Sistemi Operativi, Secondo Modulo, Canale M–Z
Riassunto della lezione del 20/03/2017

Igor Melatti

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
  - 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
    - 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 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 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}`

Altri comandi per elaborare contenuti di testo

- Comando `sed [-e script] [-f file_script] [-r] [-s] [files...]`
  - versione semplificata di `awk`: tutto ciò che si può fare con `sed` si può fare anche con `awk` (il viceversa non vale), ma per alcune cose `sed` è più facile e stringato
  - vale tutto quanto detto nella lezione 7 per `awk`, ma anziché programmi ci sono azioni (più limitate dei programmi, ovviamente)
  - inoltre, le singole righe non vengono tokenizzate, e i files di input sono visti solo come una sequenza di righe (però c'è l'opzione `-s`, che permette di considerarli separati)
  - quindi uno script `sed` è fatto di coppie (condizione, azione)
  - lo script `sed` può essere fornito o direttamente tramite `-e` (o anche senza opzioni), o in un file a parte tramite `-f`
  - le condizioni sono anch'esse più limitate di `awk`; per i nostri scopi basteranno le seguenti:

    n con $n \in \mathbb{N}$: vale vero alla riga n-esima

n, m con $n, m \in \mathbb{N}$: vale vero dalla riga n-esima alla m-esima (incluse)

`/regex/` vale vero se la riga *contiene* un match con la BRE `regex`; `regex` viene considerata ERE (ma senza backreferences) con l'opzione `-r` (ma attenzione: l'opzione `-r` va data *prima* di `-e` o `-f`!)

`$` vale vero all'ultima riga

- grande differenza con `awk`: tutte le righe che *non* soddisfano alcuna condizione vengono stampate in output così come sono (mentre sono ignorate da `awk`)
- altra grande differenza: se ci sono più condizioni vere, viene applicata solo la prima vera
- peròse, come risultato di un'azione, qualche condizione c successiva risulta vera, allora si applica anche l'azione corrispondente a c! Vedremo un esempio a breve
- per quanto riguarda le azioni, ci limitiamo alle seguenti:

  `r filename` appende il contenuto di `filename` alla fine della riga

  `d` cancella la linea

  `i\riga` inserisce `riga` prima della riga

  `a\riga` inserisce `riga` alla fine della riga

  `c\riga` sostituisce `riga` alla riga

  `s/regex/repl/` sostituisce la prima più grande occorrenza di `regex` nella riga con `repl`

  `s/regex/repl/g` sostituisce tutte le più grandi occorrenze di `regex` nella riga con `repl`

  - vale quanto detto per la `gensub` di `awk` (ma occorre l'opzione `-r`...)

- provare il seguente script su un input che contenga una riga `aa`; come si interpreta il risultato?

```
/a{2,3}/ s/a/b/g
/b{3,4}/ s/b/c/g
```

- **esercizio:** scrivere uno script sed che, nelle righe che contengono solo numeri o spazi, sostituisca i numeri con la sola prima cifra del numero stesso; se una riga contiene un identificatore, allora eliminare da esso tutte le cifre; per tutte le altre righe, scrivere ciao sia prima che dopo la stringa

- Comando `tr [-d] [-c] [-t] stringa1 [stringa2]`
  - sempre di meno: qui si possono solo tradurre insiemi di caratteri
  - legge sempre da tastiera e scrive sempre su schermo

  - `stringa1` e `stringa2` sono successioni di caratteri, anche se si possono usare le classi POSIX di tabella 2 in lezione 5
  - con l'opzione `-d` cancella tutti i caratteri in `stringa1`, e `stringa2` non va data
  - con `-c`, `stringa1` viene complementata (si prendono i caratteri *non* presenti in essa); usata soprattutto in combinazione con `-d`; attenzione, se usata con input da tastiera, occorre dare tutto l'input (quindi arrivare fino al CTRL+d) prima di proseguire
  - altrimenti, rimpiazza ogni occorrenza del carattere i-esimo di `stringa1` con l'i-esimo di `stringa2`
  - quindi, `stringa1` e `stringa2` devono essere lunghe uguali
  - se `stringa2` è più corta, l'ultimo suo carattere viene ripetuto fino a colmare la lacuna; ma se viene data `-t` allora è `stringa1` che viene troncata alla lunghezza di `stringa2`
  - nel caso contrario, i caratteri in più di `stringa2` vengono ignorati
  - **esercizio:** vedere cosa succede se `stringa1` contiene caratteri ripetuti
  - **esercizio:** vedere cosa succede veramente con l'opzione `-c`, soprattutto quando si traduce (quando si cancella è facile...)
  - **esercizio:** è sempre vero che con l'opzione `-c` occorre dare prima tutto l'input? può accadere anche senza l'opzione `-c`?

- Comando `uniq [-u] [-d] [-c] [filein [fileout]]`
  - solito per l'input: da file o da tastiera; se viene dato il file di input, si può specificare un file di output (altrimenti, a schermo)
  - elimina le righe identiche *consecutive*
  - con `-c`, per ogni riga stampata dice anche quante volte era ripetuta
  - con `-d`, non stampa le righe singole (mai ripetute)
  - con `-u`, stampa solo le righe singole (mai ripetute)
  - **esercizio:** realizzare `uniq` con un programma `awk`, comprese le opzioni. Si può fare anche con `sed`?

- Comando `sort [-r] [-f] [-n] [-u] [-t sep] [-k POS1[,POS2]] [file...]`
  - ordinamento per righe dei file in input (o da tastiera)
  - ordinamento lessicografico, dove le cifre vengono prima delle minuscole che vengono prima delle maiuscole (ma con `-f` non fa differenza tra maiuscole e minuscole); una parola p che è prefisso di una parola q è minore di di q
  - se le righe contengono numeri, e si vuole che `sort` li interpreti come tali, allora occorre usare l'opzione `-n`

  - ordinamento dal più piccolo al più grande; con `-r` dal più grande al più piccolo
  - con `-u`, stampa una sola volta le righe uguali
  - con `-k`, tokenizza ogni riga come fa `awk` (secondo gli spazi, o secondo il separatore specificato da `-t`), e poi ordina rispetto ai contenuti che vanno dal campo numero `POS1` al campo numero `POS2` (se `POS2` non è dato, allora è uguale a `POS1`)
  - in alternativa, la stessa cosa si può ottenere con `sort +POS1 -POS2` (ma è una notazione "deprecated")
  - **esercizio:** realizzare `sort` con `awk` (leggere il `man` di quest'ultimo, e usare la funzione `asort`...)

- Comando `cut [-d sep] [-f fields] [file...]`
  - versione molto ridotta di `awk`: tokenizza ogni riga e ne stampa i campi dati
  - sempre il solito discorso per l'input: o da file, o da tastiera
  - `-d` serve per ridefinire il separatore
  - `-f` si aspetta una sequenza di range (separati da virgola)
  - **esercizio:** mostrare come ottenere lo stesso risultato con `awk`

- Comando `wc [-c] [-l] [-m] [-w] [-L] [file...]`
  - stampa statistiche su file di testo: numero di righe, di parole e di bytes
  - sempre il solito discorso per l'input: o da file, o da tastiera
  - le opzioni servono a controllare cosa stampare: `-c` numero di bytes, `-m` numero di caratteri (diverso dal precedente se ci sono caratteri accentati), `-l` numero di righe, `-w` numero di parole, `-L` numero caratteri della sola riga più lunga
  - **esercizio:** mostrare come ottenere lo stesso risultato con `awk`