Sistemi Operativi, Secondo Modulo, Canale A–L e Teledidattica
Riassunto della lezione del 13/03/2019

Igor Melatti

Comandi vari

- Comandi `less` {`files`} e `more [-num] [+num] [-d]` {`files`}
  - come `cat`, ma paginano l'output se è troppo lungo
  - nonostante i nomi, `more` è più limitato di `less`
  - `less` è normalmente usato da `man` per mostrare una pagina di manuale
  - differiscono in svariati comportamenti:
    * `less` permette di muoversi sempre sia in avanti che all'indietro, `more` solo se usato senza pipelining (ovvero, solo nel modo detto sopra)
    * tutti e due si chiudono premendo `q`, ma una volta chiuso `less` sparisce tutto quello che era scritto prima, con `more` resta l'ultima schermata
    * `less` pagina sempre, `more` solo se l'output è più grande di una pagina
    * far riferimento al `man` per una descrizione dei comandi interattivi ammessi da `more` e `less`
    * opzioni di `more`: `-num` setta a `num` il numero di righe in una pagina; `+num` comincia la visualizzazione dalla riga `num`, `-d` mostra una sorta di lista di comandi in basso
    * tra i comandi interattivi c'è la ricerca di espressioni regolari: basta digitare `/`, seguita dall'espressione regolare (estesa); in `more` non trova le occorrenze nella pagina attuale, ma solo in quelle seguenti; in `less` trova tutte le occorrenze, evidenziandole
    * tra i comandi interattivi c'è la `h`, che mostra l'help dei comandi interattivi
    * **esercizio:** trovare il modo per andare avanti ed indietro di k righe e di k pagine con `more` e `less`

Sezione	Contenuto
1	User Command
2	System Calls
3	Library Functions
4	Special Files
5	File Formats
6	Games
7	Miscellaneous
8	System Administration and Privileged Commands
9	Kernel Interfaces (dipende dalla distribuzione)

Table 1: Sezioni del manuale

- Comando `man`, già visto in lezione 2
  - la lista completa delle sezioni è riportata in Tabella 1
  - **esercizio:** fare in modo che il contenuto di una pagina di manuale sia scritto tutto insieme, senza che occorra premere `q` per ritornare al prompt
- Comando `clear`
  - nessun argomento, nessuna opzione
  - pulisce lo schermo: ritorna il prompt in alto a sinistra
- Comando `echo [-n] [-e] stringa`
  - già visto, vengono ora precisati gli argomenti
  - stringa può contenere anche degli spazi
  - `-n`: non va a capo dopo aver stampato `stringa`
  - l'opzione `-e` abilita le sequenze di escape con il \: `\t` è la tabulazione, `\n` è l'andata a capo, `\xHH`, con `HH` cifre esadecimali, è il carattere con codice ASCII `HH` (vedere Figura 1: la prima colonna dà la prima cifra, la prima riga la seconda)
  - **esercizio:** tenendo presente la Figura 1, scrivere il carattere di tabulazione, seguito da una `c`, e andare poi a capo
- Comandi `cat [-T] [-n] [-E] [file...]` e `tac [file...]`
  - già visto, vengono ora precisati gli argomenti
  - `-T` anziché stampare le tabulazioni nel modo standard (ovvero aggiungendo spazi fino ad arrivare al prossimo multiplo di 8 nella riga corrente), stampa `^I`
  - `-n`: numera tutte le linee di output (e le precede con degli spazi)
  - `-E`: aggiunge $ alla fine di ogni riga

ASCII Code Chart

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2		!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

Figure 1: Il codice ASCII

  - spiritosamente, `tac` stampa dall'ultima riga alla prima
  - usa come encoding quello mostrato in `echo $LANG`, che di solito è UTF-8 (che contiene il codice ASCII di Figura 1)
  - Quindi, ogni sequenza di byte letta dal file viene stampata seguendo lo standard dato

- Comando `od [-N bytes] [-j bytes] [-A base] [-t type] [-c] [file...]`
  - fa il dump di un file, ovvero ne mostra il contenuto in esadecimale
  - quindi, è possibile passargli anche file non di testo, come ad esempio gli eseguibili
  - infatti, qui non c'è nessuna interpretazione (come fa `cat`) dei bytes letti: viene stampato il loro valore come numero (decimale, ottale o esadecimale)
  - senza argomenti legge da tastiera, ma occorre specificare l'output per intero (premendo CTRL+d alla fine)
  - `-N bytes`: fai il dump di solo `bytes` bytes
  - `-j bytes`: fai il dump saltando i primi `bytes` bytes
  - `-t type`: scegli la base per il dump vero e proprio: `x` per esadecimale, `d` per decimale, `o` per ottale (default); si può anche specificare quanti bytes per ogni intero (solo potenze di 2, e fino ad un certo punto dipendente dall'implementazione); se si aggiunge una `z` al tipo stampa alla fine di ogni riga i caratteri stampabili
  - `-A base`: scegli la base per gli offset (prima colonna dell'output), stesse scelte del `-t` (non proprio vero, ma ok per i nostri scopi; vedere il `man`)
  - `-c`: mostra anche la corrispondenza con i caratteri stampabili

  - **esercizio:** farsi visualizzare i bytes che vanno dal decimo al trentesimo del contenuto dell'eseguibile del comando `more` (vedere anche il comando `which` più avanti), prima in ottale e poi in esadecimale, con la prima colonna sempre in esadecimale; se per caso qualche byte è un carattere stampabile, farselo stampare
  - **esercizio:** fare il dump di alcuni caratteri immessi da tastiera, e verificare con `echo` che la conversione sia esatta (sia in ottale che in esadecimale)

- Comando `head [-c car] [-n righe] [file...]`
  - al solito, senza argomenti legge da tastiera (una riga per volta)
  - come `cat`, ma stampa solo i primi `car` caratteri o le prime `righe` righe (ciò che è minore) dei file dati
  - senza opzioni stampa 10 righe, senza limiti sui caratteri
  - **esercizio:** scrivere quanto segue su un file, e poi farsi stampare solo la prima riga, ma usando l'opzione `-c`:

    ```
    ciao
    addio
    ```

- Comando `tail [-n righe] [-f] [file...]`
  - al solito, senza argomenti legge da tastiera (ma questa volta occorre premere CTRL+d alla fine dell'input)
  - come `cat`, ma stampa solo le ultime `righe` righe dei file dati
  - con `-f`, aggiorna di continuo la stampa: utile se si vuole leggere un file cui vengono continuamente aggiunti dati (ad esempio, come risultato di una qualche computazione, *mentre* la computazione stessa è in esecuzione)
  - **esercizio:** creare un file con `gedit` (lanciato in background), scriverci dentro almeno 4-5 righe e poi salvare; tornare sulla shell ed eseguire `tail nomefile`. Poi aggiungere qualche altra riga, ed eseguire nuovamente `tail nomefile`. Effettuare nuovamente questi passaggi, ma eseguendo stavolta `tail -f nomefile`. È sufficiente scrivere le modifiche sul file, affinché vengano visualizzate dal `tail`? Perché?

- Comandi `cmp [-b] file1 file2` e `diff [-i] [-b] [-r] [-q] file1 file2`
  - confrontano i 2 file dati, ma esiste anche `diff3`
  - `cmp` si limita a trovare la prima occorrenza di un byte in cui differiscano (con `-b`, stampa anche tale byte, sia in esadecimale che nella versione stampabile, if any)

  - `diff`, invece, lista *tutte* le differenze; le differenze vengono mostrate riga per riga (2 righe che differiscono per un solo carattere verranno mostrate come differenti *tout-court*)
  - `-b`: ignora le differenze, se consistono solo in un numero diverso di spazi
  - `-i`: ignora le differenze, se consistono solo nel *case* diverso (minuscolo vs. maiuscolo)
  - `-r`: confronta ricorsivamente 2 directory, confrontando con `diff` i file che hanno lo stesso percorso relativo all'interno di tali directory, e segnalando quali file sono presenti in una sola delle 2 directory
  - `-q`: dice solo se sono uguali o se sono diversi, senza listare le differenze
  - **esercizio:** creare un file e scriverci dentro qualche riga; poi copiarlo in un altro file, farci delle modifiche e poi confrontare i 2 file sia con `diff` che con `cmp`. Rifare poi la stessa cosa, ma al posto della copia creare un link. Rifare la stessa cosa, ma con directory al posto dei file; modificare alcuni file e poi eseguire `diff` sia con che senza `-r`

- Comando `patch [--dry-run] [-f] file diff_file`
  - si supponga di aver confrontato `file1` e `file2` con `diff`, e di aver salvato il risultato in `diff_file`
  - allora, il comando `patch file1 diff_file` trasforma `file1` in `file2`
  - con `--dry-run`, si limita a scrivere `file2` a schermo, senza modificare `file1`
  - con `-f`, non chiede nessuna conferma, come potrebbe accadere per alcune scelte
  - **esercizio:** creare un file e scriverci dentro qualche riga; poi copiarlo in un altro file, farci delle modifiche e poi confrontare i 2 file `diff`. Copiare il risultato dentro un nuovo file, e applicare la patch. Controllare con `diff` il risultato

- Comando `date [-s data] [-d data] [+formato]`
  - senza argomenti, scrive la data e l'ora (con fuso orario usato)
  - con `-s data`, cambia la data e la setta a `data` (solo con `sudo`)
  - con `-d data`, mostra la `data` (utile per fare conversioni)
  - con `+formato`, si cambia il modo con cui la data viene mostrata; `formato` può essere composto dalle seguenti sottoparti:
    - `%F`: scrive solo la data come YYYY-MM-DD; equivalente a `%Y-%m-%d`
    - `%H`: scrive solo l'ora (come in un orologio digitale)
    - `%M`: scrive solo i minuti (come in un orologio digitale)

    - `%S`: scrive solo i secondi (come in un orologio digitale, ma c'è anche il valore 60! Per farsi una cultura, cercare "leap second" o "secondo intercalare")
    - `%N`: scrive solo i nanosecondi
    - `%s`: numero di secondi intercorso dalla mezzanotte in punto del primo gennaio 1970 (più o meno l'accensione del primo Unix, detto anche *epoch*)
    - per gli altri, vedere `man date`
  - il formato di `data`, invece, è un po' più libero, vedere il `man` (ma anche `info date` per altre informazioni)
  - `-d` può servire a convertire da epoch a data umana: ad esempio il comando `date -d @1000 +%F-%H-%M-%S` mostra la data dopo 1000 secondi da epoch
  - **esercizio:** farsi stampare solo l'anno in cui ci troveremo, dopo 10 miliardi di secondi dal 1970

- Comando `find` {`dir`} {`espressione`}
  - altro comando con opzioni non semplici
  - serve per trovare file (che vengono cercati solo nelle directory `dir`), ed eventualmente effettuare delle azioni su di essi
  - le espressioni possono essere:

    opzioni le più importanti sono `-maxdepth M`, che impedisce di scendere per più di `M` sottodirectory in una delle directory di `dir`, e `-regextype T`, che permette di scegliere quali espressioni regolari usare: `T` può essere una tra `posix-awk`, `posix-egrep` e `posix-extended` (sono essenzialmente le ERE di lezione 5, con minime differenze; vedere `https://www.gnu.org/software/findutils/manual/html_node/find_html/posix_002dawk-regular-expression-syntax.html#posix_002dawk-regular-expression-syntax`) e `posix-basic` (le BRE di lezione 5)

    test se un test ritorna vero quando applicato ad un file, su tale file verranno applicate le azioni specificate; i test più importanti sono:
      - `-name pattern`: vero se `pattern` è soddisfatto; attenzione, `pattern` segue le regole del wildcarding, non delle BRE o delle ERE
      - `-iname pattern`: come `-name`, ma ignora la differenza maiuscole/minuscole
      - `-type [bcdolfls]`, dove `[bcdolfls]` è da intendersi come una BRE: ritorna vero se il file è del tipo specificato (ad esempio, `f` è per i file regolari, `l` per i link simbolici, `d` per le directory; vedere il `man`)

* `-size c[cwbkMG]`, dove `[cwbkMG]` è da intendersi come una BRE: ritorna vero se la dimensione del file è esattamente quella indicata (`c` sta per bytes, `k` per kB; vedere il `man`)
* `-user uname`: vero se il file appartiene all'utente `uname`
* `-perm mode`: vero se il file ha i permessi indicati in `mode`
* `-regex pattern`: come `-name`, ma questa volta `pattern` è interpretato come una BRE o come una ERE, dipendentemente da `-regextype`; inoltre, fa match con l'intero path ritornato (mentre `-name` fa match solo con il nome del file)
* `-atime T`: vero se il file è stato acceduto 24`T` ore fa; ci sono analogamente anche `-mtime` e `-ctime`
* `empty`: vero se il file (o la directory) è vuoto
* `-cnewer file`: vero se il file è più recente (come data di modifica) di `file`; ci sono analogamente anche `-anewer` e `-newer`
* in generale, si possono combinare tra di loro più condizioni di test, e verrà applicato l'AND logico tra di esse
* si possono anche usare NOT (`!`), AND (`-a`) ed OR (`-o`) tra le varie condizioni di test, con `\(\)` per raggruppare sottoespressioni

azioni da applicare ai file che superano i test; per essere più precisi, tali azioni vengono applicate, una alla volta, a ciascun file/directory che supera i test. Le azioni più importanti sono:

* `-ls`: applica `ls -dils` al file
* `-delete`: cancella il file
* `-exec command ;`: esegue il comando `command`, all'interno della quale si può usare {} al posto del nome del file
  - *sembra* facile ma... cominciano ad entrare in gioco le stranezze delle shell
  - inanzitutto, il `;`, dato così, verrebbe interpretato dalla shell stessa come separatore di comandi *prima* di essere passato come argomento a `find` (provare ad eseguire `echo ciao; echo ciao`)
  - quindi, occorre sostituirlo con uno dei seguenti: `\;`, `";"`, `';'` (provare ad eseguire lo stesso comando di cui sopra, sostituendo `;` in uno di questi 3 modi)
  - secondo, occorre che il `;` sia un argomento a parte di `-exec` (vedere com'è scritto nel `man`...); ovvero, `-exec` vuole 2 argomenti: il comando e il `;`
  - pertanto è necessario mettere (almeno) uno spazio prima del `;` "riscritto" (notare invece che scrivere `echo ciao;echo ciao` e `echo ciao ; echo ciao` è la stessa cosa)

∗ `-print`: stampa il nome del file (default action)

- ... e occhio a non confondere globbing con i pattern (soprattutto quelli con `-name`...); conviene sempre mettere i pattern tra `''`, così da evitare che la shell li interpreti prima di passarli a `find` (vedere esempio finale di lezione 5)
- **esercizio:** con una sola riga di comando, cancellare tutti i file, nel raggio di 3 sottodirectory da quella attuale, che abbiano un nome palindromo lungo 7 caratteri alfanumerici (è possibile usare `-name`?)
- **esercizio:** come prima, ma eseguire `ls` anziché cancellare, e limitarsi alla sola directory corrente. È proprio necessario usare `find`, o si può usare `ls`?
- **esercizio:** con una sola riga di comando, eseguire `ls` su tutti i file, nella directory attuale, che inizino con 2 caratteri alfanumerici, e poi continuino con stringhe non vuote qualsiasi. È proprio necessario usare `find`, o si può usare `ls`?
- **esercizio:** con una sola riga di comando, far sì che venga stampata una linea per ogni file che sia un link simbolico; tale linea dev'essere "Il file nomedelfile e' un link simbolico"

- Comando `id [-u] [-g] [-G] [username]`: stampa user id, group id, e i tutti i gruppi cui l'utente `username` (o quello attuale, se `username` non è specificato) appartiene
  - `-u`: stampa solo lo user id
  - `-g`: stampa solo il group id
  - `-G`: stampa solo gli id dei gruppi cui `username` appartiene
- Comando `who [-a]` e `uptime`: informazioni sul sistema, ma disponibili a tutti
  - `who`: chi è attualmente loggato (con `-a` li mostra proprio tutti, comprese le schermate di testo attivabili con CTRL+ALT+Fn)
  - `uptime`: da quanto tempo il sistema è stato avviato
- Comando `whoami`: non sempre c'è il prompt come ce lo si aspetta, non sempre ci si ricorda quale utente si sta impersonando (magari dopo un logout...), quindi questo comando stampa l'utente attualmente loggato
- Comando `which [-a] comando`: dove si trova l'eseguibile relativo al comando
  - variabile d'ambiente `PATH`: contiene le directory dove cercare quando si dà un comando (scrivere `echo $PATH`)
  - un comando viene eseguito prendendo il primo eseguibile effettivamente trovato in una di quelle directory

  - potrebbe succedere che ce ne sia più d'uno, ma per l'appunto conta il primo match
  - `which` mostra il path assoluto del file eseguibile relativo al comando
  - provare `which ls`, `which -a ls` e `which -a which`
  - provare anche `which cd`: i comandi built-in non hanno un file eseguibile, ci pensa direttamente bash

- Comando `time [-o file] [-a] [-f formato] [-v] [-p] comando`: raro caso di comando sia built-in che non
  - cioè esiste la versione built-in, che è quella che viene eseguita da Bash
  - per eseguire l'altra (che è più utile e ha tutte le opzioni riportate qui di seguito) occorre dare l'intero path (eseguire prima `which time`)
  - esegue `comando` (che può avere degli argomenti) e poi stampa statistiche sull'uso di CPU
  - con l'opzione `-o file` scrive il suo risultato (non quello di `comando`!) su `file`, sovrascrivendolo; se c'è anche l'opzione `-a`, appendendo alla fine
  - con l'opzione `-f formato` si può scegliere come dev'essere l'output di `time`; vedere `FORMATTING THE OUTPUT` nel `man time`
  - informazioni standard (senza opzioni):
    * tempo *reale*, detto anche *elapsed time* o *wallclock time*: si fa partire l'orologio quando viene dato il comando, e lo si ferma quando termina; sarebbe quello che nel modulo 1 veniva chiamato *turnaround time*
    * tempo *utente*, detto anche *user time*: si tolgono i tempi di attesa (senza distinguere tra attesa quando pronto o quando bloccato), quindi resta solo il tempo usato effettivamente in esecuzione su un processore; di questo, tuttavia, si considera solo il tempo eseguito in modalità utente (ovvero, di esecuzione di codice non di sistema, quindi escludendo le chiamate a sistema e le gestioni delle eccezioni)
    * tempo *di sistema*, detto anche *system time* o *kernel time*: si tolgono i tempi di attesa (senza distinguere tra attesa quando pronto o quando bloccato), quindi resta solo il tempo usato effettivamente in esecuzione su un processore; di questo, tuttavia, si considera solo il tempo eseguito in modalità kernel (ovvero, dovuto ad una chiamata di sistema, o anche ad un'eccezione)
  - `-p`: usa il formato predefinito da POSIX
  - `-v`: dà tutte le informazioni, compreso l'uso della memoria
  - **esercizio:** confrontare le informazioni che `ps`, nel suo formato lungo con in più le informazioni su `vsz`, dà su se stesso con quello riportato da `time` nella sua versione completa

Il comando `awk`

- Comando `awk [-F separatore] [--posix] [-f file_awk] [-v var=var...] [programma_awk] [file...]`
  - nei moderni Linux, `awk` è un link a `gawk` (GNU awk)
  - la presente descrizione riguarda `gawk`; `awk` è la versione "vecchia", presente su alcuni vecchi Linux, e non supporta tutte le funzionalità descritte nel seguito
  - `gawk` è il programma principe per elaborare contenuti di testo
  - si basa su un vero e proprio programma, che può essere dato direttamente come argomento (`programma_awk`) o messo dentro un file (e allora si usa l'opzione `-f`)
  - questo programma è scritto in un linguaggio che è praticamente come il C, dove però si semplifica l'accesso ai file (e alle loro righe) e non è necessario compilare
  - quindi, si può fare praticamente *tutto*
  - l'input di `awk` è dato dai files dati come argomento; se non ci sono argomenti, allora legge ciò che viene scritto da tastiera
  - su tale input, `awk` lavora riga per riga
    - se l'input è da tastiera, allora dopo ogni pressione dell'invio `awk` valuta la riga e stampa evantualmente il suo output; quindi input ed output si vedranno mischiati
    - niente di nuovo: succedeva, ad esempio, anche per `grep`
    - se l'input è da file, allora l'output non sarà misto all'input
  - quindi, il programma awk specifica cosa occorre fare in una generica riga
  - più in dettaglio, un programma awk è una lista di righe di questo tipo:

    [`condizione`11 [, `condizione`12]] {`programma`1}

    ⋮

    [`condizione`n1 [, `condizione`n2]] {`programma`n}
  - per ogni riga, vengono valutate le condizioni e, se il risultato è vero, viene eseguito il corrispondente programma
    - quindi, per ogni riga possono essere eseguiti 0, 1 o più programmi
    - se ci sono 2 condizioni sulla stessa riga, separate da virgola, allora il programma viene applicato a tutte le righe che si trovano tra la prima riga che soddisfa la prima condizione e l'ultima riga che soddisfa l'ultima condizione
    - il programma si può anche articolare su più righe

    - non mettere la condizione equivale a dire che il rispettivo programma va eseguito per tutte le righe
- prima di essere passata a condizioni e programmi, ogni riga viene spezzata in svariati campi (o meglio, ridotta in *token*), a seconda del *field separator* (`FS`)
    - di default, `FS` è un qualunque spazio; con l'opzione `-F` lo si può ridefinire ad un qualunque carattere (in generale, ad un'espressione regolare)
- all'interno di condizioni e programmi si possono usare alcune variabili speciali (ce ne sono anche altre, vedere il `man`):

  `FNR` : numero di riga del file attuale

  `NR` : numero di riga tra tutti i file

  `ARGIND` : indice del file attuale (il primo ha indice 1)

  `NF` : numero di campi

  `FS` : separatore di campi

  `$`n : se n è compreso tra 1 ed `NF`, il valore dell'n-esimo campo

  `$0` : l'intera riga non spezzata
- si possono inoltre usare tutte le variabili eventualmente specificate con l'opzione `-v`
- per quanto riguarda le *condizioni*, possono essere definite come segue:
    - `var` $\sim$ `/extregex/`: valida solo se il contenuto della variabile `var` ha una sottostringa che soddisfa la ERE `extregex`
        - scrivendo solo `/extregex/`, si intende che `var` sia `$0`
        - rispetto alle ERE, per fare pattern matching con il letterale `/` occorre scrivere `\/` (a meno che non sia all'interno di un range)
        - senza l'opzione `--posix` o `-re-interval` sono delle ERE "azzoppate", senza gli intervalli {}; in ogni caso, non ci sono le backreference, e ci sono anche altre piccole differenze (vedere i link dati in lezione 5)
    - `var_o_const cmp var_o_const`: dove `cmp` è un operatore di confronto (`==`, $>$, etc)
    - le precedenti condizioni sono atomiche; possono essere combinate con AND (`&&`), OR (`||`) e NOT (`!`), e raggruppate con le normali parentesi
    - ci sono 2 condizioni speciali: `BEGIN` (vale solo prima della prima riga del primo file) e `END` (vale solo dopo l'ultima riga dell'ultimo file)
- per quanto riguarda i *programmi*, possono essere definiti come segue:
    - vale la sintassi del C (quindi anche del Java 1.6, limitatamente ai corpi dei metodi delle classi)

- assegnamenti con `=`, test di uguaglianza con `==`, `for (init; cond; iter) istruzioni`, `while (cond) istruzioni`, `do istruzioni while (cond)`, `break`, `continue`
- se `istruzioni` è un blocco che contiene più istruzioni, va racchiuso dalle parentesi graffe

* principali differenze con C/Java:
  - uso estremamente libero delle stringhe (lo ritroveremo negli script): la concatenazione tra variabili e/o costanti avviene senza operatori (tra costanti, lo fanno anche C e Java, ma con variabili no...)
  - confronto tra stringhe tramite `==`
  - confronto tra stringhe e regex con $\sim$ (le regex si riconoscono perché sono tra `//` anziché tra `""`)
  - il comando `exit` n fa terminare l'intero programma `awk` (anche se ci sono altre righe e/o altri file da analizzare); viene però eseguito il blocco `END`, se presente
  - il `;` è un separatore e non un terminatore, quindi può essere omesso dopo l'ultima istruzione
  - ad essere ancora più precisi, il `;` come separatore serve solo se si sta scrivendo un programma (o comunque, più di una istruzione) awk su una sola riga; altrimenti, si può omettere il `;` ed andare semplicemente a capo
  - non serve dichiarare le variabili, siano esse semplici o array: ci si limita ad usarle
  - niente errore se una variabile viene usata prima che sia assegnata: varrà la stringa vuota `""` (o anche zero, se viene usata come numero)
  - nel caso degli array, c'è l'istruzione speciale `delete(array)` che cancella tutti i dati contenuti nella variabile `array`
  - casting implicito da stringa ad intero a float (ovvero: dopo aver assegnato `a = 2`, allora sia `a == 2` che `a == "2"` sono veri)
  - comando `print s` (non funzione: i suoi parametri non sono dati come argomenti), dove `s` può essere ottenuta anche concatenando con la virgola (aggiunge uno spazio)
* principali funzioni utilizzabili:
  - uso diretto di array, sia "tradizionali" (con indici numerici) che "associativi" (con indici di qualsiasi tipo)
  - esistono anche le matrici, o in generale array con più di un indice (stessa sintassi di C, Java, Python)
  - `length(s)`: ritorna la lunghezza della stringa `s` (o se `s` è un array, il suo numero di elementi, ma non in tutte le versioni di `awk`)

- `split(s, a, sep)`: tokenizza la stringa `s` nell'array `a` (distruggendolo se già esisteva; il primo indice è 1), usando il separatore `sep` (può non essere dato, e allora si usa `FS`); ritorna il numero di token ottenuti
- `tolower(s), toupper(s)`: ritornano la stringa `s` con le lettere tutte minuscole o tutte maiuscole
- `strtonum(s)`: ritorna il numero rappresentato da `s`; in pratica sarebbe inutile, tanto c'è il cast implicito, ma può essere usato per convertire da esadecimale (`0x`) od ottale (`0`) a decimale
- `int(d)`, `log(d)`, `exp(d)`, `sqrt(d)`, funzioni standard (la `int` non arrotonda, ma tronca)
- è possibile definire funzioni utente, richiamabili da qualsiasi programma (vedere il `man`)
- `printf` come nel C; quasi uguale al `PrintStream.print` del Java: scrive una stringa formattata; `sprintf` versione semplificata del C (ci ritorneremo): ritorna una stringa formattata (senza scriverla su schermo, tipicamente usata per assegnare un valore ad una stringa)
- `gensub(s1, s2, n[, v])`: sostituisce, *ritornando il risultato*, tutte (se `n` è `"g"` o `"G"`) le occorrenze di `s1` (che può essere un'espressione regolare, sia tra apici `"` che tra slash `//`) con `s2` nella variabile `v` (se non data, `v` è `$0`)
- `s2` può contenere `\\&` o `\\0` ad indicare l'intero testo che ha fatto match con `s1`, e anche le backreference per indicare un match con una sottoespressione
- le backreference non possono essere usate in `s1`; per il resto `s1` è una ERE
- se `n` è un numero, allora viene sostituita solo la `n`-esima occorrenza di `s1`
- attenzione: il match è sempre quello più grande possibile
- `substr(s, da [, quanti])`: restituisce la sottostringa di `s` che inizia da `da` (il primo carattere è 1) ed è lunga `quanti` (se non dato, fino alla fine della stringa)
- `index(s, t)`: restituisce il primo indice di `s` in cui comincia la sottostringa `t`, oppure 0 (quindi conta da 1...); stavolta `s` dev'essere una stringa, non una regex (altrimenti, usare `match`)

– **esercizio:** passare ad `awk` 3 file, e farsi stampare i nomi di tali file (guardare il `man`...). Attenzione, i nomi di tali file vanno stampati una volta sola. Usare sia condizioni e programmi che solo programmi

– **esercizio:** prendere lo script dato a lezione 3, passarlo a `gawk` e farsi stampare in output lo stesso script dove, al posto delle

righe del tipo `for unacertavariabile in listadiinteri`, scrive `for ((unacertavariabile=iniziolistainteri; unacertavariabile<=finelistainteri; unacertavariabile++))`. Copiare il risultato su un nuovo file, creare 2 nuove directory, ed eseguire all'interno di ciascuna di esse sia lo script originario che quello modificato. Verificare con `diff` che il risultato sia lo stesso

- **esercizio:** come sopra, ma con la seguente modifica: tutte le volte che c'è un `$` non seguito da {, occorre sostituirlo con `${`. Inoltre, occorre mettere un } dopo un certo numero di caratteri dal `$}`, ovvero quando viene trovato un simbolo non alfanumerico (ad es.: uno spazio, l'andata a capo, uno `/`, un ....). Ad esempio: la riga `chmod ${file}00 dir.$dir/$file` deve diventare `chmod ${file}00 dir.${dir}/${file}`