda a un file storico delle utenze il proprio nome di utente, in quale directory sta lavorando, la data odierna e una riga di separazione formata da trattini "– ".

2 Scrivere lo script che consente di calcolare il numero dei file e lo spazio totale occupato in byte per i file che iniziano con il carattere *a* presenti nella directory corrente.

3 Costruire uno script per copiare tutti i file creati oggi nella home directory in un'altra directory di nome *odierna*, dopo averla creata all'interno della home directory; man mano si visualizzi il loro nome e i file vengano cancellati dalla home directory.

4 Dato un file *persone* contenente per ogni riga un nome e una città separati dal punto e virgola, calcolare il numero delle persone che abitano in una città fornita come argomento dello script; controllare anche che venga inserita la città come parametro insieme al comando.

5 Dato un file *libri* a tre colonne con titolo, editore e prezzo, calcolare il prezzo medio dei libri di un editore fornito come argomento dello script. Comunicare in output la media dei prezzi e il numero di libri selezionati per quell'editore.

6 Creare con l'editor un file *verifiche* contenente per ogni riga i seguenti campi: giorno, mese, anno, nome studente, voto. Costruire poi uno script per contare il numero di prove effettuate in un mese per uno studente; il mese e il nome dello studente vengono forniti come argomenti dello script al momento dell'esecuzione. Si rappresentino i mesi con una stringa di tre caratteri (gen, feb, mar, ecc.).

7 Dato un file *elenco* contenente per ogni riga due colonne, nazione e milioni di abitanti, costruire un comando per fornire in output il numero di abitanti di una nazione fornita dall'utente come argomento, oppure un messaggio di non trovato.

8 Dato un file *studenti* contenente per ogni riga un nome e un comune separati dal punto e virgola, calcolare il numero degli studenti che abitano in un comune fornito come argomento dello script; suggerire all'utente anche la sintassi corretta nel caso in cui non venga inserito il comune insieme al comando.

9 Simulazione di un motore di ricerca: cerca in una directory i file che contengono una parola cercata e fornita, insieme al comando, dall'utente. Prima fornisce il numero dei file trovati, poi di ciascuno visualizza il nome e le prime cinque righe del file. Controlla anche che l'utente fornisca una parola come argomento del comando.

10 Abbiamo ricevuto da un utente un file contenente per ogni riga un'espressione da calcolare: costruire il comando *calcposta* che, ricevendo come argomenti il nome del file e il nome dell'utente, invia per posta elettronica i risultati di un calcolo all'utente che li ha richiesti. Controllare che insieme al comando vengano forniti due argomenti.

11 Costruire uno script che consente all'utente di fornire una lettera e un numero: le lettere possono essere una delle quattro seguenti, e si deve fornire come output il risultato del calcolo richiesto: S = successivo, P = precedente, D = doppio, M = metà, del numero fornito dall'utente.

12 Costruire uno script che calcola l'IVA su un importo passato come parametro a una funzione.