awk: riconoscere ed elaborare pattern

Un “programma” awk è costituito di righe della forma:

pattern {*action*}

Il comando: **awk '***program***' f1 f2** ... :

- legge `f1`, `f2` (per default, la standard input) una riga alla volta
- confronta la riga con ogni *pattern* in *program*, nell’ordine,
- se la riga sta nel *pattern*, compie la *action*

I pattern sono come in `egrep` o ancora più complessi (v. p. 4 e 5).

Esempio: cercare la stringa `ainabl` nelle parole del dizionario:

```
/etc> awk '/ainabl/ { print }' /etc/words  | c
ascertainable    ascertainably    attainable       attainableness
attainably       containable      explainable      gainable
maintainable     obtainable       obtainably       restrainable
retainable       stainable        strainable       sustainable
trainable        unascertainable  unascertainably  unattainable
uncontainable    uncontainably    unexplainable    unobtainable
unobtainably
```

In ogni riga di un programma awk,

- se manca *pattern*, *action* viene fatta per ogni riga di input,
- se manca *action*, è come `{ print }` (invia riga a standard out)

Dunque il primo esempio non è diverso da:

```
/etc> awk '/ainabl/' /etc/words  | c
```

e lo stesso effetto di `cat f` si ha con:

```
/etc> awk '{ print }' passwd
user1::3:1:Franco Baresi:/home/user1:/bin/sh
user2::4:1:Roberto Baggio:/home/user2:/bin/sh
```

awk `-f` *awkfile f1 f2* legge il programma awk da *awkfile*

Programmi awk - campi

awk considera ogni riga di input divisa in ***campi*** dagli spazi bianchi
p. es. l'output di who ha 5 campi:

```
/home/user1> who
carlo      tty2    Nov 23 11:45
stefano    tty6    Nov 23 10:15
```

awk chiama il campo 1 con $1, 2 con $2, ... l'ultimo con $NF:
la variabile NF vale *automaticamente* il numero dei campi sulla riga.
P.es.

```
/home/user1> who | awk '{ print $NF, $1 }'
11:45 carlo
10:15 stefano
/home/user1> ls -l | awk '{ print $NF }'
16
affari
agenda
telefoni
```

Nell'associare $*n* al campo *n*, awk scarta gli spazi iniziali

Si può cambiare il separatore di campo in ***c*** con l'opzione -F***c***. Es.

```
$ awk -F\: '{ print $1, $5 }' /etc/passwd
user1
user2
```

Se si usa -F, gli spazi iniziali faranno parte di $1 $2 ... P.es.:

```
/home/user1> awk -F: '{ print $2 }'
campo1:   campo2
   campo2
^D
```

Output in awk

Altre importanti variabili (automatiche) awk:

`NR` numero di righe in input

`$0` la riga corrente di input.

Per numerare le righe di output, p.es. prodotte da `ls`:

```
/home/user1> ls | awk '{ print NR, $0 }'
1 affari
2 agenda
3 compiti
```

La virgola tra `NR` e `$0` introduce uno spazio nell'output.

Per un più sofisticato output di *arg1*, *arg2* ... secondo un *formato*:

`printf "`*formato*`", arg1, arg2 ...`

formato contiene:

- caratteri, che vengono copiati sull'output (nell'es. spazi, =, `\n`)
  NB: `\n` e `\t` sono *newline* e *tab* rispettivamente
- specifiche di conversione, applicate ordinatamente a *arg1*,*arg2*…
  (%s stampa una stringa, %*n*d un decimale di *n* cifre)

```
$ ls | awk '{ printf "%4d = %s\n", NR, $1 }'
   1 = affari
   2 = agenda
   3 = compiti
```

Un programma per rientrare (*indentare*) il suo input:

```
$ awk '{ printf "\t%s\n", $0 }'
casa, dolce casa
        casa, dolce casa
^D
```

NB: a differenza di `print`, `printf` non va a capo automaticamente

Pattern di awk

Per vedere se qualche utente non ha password, si può usare:

```
$ awk -F: '$2 == ""' /etc/passwd
```

NB: il pattern usa ***operatore di eguaglianza*** == e ***stringa vuota*** ""

Altre forme equivalenti usano:

- l'operatore ***contiene*** ~ (seguito da un pattern tra / e / di *egrep*): `$2 ~ /^$/`;
- la ***negazione*** `!`: `$2 !~ /./` oppure `!($2 ~ /./)`
- la funzione `length`: `length($2)==0`

Nei pattern si possono usare questi operatori relazionali:

`>` `<` `>=` `<=` `==` `!=` `~` `!~`

Per cercare righe più lunghe di 10 caratteri:

```
$ awk 'length($0)>10 { print "Line", NR, "too long" }'
Cantami o diva del Pelide Achille
Line 1 too long
```

I pattern possono contenere operatori aritmetici: `+ - * / %` (resto):
p. es. segnalare le righe con numero di campi dispari:

```
/home/user1> awk 'NF%2 != 0'
casa dolce
casa dolce casa
^D
casa dolce casa
```

La funzione `substr(`*s*`,`*m*`,`*n*`)` (nel pattern o nell'azione) restituisce la sottostringa di *s* che inizia con il carattere *m* ed è lunga *n*. P. es.:

```
$ awk 'substr($2,3,3)=="a" { print substr($1,2,4) }'
oggi sta bene
ggi
```

Pattern booleani

Pattern vero se *p1* **o** *p2* sono veri: *p1*||*p2*

Pattern vero se *p1* **e** *p2* sono veri: *p1*&&*p2*

Esempio:

```
$ awk '($1=="casa" && $2=="dolce") || $2=="bella"'
casa
casa dolce casa
casa dolce casa
casetta bella
casetta bella
^D
```

Pattern `BEGIN` e `END`

I pattern `BEGIN` e `END`:

- entrano in azione rispettivamente prima della prima riga e dopo l'ultima;
- si usano per inizializzazioni e finalizzazioni.

P. es., per cercare utenti senza password serve il separatore `:`

Anziché usare `-F:` si può inizializzare la variabile `FS` di `awk`:

```
$ cat prog
BEGIN { FS = ":" }
$2 == ""
$ awk -f prog /etc/passwd
user1::3:1:Franco Baresi:/home/user1:/bin/sh
user2::4:1:Roberto Baggio:/home/user2:/bin/sh
```

Aritmetica e variabili

Oltre alle variabili “automatiche” `FS NF NR $0 $1 ... $NF`,
awk ha variabili (minuscole) cui assegnare (con =) stringhe o numeri.

Es. totale e media di una colonna

```
$ cat prog
        { s = s+$1 }
END     { print "Tot=" s, "Media=" s/NR}
$ awk -f prog
30
27
21
^D
Tot=78 Media=26
```

NB: le variabili come s vengono inizializzate a `0`.

Per le assegnazioni esistono forme abbreviate:

- “x ***op***= e” sta per “x = x ***op*** e” (p. es. x+=1 invece di x=x+1)
- x++ e ++x stanno per x=x+1, similmente `x--` e `--x`.
- x++ e ++x valgono, come espressioni, rispettivamente
  x prima e dopo l’incremento:

```
$ awk '{ print x++}'
                                        battere Return
0
^D
$ awk '{ print ++x}'
                                        battere Return
1
^D
```

Per contare righe e parole con awk:

```
$ cat wcprog
        { nc += length($0); nw += NF }
END     { print "righe=" NR, "parole="nw, "caratteri="nc }
$ awk -f wcprog
casa dolce casa
home sweet home
^D
righe=2 parole=6 caratteri=30
```

Programmi e istruzioni awk

Nell'esempio precedente si è visto l'uso di `;` come separatore di istruzioni nella parte *azione* di un'istruzione awk.

In effetti sarebbe bastato anche un ritorno a capo:

```
$ cat wcprog
        { nc += length($0)
          nw += NF }
END     { print "righe=" NR, "parole="nw, "caratteri="nc }
```

Altri meccanismi per creare istruzioni composte sono:

```
if (condizione)
            istruzione1
else
            istruzione2
```

```
for (expression1; condition; expression2)
            statement
```

```
while (condition)
            statement
```

Inoltre `break` esce da `while` e `for` che lo racchiudono, e `exit` porta al pattern `END` (se c'è).

All'interno di un'azione si possono usare le parentesi graffe per creare istruzioni composte:

```
BEGIN {
   for (i = 1; i < 10; i++) {
      if (i == 3) break;
      printf "%d\n", i ;
   }
}
```

senza { e } di chiusura, scrive 3!

Esempio

Contare parole doppie in un testo.

Programma awk:

```
/home/user1> cat double
NF > 0  {                       NB: all'inizio lastword vale ""
    if ($1 == lastword)         confronta $1 con ult. parola riga preced.
       printf "double %s at line %d\n", $1, NR
    for (i=2; i<=NF; i++)
       if ($i == $(i-1))
          printf "double %s at line %d\n", $i, NR
    lastword = $NF              aggiorna lastword
}
```

Esecuzione:

```
/home/user1> awk -f double
casa dolce casa
casa dolce dolce
double casa at line 2
double dolce at line 2
```

Array in awk

awk consente di usare array, senza dichiararli.

P. es. per scrivere le righe di input dall'ultima alla prima:

```
/home/user1> cat reverse
        { line[NR] = $0 }
END     {for (i=NR; i>0; i--) print line[i]}
/home/user1> awk -f reverse
Silvia, rimembri ancora quel tempo di tua vita mortale
quando belta' splendea negli occhi tuoi ridenti e fuggitivi
^D
quando belta' splendea negli occhi tuoi ridenti e fuggitivi
Silvia, rimembri ancora quel tempo di tua vita mortale
```

La funzione `split(`*string*`,`*array*`,`*separator*`)`:

- individua in *string* dei campi separati dal carattere *separator*
- assegna ordinatamente ogni campo individuato a un elemento successivo di *array*
- restituisce il numero di campi trovati (la dimensione di *array*)

Esempio: estrarre l'anno da una data in formato `gg/mm/aa`:

```
$ awk '{ split($0,x,"/"); print x[3] }'
11/11/92
92
^D
```

Un possibile uso del valore restituito da `split`:

```
$ awk '{ n=split($0,x,"/"); print x[n] }'
11/11/92
92
```

Funzioni predefinite di awk

Oltre a `split`, `length`, `substr`, sono disponibili le funzioni:

`cos(`*expr*`)`	coseno di *expr*
`exp(`*expr*`)`	esponenziale di *expr*
`int(`*expr*`)`	parte intera di *expr*
`log(`*expr*`)`	logaritmo naturale di *expr*
`sin(`*expr*`)`	seno di *expr*
`getline()`	legge una riga di input, restituisce 0 se alla fine del file, 1 altrimenti
`index(`*s1*`,`*s2*`)`	posizione di *s1* come sottostringa di *s2*, vale `0` se *s1* non è in *s2*

Array associativi

In awk anche le stringhe possono essere indici di array.

Inoltre, la forma: `for (`*var* `in` *array*`)` *statement*
serve per far variare *var* tra gli indici per cui *array*`[`*var*`]` è definito.

Esempio: data una sequenza di coppie *nome valore,*
sommare i valori di ciascun *nome*, p. es.

```
$ awk -f sum
Carlo  40
Luca   90
Carlo 120
Marco  45
Luca   15
^D
Marco 45
Luca 105
Carlo 160
```

Questo si ottiene con il programma awk:

```
$ cat sum
        { tot[$1] += $2 }
END     { for (name in tot) print name, tot[name] }
```

A causa delle tecniche ***hash***, name varia in maniera imprevedibile, inoltre il tempo di accesso a `tot[name]` è costante con name.

Esempio: calcolo frequenza delle 2 parole più numerose nell'input

```
$ cat wordfreq
        { for (i=1; i<=NF; i++) num[$i]++ }
END     { for (word in num) print word, num[word] }
$ awk -f wordfreq | sort  -r -n +1 | awk 'NR<3' | c
casa dolce casa
casa mia casa mia
^D
casa 4     mia 2
```

Esempio: accorciare righe con awk

Problema: scrivere un programma awk che spezzi una riga subito dopo il 20° carattere, appenda uno \ come avvertimento e scriva a capo il resto della riga, giustificato a destra.

```
BEGIN    {
   N = 20                    # inizializza N (max n. caratteri in una riga)
   for (i = 1; i <= N; i++)
      blanks = blanks " "    # blanks è una stringa di N blanks
}
{
   if ((r=length($0)) <= N)
      print
   else { # ciclo: stampa N caratteri alla volta degli r rimanenti
      for (i=1; r>N; r-=N) {             # i: pos. da cui stampare
         printf "%s\\\n", substr($0,i,N)# r: restanti da stampare
         i += N;
      }
   # qui restano da stampare gli ultimi r (r≤N) caratteri di $0, a partire da i,
   # giustificati a destra
      printf "%s%s\n",
             substr(blanks,1,N-r),   # stampa N-r spazi
             substr($0,i)            # stampa il resto di $0
   }
}
```

Spiegazione:

- all'inizio di ogni iterazione del `for`, la riga di input `$0` è stata stampata fino alla pos. `i-1`;
  restano da stampare `r` caratteri a partire dalla pos. `i`;
- dunque, se `r>N`, si stampano `N` caratteri a partire da `i` con `printf "%s\\\n", substr($0,i,N)`
  e si aggiornano `i` e `r`, con `i=i+N` e `l=l-N`
- se invece `r≤N` si stampano gli `N` caratteri restanti a partire da `i`; prima, però, occorrono `N-r` spazi bianchi come riempimento.

Esempio: interazione tra awk e la shell

NB: il programma sulla riga di comando che invoca awk può non essere quotato, o essere diviso in più porzioni quotate.

Si vuole un programma di shell `field`, tale che `field` ***n*** scriva il campo ***n*** della riga di input.

Il file eseguibile `field` contenente la riga:

```
awk '{ print $'$1' }'
```

risolve il problema.

Infatti `$1` non è racchiuso tra `'` e `'` ed è quindi interpretato dalla shell (che lo rimpiazza con il 1° argomento di `field`) e non da awk.

Esercizi

1. data un’agenda con righe:

   ```
   cognome nome indirizzo città numero
   ```

   scrivere un programma awk per stampare le righe con `nome` uguale a `Giovanni` e `città` uguale a `Milano`

2. scrivere un programma awk per contare il numero di volte che le parole `il`, `lo`, `la` compaiono in un testo

3. scrivere un programma awk per eliminare la parola `io` da un testo

4. scrivere un programma awk per sostituire in un testo la parola `io` con la parola `I` e contare le sostituzioni effettuate

5. scrivere un programma awk per distribuire le parole del testo in input su righe di max 72 caratteri

6. come (5), ma con le righe “giustificate” (cioè fine riga sempre sulla colonna 72, grazie all’aggiunta di spazi tra le parole)