Sistemi Operativi, Secondo Modulo, Canale M–Z
Riassunto della lezione del 14/03/2016

Igor Melatti

Comandi per elaborare contenuti di testo

- Comando `awk [-F separatore] [--posix] [-f file_awk] [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
  - esiste una terza versione, `mawk` (installato sui computer del laboratorio), che ha differenze minime con `gawk`
  - `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
    - 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:

    [`condizione11` [, `condizione12`]] {`programma1`}

⋮

[`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 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
- per quanto riguarda le *condizioni*, possono essere definite come segue:
  - `var` $\sim$ `/extregex/`: valida solo se il contenuto della variabile `var` 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 anche `-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 comando `last` fa terminare la scansione del solo file attuale
        - il `;` è un separatore e non un terminatore, quindi può essere omesso dopo l'ultima istruzione
        - 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, ma il loro uso può essere cervellotico (vedere `http://stackoverflow.com/questions/3060600/awk-array-iteration-for-multi-dimensional-arrays`)
- `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 $\widehat{\text{FS}}$); 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 (ci ritorneremo): scrive una stringa formattata; `sprintf` versione semplificata del C (ci ritorneremo): ritorna una stringa formattata
- `gensub(s1, s2, n[, v])`: sostituisce, ritornando il risultato, tutte (se `n` è `"g"` o `"G"`) le occorrenze di `s1` 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
- 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}`