Cenni sulle Espressioni Regolari

Calzoni Pietro aka *Calzo*

12 agosto 2007

Sommario

Questo breve documento ha come obiettivo di spiegare brevemente la sintassi delle espressioni regolari accettate dai programmi tipicamente presenti in ambiente UNIX (accessibili via shell) confrontandoli all'occorrenza con quanto definito da *perl*. Questo articolo è frutto dell'esperienza relativa fatta in modo autodidattico; non è (e non vuole essere) un testo completo ed esaustivo sull'argomento, ma vuole solo definire le basi per un eventuale studio più approfondito. Mi scuso anticipatamente per eventuali (inevitabili) errori o imprecisioni.

L'articolo documenta il workshop tenuto a Mantova in occasione del Linux Day 2006 su questo argomento.

Indice

1 Intro&Info

Una espressione regolare (abbreviata con RegEx, d'ora in poi) è una stringa di caratteri costruita con una specifica sintassi attraverso la quale si possono rappresentare determinati insiemi di stringhe e procedere eventualmente all'elaborazione del testo in esame. Le RegEx sono tipiche del mondo UNIX e benchè siano note da tempo, il loro massimo splendore lo hanno conosciuto attraverso il linguaggio *perl* il quale, facendone un uso intensivo, ha permesso alle RegEx di diventare uno standard di fatto.

La diffusione di questi strumenti per l'elaborazione del testo ha fatto si che le RegEx siano state adottate in moltissimi altri linguaggi come C/C++, Java,

Ruby, addirittura Visual Basic[1], ecc. Il principale utilizzo delle RegEx resta comunque la ricerca e l'ordinamento del testo. Anche se si dividono in Basic ed Extended RegEx, tipicamente vengono utilizzate quelle di tipo Extended. Qualche informazione in più la si trova su

- http://it.wikipedia.org/wiki/Espressione_regolare: RegEx condite con un po´di storia
- man 3 e 7 di regex
- man dei comandi come *grep*, *sed*, ecc

2 Metacaratteri

I metacaratteri costituiscono la sintassi vera e propria di una espressione regolare e permettono di definire le (talvolta complesse) regole per l'analisi del testo. Si ponga attenzione al fatto che alcuni metacaratteri assomigliano ai più comuni "caratteri jolly", ma con essi non poco a che fare, pur avendo un significato simile.

Metacaratteri standard

^ Posto all'inizio della stringa, indica con quali caratteri il testo in esame deve cominciare; per esempio `^A` troverà tutte le stringhe che cominciano con il carattere `A`.

$ Posto alla fine della stringa, indica con quali caratteri il testo in esame deve terminare; per esempio `A$` troverà tutte le stringhe che finiscono con il carattere `A`. Nota `$` e `^` sono detti anche àncore

. Indica un qualsiasi carattere; per esempio l'espressione "`c.s.`" soddisferà tutte le occorrenze di `cosa`, `case`, `casa`, ecc

\ Posto prima di un metacarattere lo rende un carattere normale, mentre prima di determinati caratteri può definire particolari gruppi, come si vedrà in seguito; per esempio "`\.`" indica che occorre considerare il carattere "`.`" e non il metacarattere "`.`"

? Indica che il carattere (o il raggruppamento[2]) che lo precede può comparire al più una volta; per esempio "`cas?i?ne`" soddisfa le occorrenza di `casine`, `cane`, ecc

* Indica che il carattere (o il raggruppamento) che lo precede può comparire per un numero di volte maggiore o uguale a 0; per esempio l'espressione `io*` soddisfa le stringhe `i`, `io`, `ioo`, `iooo`, ecc

[1] MS Visual Basic supporta le espressioni regolari POSIX. La loro introduzione dovrebbe risalire alla versione 6 di Visual Studio

[2] I raggruppamenti non sono accettati da tutti i programmi

+ Indica che il carattere (o il raggruppamento) che lo precede deve comparire minimo una volta; per esempio "`io+`" soddisfa `io`, `ioo`, `iooo`, ecc

{ } Indicano un intervallo definito di ripetizione di ciò che precede

- {*n*} *n* è un numero intero positivo e indica che ciò che precede deve comparire esattamente *n* volte; per esempio `io{3}` soddisferà solo `iooo`
- {*n*,} *n* è un numero intero positivo e indica che ciò che precede deve comparire minimo *n* volte; per esempio `io{2,}` soddisferà `ioo`, `iooo`, ecc
- {,*n*} *n* è un numero intero positivo e indica che ciò che precede deve comparire massimo *n* volte; per esempio `io{,2}` soddisferà `i`, `io`, `ioo` e basta
- {*n*,*m*} *n* e *m* sono numeri interi positivi tali che $m>n$ e indicano che ciò che precede deve comparire minimo *n* e massimo *m* volte; per esempio `io{2,3}` soddiferà `ioo`, `iooo` e basta

[ ] Indica un insieme di caratteri che in una determinata posizione soddisfano l'espressione

- [1-4] il carattere "`-`" posto tra due caratteri indica che l'insieme dei caratteri va dal carattere di sinistra a quello di destra; in questo caso l'insieme è costituito dai caratteri 1, 2, 3, 4. Nota: una scrittura del tipo [1-4-8] non è valida
- [1-] il carattere "`-`" posto alla fine indica che il "`-`" fa parte dell'insieme
- [^x] l'accento circonflesso "`^`" all'inizio, nega il contenuto dell'insieme; in questo caso si considereranno tutti i caratteri diversi da `x`. Nota: se "`^`" fosse all'interno dell'insieme insieme, sarebbe da intendesi come un normale carattere (per esempio `[x^]` soddisferebbe sia le occorrenze di `x` che di `^`); se "`^`" fosse l'unico carattere tra parentesi allora occorrerebbe definirlo con [\^] per evitare che línterprete restituisca errori.
- [\x*n*] definisce un carattere dato il numero *n* corrispondente definito in formato esadecimale; per esempio `[\x60-\x65]` indica i caratteri compresi tra `a` ed `e`
- [\t] indica la tabulazione

\<*regex*\> permette di trovare esattamente l'espressione *regex*

| É l'equivalente di *OR* e permette di definire due o più espresioni alternative; per esempio `io|tu|l[eu]i` soddisferà tutte le occorrenze `io`, `tu`, `lei` e `lui`

Classi POSIX

Lo standard POSIX ha introdotto altri metacaratteri che identificano specifici gruppi definiti anche classi. La Tabella 1 riporta l'elenco:

Tabella 1: Classi POSIX delle espressioni regolari.

POSIX	Standard	Descrizione
[:alnum:]	[0-9a-zA-Z]	Caratteri alfanumerici tranne "_"
[:alpha:]	[a-zA-Z]	Caratteri alfabetici (lettere)
[:digit:]	[0-9]	Caratteri numerici (cifre)
[:xdigit:]	[0-9a-fA-F]	Cifre esadecimali
[:ascii:]	[\x01-\x7F]	Caratteri ASCII a 7 bit
[:lower:]	[a-z]	Lettere minuscole
[:upper:]	[A-Z]	Lettere maiuscole
[:punct:]	[!"#$%&'()*+,-./:;<=>?@[\\]^_‘{|}~]	Caratteri di punteggiatura
[:space:]	[\n\r\t\x0B]	Tutti gli spazi
[:blank:]	[\t]	Spazi orizzontali
[:graph:]	[^ \t\f\v\n\r]	Caratteri stampabili eccetto spazi
[:print:]	[^\t\f\v\n\r]	Caratteri stampabili
[:cntrl:]		Caratteri di controllo

Vengono usati tipicamente all'interno dei metacarateri "[ ]", per esempio `[[:alnum:]]`.

Gruppi e caratteri speciali

Esistono particolari metacaratteri che permettono di definire dei gruppi di caratteri predefiniti. Attenzione però che non sono riconosciuti tutti e da tutti i programmi, per esempio *sed* accetta `\w` e `\b`, ma non `\d`, mentre *perl* accetta tutto.

\w Indica tutti i caratteri alfanumerici compreso l'under score "_". Il suo negato è \W.

\s Indica tutti i caratteri di spazio tra cui anche la tabulazione. Il negato è \S

\d Indica tutti i caratteri numerici, ossia equivale a [0-9]. Il negato è \D

\b Da solo individua inizio e fine di una parola, mentre, se seguito da dei caratteri, soddisfa la RegEx se la parola comincia con i caratteri precisati; per esempio `\bcar` è soddisfatta da prole come `carro`, `cartello`, ma non da `autocarro`.

\a	Dove *a* può essere `n`, `r`, `t`, `f` e `v`; sono caratteri che hanno la stessa valenza dei caratteri passati alla stringa di formattazione della finzione *printf* del C; indicano rispettivamente return, carriage-return, tab, form-feed e vertical-tab.
\x*n*	*n* indica un valore esadecimale al quale corrisponderà un determinato carattere; per esempio `\x61` indica il carattere `a`.
xyz	*x*, *y* e *x* sono le tre cifre di un numero in formato ottale; usato soprattutto per la rappresentazione di caratteri di escape (`\033` ossia 27 in decimale)
\E	Indica il caratteri di escape "ESC"

3 Struttura di una RegEx

Con le nozioni acquisite fino ad ora si è in grado di capire e costruire una RegEx più o meno complessa. Tale espressione però sarebbe adatta solo a certi programmi che tipicamente operano su testi per effettuare ricerche più o meno avanzate. I programmi che però fanno uso intensivo delle RegEx e che possono anche operare sul testo individuato, richiedono che l'espressione regolare vera e propria venga definita in uno di modi seguenti:

$$x/regex/expr/y$$

$$x\{regex\}\{expr\}y$$

Le due espressioni sono equivalenti con la differenza che la prima è utilizzata dalla maggior parte dei programmi (come per esempio *sed*), mentre la seconda è usata da *perl* (che comunque accetta anche la prima forma).

Per quanto riguarda i termini, *x* e *y* indicano delle opzioni relative a come operare quando avviene il match della *regex* e a come eventualmente muoversi nel testo per effettuare la ricerca (termini molto comuni sono *x*=`s` che indica di effettuare la sostituzione e *y*=`g` che indica di non fermarsi alla prima occorrenza, ma di fare una ricerca globale). *regex* è l'espressione da soddisfare, mentre *expr* è l'espressione da sostituire se *regex* è soddisfatta.

Programmi

Verrà ora data una breve descrizione dei programmi che verranno usati d'ora in avanti con le opzioni con cui vengono lanciati

perl	Practical Extraction and Report Language. Non verrà approfondito completamente, ma verranno solo analizzate le potenzialità relativamente all'elaborazione di RegEx a linea di comando. Verrà lanciato con le opzioni *-pe* che indica di leggere il file che segue riga per riga e applicando la RegEx. Può essere usato come interprete comandi.

sed Stream EDitor. Esegue (relativamente) semplici trasformazioni del testo. Verrà lanciato con *-r* che indica che la stringa che segue è una espressione regolare estesa. Può essere usato come interprete comandi in uno script, anche se sconsigliato vista l'estrema complessità che possono raggiungere alcuni script (vedere *info sed*)

gawk Pattern scanning and processing language. Versione GNU di awk, è un programma abbastanza avanzato con alcune funzioni interne relativamente versatili che gli permettono di diventare un buon interprete comandi. L'uso tipico di *gawk* è comunque di frazionare il testo tramite dei separatori di campo (di default è lo spazio) definiti co l'opzione *-F*. La stampa del testo viene però fatta attraverso specifici comandi racchiusi tra apici e parentesi graffe `'{` *comandi* `}'`

tr Translator. Permette fondamentalmente di trasformare un particolare set di caratteri in un altro.

Le RegEx vanno quindi passate (come stringhe) ai vari interpreti in uno dei seguenti modi:

apici singoli: ' ' permettono indicano a bash che la stringa da passare al comando è costituita esattamente dai caratteri compresi tra i due apici. Problemi potrebbero derivare dal fatto che se nella stringa compare una variabile, la stringa deve essere passata spezzandola:

esempio: `sed -r 's/‘'$X'‘/(found)/g'` se per esempio `X=GNU` e la stringa in esame fosse per esempio "`‘GNU‘`", allora verrebbe sostituita con "`(found)`".

questo è il metodo più usato e consigliabile

apici doppi: " " permettono di definire una stringa nella quale possano essere valutate anche le variabili di ambiente o l'output dei comandi:

esempio: `sed -r "s/\‘$X\‘/(found)/g"` se per esempio `X=GNU`, avremo lo stesso risultato dell'esempio precedente. In questo caso però occorre dire a *bash* che "`‘`" è un normale carattere anteponendo la "\", altrimenti la shell cercherà di eseguire il contenuto di `X` come se fosse un comando.

diretto: la scrittura diretta è sconsigliata anche perchè in certi casi è comunque necessario ricorrere agli apici

esempio `sed -r s/\‘$X\‘/\(found\)/g` se per esempio `X=GNU`, avremo lo stesso risultato dell'esempio precedente. In questo caso però occorre dire a *bash* anche che la parentesi è un carattere.

questa modalità è la più complessa ed inutilmente prolissa, quindi sconsigliata

3.1 Raggruppamenti e richiami

Per effettuare una più avanzata elaborazione delle stringhe è spesso necessario effettuare dei raggruppamenti. Per fare ciò si ricorre alle parentesi tonde. Ogni raggruppamento viene memorizzato in sequenza e numerato in modo crescente e può essere richiamato sia all'interno della RegEx che nella stringa di sostituzione. Per richiamare quanto memorizzato si può procedere fondamentalmente in due modi:

$\backslash n$ con n numero intero non negativo che indica il numero progressivo con cui sono state memorizzate le porzioni di RegEx fino ad un massimo di 9. Tipicamente 0 indica tutta la parte i stringa che ha soddisfatto la RegEx (può comunque dipendere dal programma). Questa notazione è la più usata (per esempio da *sed*).

$\$n$ il significato è lo stesso del precedente, ma viene usato da programmi come *perl* e $(g)awk$. Si noti che in se in perl si usano le RegEx con la notazione di *sed*, allora è auspicabile usare $\backslash n$. Nota: in *perl* n non ha limiti

esempio 1: `sed -r 's/([0-9])(\w)/\0\2\1/g'` con questa RegEx se una stringa comincia per un numero e una lettera, allora la lettera verrà aggiunta al risultato della RegEx la lettera seguita dal numero. L'ipotetica stringa `1w2d3` diventerebbe `1ww12dd23`.

esempio 2: `sed -r 's/(([0-9])\w)/\1\2/g'` con questa RegEx viene aggiunto il numero dopo la lettera, ma in questo caso il raggruppamento è nidificato. Ciò che cambia è che ora il numero memorizzato non è più indicato con il numero 1, ma con il numero 2. Quindi, come è ovvio, l'indice più basso è dato alla parentesi più esterna. Sempre a titolo di esempio, la sintassi alternativa per *perl* è `perl -pe 's{((\d)\w)}{$1$2}g'` .

esempio 3: `sed -r 's/(\w{,4})\s+\1/\1/'` in questo esempio si nota come un raggruppamento può essere usato nella stessa *regex* per individuare, in questo caso, la prima ripetizione multipla di parole (nell'esempio parole di massimo 4 caratteri) separate da uno o più spazi; nell'esempio stringhe come "`casa casa mia`" o "`1  1 2 2`" diventeranno "`casa mia`" e "`1 2 2`" (notare che nella regex non c'è la `g` finale e quindi "`2 2`" rimane).

3.2 Condizionamenti

In alcuni casi è possibile avere delle espressioni le quali vanno elaborate solo a fronte del fatto che siano verificate alcune condizioni. I metodi per realizzare ciò variano tipicamente da programma a programma. Di seguito quindi verranno trati alcuni metodi specifici per le applicazioni tipiche, senza entrare particolarmente nel dettaglio:

PERL

Perl è un linguaggio vero e proprio che permette la definizione anche di costrutti *if...then* anche applicati alle espressioni regolari mediante l'operatore `~=`. Di seguito però verranno solo analizzati dei metodi per condizionare una espressione direttamente con una opportuna sintassi all'interno della RegEx. Ciò viene fatto soprattutto come confronto con gli altri programmi.

(?:*regex*) Raggruppamento senza catturare. Permette di verificare se una certa espressione è verificata, ma non ne memorizza il contenuto. É utile soprattutto quando si vuole avere una particolare ripetizione di una *regex* il cui valore non è noto a priori.

esempio: `perl -pe 's/(?:\w\w?)\s+(.+)/\1/g'` è tale per cui le stringhe "`in casa`" o "`a chi?`" diventino rispettivamente "`casa`" e "`chi?`". Si nota che \1 si riferisce alla seconda parentesi contenente l'espressione "qualsiasi carattere ripetuto indefinitamente" `(.+)` e non al raggruppamento `(?:`*regex*`)`

(?<=*regex*) Detto anche positive look-behind, se *regex* è verificata allora la RegEx completa viene applicata, ma le eventuali sostituzioni toccheranno solo la restante RegEx al di fuori della look-behind.

esempio: `perl -pe 's{(?<=GNU/)[Ll]inux}{TUX}g'` questa RegEx sostituirà la parola "`TUX`" a "`Linux`" solo se "`Linux`" è preceduto da "`GNU/`". `GNU/Linux` diventerà `GNU/TUX` e come si nota la sostituzione non tocca quanto definito dalla look-behind, come volevasi dimostrare.

(?<!*regex*) Detto anche negative look-behind, se *regex* NON è verificata allora la RegEx completa viene applicata, ma le eventuali sostituzioni toccheranno solo la restante RegEx al di fuori della look-behind.

esempio: `perl -pe 's{(?<!GNU/)[Ll]inux}{TUX}g'` questa RegEx sostituirà alla parola Linux la parola TUX solo se Linux NON è preceduta da GNU/. La stringa "`GNU/Linux è opensource e Linux è il famoso pinguino`" diverrà "`GNU/Linux è opensource e TUX è il famoso pinguino`".

SED

In *sed* più che di condizionamenti è meglio parlare di interrogazioni vere e proprie. Per prima cosa si noti che in *sed* (come in altri programmi) è possibile definire più di una espressione regolare separando le varie espressioni con ";" come nell'esempio che segue:

esempio: `sed -r 's/a/b/g; s/bc/--/g' <<< "acbca"` restituisce come output "`----b`" in quanto applica sulla stringa in ingresso la prima RegEx trasformandola in "`bcbcb`" e quindi applica la seconda.

Per esprimere una condizione invece occorre definire una prima espressione regolare contenuta tra due "/" senza punto e virgola[3] che separi ciò che segue, ossia la vera RegEx:

/regex/ Interrogazione positiva. Se *regex* è verificata, allora si procede

esempio `sed -r '/GNU\// s/[Ll]inux/TUX/g'` in analogia con gli esempi in *perl*, "`linux`" verrà sostituito da "`TUX`" se nella stringa compare "`GNU/`". Attenzione che "`GNU/`" può comparire in qualsiasi punto della stringa.

/regex/! Interrogazione negativa Se *regex* NON è verificata, allora si procede

esempio `sed -r '/GNU\//! s/[Ll]inux/TUX/g'` "`linux`" verrà sostituito da "`TUX`" se nella stringa NON compare "`GNU/`". Anche in questo caso "`GNU/`" può comparire in qualsiasi punto della stringa.

GAWK

Gawk è un programma più strutturato e, tutto sommato, la sua sintassi è di più alto livello. Come *perl*, anche *gawk* permette la definizione di costrutti *if*, variabili interne, ecc ed inoltre contiene alcune funzioni interne che permettono una agevole manipolazione del testo. Come detto, però, la funzione basilare di *gawk* è di frammentare il testo a fronte di un separatore di campo. In precedenza è stato detto che le istruzioni di *awk* vengono passate tra apici e così è anche per le RegEx. Le interrogazioni hanno una sintassi simile a quella di *sed* con la differenza che non viene accettata l'interrogazione negativa.

esempio `gawk -F__ '/[[:lower:]]/ { print $1" "$2" "$3 }'` questa RegEx indica che se nella stringa compare un carattere minuscolo, allora si procede con la separazione dei campi. Per come è scritta l'espressione in questo esempio, una stringa "`x_y_z`" viene riproposta (e quindi stampata) uguale perchè compare almeno un carattere minuscolo, ma non vi è il separatore corretto (quindi `$1=$0=x_y_z`). La stringa "`T__H__D`", pur avendo il corretto separatore non verrà processata e il risultato sarà una stringa nulla, mentre "`t__H__d`" soddisfa tutte le richieste e restituisce "`t H d`".

4 Esempi

Gli esempi che seguono sono relativamente semplici e permettono di vedere la differenze tra i programmi trattati.

[3] L'interposizione di ";" tra le due RegEx farà sì che *sed* restituisca un errore

Stampa di un file di testo (*perl*, *sed*, *gawk*) è un esempio tendenzialmente inutile, ma mostra subito una differenza tra *perl* e *sed*. Supponendo di avere un file di testo file.txt, l'output deve essere lo stesso di cat file.txt:

perl `perl -pe 's///' file.txt` oppure `perl -pe 's///' < file.txt`

sed `sed -r 's/ / /' file.txt` si noti che gli spazi in sed sono fondamentali

gawk `gawk '{ print }' < file.txt`

Cambiare formato alla data si vuole passare dalla data definita in anno-mese-giorno (aaaa-mm-gg) in giorno-mese-anno (gg/mm/aa):

perl `perl -pe 's/\d\d(\d\d)-(\d\d)-(\d\d)/\3\/\2\/\1/g'` dove l'ipotetica data 1981-04-01 diventerebbe 01/04/81

sed `sed -r 's/[0-9]{2}([0-9]{2})-([0-9]{2})-([0-9]{2})/\3\/\2\/\1/g'`

gawk `gawk -F- '{ printf $3"/"$2"/"substr($1,3) }'` in questo caso occorre ricorrere alla funzione interna *substr* per cancellare parte del numero relativo all'anno.

Upper Case si vuole portare tutti i caratteri da minuscoli a maiuscoli:

sed improponibile, ma possibile come mostrato negli esempi in *info sed*. L'espressione è inutilmente lunga e complessa

gawk `gawk '{ print toupper($0) }'` anche in questo caso si ricorre alle funzioni interne di *gawk*

tr `tr [a-z] [A-Z]` decisamente più elegante

Evidenziare un testo supponendo di avere un testo con errori di ripetizione multipla, si vuole evidenzaire in rosso tale ripetizione. Si supponga a tal proposito di aver definito le variabili RED (`RED=$'\E[31m'`) e DEF (`DEF=$'\E[0m'`) che indicano rispettivamente il colore rosso e il colore di default del testo della shell; si supponga di avere un file di testo `file.txt` con alcuni errori di ripetizione, per esempio:

```
GNU/Linux è un un  un sistema operativo opensource.
SlackWare è la prima distribuzione Linux rilasciata;
è compilata personalmente da da P. J. Volkerding.
```

sed `sed -r 's/((\b[[:alnum:]]+)(\s+\2)+)/'$RED'\1'$DEF'/g' file.txt | more` oppure per salvare in un file sostituire a "`| more`" la redirezione "`> output.txt`". Le parole ripetute si evidenzieranno di rosso. Notare che se non ci fosse "`\b`" verrebbero rese rosse anche la "`o`" finale e iniziale rispettivamente di "`operativo`" e "`opensource`", mentre "`(\s+\2)+`" indica di cercare almeno una ripetizione.

La stringa è analoga anche per *perl*, mentre con *gawk* ottenere esattamente lo stesso risultato potrebbe non essere immediato. L'output sarà:

```
GNU/Linux è un un  un sistema operativo opensource.
SlackWare è la prima distribuzione Linux rilasciata;
è compilata personalmente da da P. J. Volkerding.
```

Credits

L'articolo è stato scritto con LyX, (www.lyx.org) front-end multipiattaforma per LaTeX sotto SlackWare 10.2 (www.slackware.com) con la quale sono stati testati tutti gli esempi riportati.