Regular Expressions

mobytrick

27 agosto 2014

Sommario

Una (breve) introduzione alle Regular Expressions. Dalle basi al *backreferencing*.

1 Espressioni regolari

1.1 Preambolo

Le *espressioni regolari* (`regular expressions` in inglese e spesso abbreviate in `RE`) descrivono in maniera sintetica, tramite una particolare sintassi, un modello testuale ed eseguono dei confronti per scoprire se una stringa di caratteri è conforme o meno. Un esempio banale, ma comune, di `RE` è la descrizione delle targhe automobilistiche italiane. L'identificativo dei veicoli è conforme al seguente modello:

lettera lettera cifra cifra cifra lettera lettera

Tale descrizione si basa sul fatto che si sappia che cosa si intenda per "*cifra*" e "*lettera*". Per `cifra` si intende 1 delle usuali 10 cifre arabe. Per `lettera` si intende 1 lettera maiuscola dell'alfabeto internazionale, che ne conta 26.

Storicamente le `RE` esistono da molti anni. Poi col passare del tempo le `RE` sono state ampliate e potenziate tanto che il nucleo originario viene indicato come `BRE`, ovvero `Basic RE` mentre lo stato dell'arte viene designato come `ERE`, `Extended RE`. A partire da queste le applicazioni che in maniera più o meno diretta si rifanno alle `RE` non sempre hanno seguito una strada comune, o per lo meno condivisa. Il risultato: un vero e proprio ginepraio.

All'atto pratico esistono i cosiddetti "*RE engines*" ovvero software in grado di eseguire l'esame delle `RE` di complessità arbitraria ed in grado di eseguire la validazione (aderenza al modello) e la ricerca (presenza del modello) dei

dati forniti. Nel mondo Unix ci sono vari tool che permettono oppure si basano sull'uso delle `RE`. Un elenco certamente non esaustivo comprende `grep`, `flex`, `awk`, `vi`, `sed`. Ma lo strumento che di gran lunga sfrutta le potenzialità delle `RE` è il PERL. Presente pure nel mondo Windows.

1.2 I primi passi

Supponiamo di avere una certa quantità di dati testuali e di voler ricercare la presenza di una determinata stringa. Lo strumento scelto esegue il compito affidatogli segnalando le eventuali conformità. Aiutiamoci con un esempio ed operiamo in ambiente Linux. Il file da esplorare si chiama `testo.txt` e contiene i seguenti dati:

```
Nord
Sud
Est
Ovest
Ernesto
estate
foresta estesa
```

Si desidera sapere se la stringa `est` è presente o meno. Come strumento si utilizza il comando `grep`. La scelta non è casuale. `grep` è un acronimo che sta per "*general RE printer*". Il comando da utilizzare è il seguente:

```
grep --color red -E "est" testo.txt
```

la cui anatomia è esplicata nella tabella 1 a pag. 2.

`grep`	comando
`--color red`	stampa in rosso il risultato
`-E`	abilitazione all'uso delle RE estese
`"est"`	stringa da ricercare (apici esclusi)
`testo.txt`	file da esplorare

Tabella 1: Anatomia di grep

Il comando `grep` lavora "*a righe*". Se trova la stringa ricercata, stampa la riga che la contiene. Quindi molto opportunamente conviene far stampare il risultato con un colore diverso. Questo è il risultato:

Ovest
Ernesto
estate
foresta estesa

La riga che contiene `Est` non viene riprodotta sull'output poiché `Est` ed `est` sono differenti. Ciò è da tenere a mente, specialmente da coloro che sono abituati ad usare Windows (in questo sistema operativo non viene fatta distinzione tra maiuscole e minuscole, con l'unica eccezione delle password).

Dall'esempio si ricava una regola molto importante. Nel campo delle `RE` ogni carattere rappresenta se stesso. Quindi, ad esempio, 1 rappresenta quel carattere; lo stesso vale anche per `%`. Ma i seguenti 14 caratteri:

```
( ) [ ] { } ? * + $ ^ . \ |
```

hanno un comportamento differente. Sono noti come "*metacaratteri*" in quanto utilizzati nelle sintassi delle `RE`. Se devono essere usati in senso letterale, vanno fatti precedere dal carattere `\` (noto come carattere "*escape*").

Supponiamo di avere un modello così composto:

```
^est
```

Se applicato al file `testo.txt` produce questo risultato:

estate

Il metacarattere `^` indica l'*inizio riga*. `^est` viene interpretato come ricerca delle righe che iniziano con la stringa indicata.

Supponiamo di avere un modello così composto:

```
d$
```

Se applicato al file `testo.txt` produce questo risultato:

Nord
Sud

Il metacarattere `$` indica la *fine riga*. `d$` viene interpretato con ricerca delle righe che finiscono con la stringa indicata.

I metacaratteri `^` e `$`, noti singolarmente come *àncora*, possono essere usati congiuntamente. Il modello:

```
^Est$
```

serve ad individuare le righe che contengono la sola stringa indicata. Applicato al file `testo.txt` produce l'ovvio risultato:

Est

Supponiamo di avere un modello così composto:

```
ar.a
```

allora le stringhe `area`, `aria`, `arma` ed `arca` sono tutte conformi al modello. Il significato del metacarattere `.` (punto) è quello di rappresentare *1 carattere*, qualsiasi esso sia. Pertanto la stringa `aroma` non è conforme al modello.

Supponiamo di avere un modello così composto:

```
ar[ei]a
```

allora le stringhe `aria` ed `area` sono conformi, mentre non lo sono `arma` ed `arca`. Il significato dei metacaratteri `[]` è quello di rappresentare un unico carattere, purché presente all'interno delle parentesi. `[]` rappresentano quindi una *scelta*.

Talvolta all'interno delle parentesi quadre è necessario scrivere diversi caratteri. Se questi sono contigui è possibile usare una notazione più abbreviata. Resta da definire il concetto di contiguità. Per le lettere il concetto è abbastanza ovvio. Le lettere `m`, `n` ed `o` sono contigue (per rendersene conto è sufficiente recitare l'alfabeto). Anche nel campo delle cifre il concetto è facile da applicare. Le cifre 4, 5 e 6 sono contigue. Forti di queste precisazioni, il modello:

```
[A-Z]
```

rappresenta una sola lettera maiuscola dell'alfabeto internazionale, mentre il modello

```
[0-9]
```

rappresenta una sola cifra. L'abbreviazione si ottiene frapponendo il carattere `-` (trattino) tra gli estremi. Si possono indicare anche più intervalli. Il modello:

```
[a-il-vz]
```

rappresenta 1 sola lettera minuscola dell'alfabeto italiano perché a conti fatti risultano escluse le lettere `j`, `k`, `w`, `x` ed `y`.

Accanto agli intervalli, talvolta è necessario escludere la presenza di certi caratteri. In tal caso si fa entrare in gioco il metacarattere `^`, già visto come indicatore di inizio riga. Qui però ha il significato di *negazione*. Il modello:

```
ar[^i]a
```

descrive un qualsiasi stringa che contiene i caratteri `ar`, 1 carattere qualsiasi purché diverso da `i`, ed un'ulteriore presenza di `a`. Pertanto le stringhe `arca`, `area` ed `arma` sono conformi mentre non lo è `aria`. Naturalmente è possibile negare gli intervalli. A scanso di equivoci il carattere di negazione (`^`), se usato, va posto davanti a ciò che si intende escludere. Quindi:

```
[^A-J]
```

serve ad eliminare una sola lettera maiuscola compresa tra A e J, estremi inclusi.

La negazione è subdola e va interpretata in maniera corretta. Data la `RE` `q[^u]`, si mediti sul sottile distinguo riportato nella tabella 2 a pag. 5.

`q` non seguita da `u`	**errata**
`q` seguita da un carattere diverso da `u`	**corretta**

Tabella 2: Interpretazione della negazione

Talvolta esiste la necessità di ripetere una parte del modello più volte. Anziché lavorare di copia-ed-incolla, pratica sempre possibile, si usa l'apposita sintassi che fa ricorso alle `{}` (parentesi graffe). I metacaratteri `{}` operano quindi la *ripetizione*. Ad esempio:

```
a{5}
```

è la notazione per reiterare per 5 volte il carattere `a`. Così facendo si aumenta la comprensibilità delle `RE` perché `a{5}` mette in evidenza la ripetizione in maniera migliore di `aaaaa`. Si badi bene che il meccanismo di ripetizione si applica al solo elemento precedente. Se scrivo:

```
a[0-9]{2}
```

la ripetizione si applica alla sola cifra (`[0-9]`). Volendola estendere anche al carattere `a` la notazione è leggermente differente:

```
(a[0-9]){2}
```

Le `()` -parentesi tonde- sono i metacaratteri che operano il *raggruppamento* (in inglese: *grouping*). Ma hanno un effetto collaterale, detto "*capturing*" (vedi a pag. 10).

Talvolta, anziché il numero preciso, si conoscono i limiti entro i quali ci si può giostrare. In tal caso la sintassi della ripetizione assume la forma:

$$\texttt{\{limite}_{inf}\texttt{,limite}_{sup}\texttt{\}}$$

Non sempre è possibile conoscere con precisione il numero delle ripetizioni. Però è possibile indicare un numero minimo ed un massimo, oppure uno solo dei due ed anche nessuno dei due. Scrivendo:

```
r{,1}
```

si sottintende che il numero minimo di ripetizioni è pari a 0 (zero) ed il massimo è 1. Le stringhe `aia` ed `aria` sono conformi, mentre non lo è `arria`. Se invece a mancare è il massimo, il valore default è ∞, da intendersi non tanto come infinito quanto come qualsivoglia. Quindi:

```
[a-z]{3,}
```

descrive una stringa composta da 3 a qualsivoglia numero di lettere minuscole.

Una volta assimilati i concetti di scelta (`[]`) e ripetizione (`{}`), si può rimodulare in termini rigorosi la `RE` che descrive le targhe automobilistiche italiane. Partendo dalla descrizione letterale già fatta:

lettera lettera cifra cifra cifra lettera lettera

si passa ad una descrizione più concisa:

```
lettera{2} cifra{3} lettera{2}
```

per poi usare la sintassi `RE`

modello	# ripetizioni	abbreviazione
`{,}`	da 0 a ∞ volte	`*`
`{0,1}`	nessuna volta oppure 1 sola	`?`
`{1,}`	da 1 a ∞ volte	`+`

Tabella 3: Ripetizioni

```
[A-Z]{2}[0-9]{3}[A-Z]{2}
```

Il meccanismo delle ripetizioni va esplorato nella sua interezza. Alcuni tipi di ripetizione si presentano con particolare frequenza, per cui sono state introdotte delle abbreviazioni. La tabella 3 a pag. 7 le sommarizza.

L'asterisco è l'abbreviazione più nota ed utilizzata (senza sapere che fa parte delle `RE`). È dato per noto. Talvolta è chiamato anche "*wild character*" Una piccola nota destinata a chi è abituato ad usare Windows. Il carattere `%` (percento), che in quel sistema operativo rappresenta 1 carattere qualsiasi, non esiste nel mondo delle `RE`. Il carattere qualsiasi è rappresentato dal meta-carattere `.` (punto).

Il `?` (punto di domanda) serve per controllare la presenza o meno di un elemento. Supponiamo di voler descrivere, con la sintassi propria delle `RE`, i numeri da 1 a 99. I primi 9 numeri si descrivono facilmente:

```
[1-9]
```

I numeri da 10 a 99 si descrivono tramite:

```
[1-9][0-9]
```

Osservando bene le due `RE` si osserva che `[1-9]` è presente in entrambe, mentre `[0-9]` è presente solo nella seconda. Ovvero può apparire 1 sola volta oppure non apparire affatto (= zero volte). Allora le due `RE` possono essere fuse in un'unica:

```
[1-9][0-9]?
```

Anche in questo caso si tenga a mente che il fattore di reiterazione (`?`) si applica al solo elemento precedente, che è `[0-9]`.

Un ulteriore cenno riguarda l'*alternanza*. Il modello:

```
alfa|beta
```

serve per ricercare la stringa `alfa` oppure, in alternativa, la stringa `beta`, senza l'obbligo che le stringhe abbiano la stessa lunghezza. L'alternanza viene realizzata ricorrendo al metacarattere `|` (barra verticale). Ad esempio, si abbia un testo scritto in inglese che fa riferimento agli autocarri. Non è dato di sapere preventivamente se il testo è scritto in inglese oppure in americano. Quindi, per cercare la parola autocarro bisogna indicare sia la stringa "van" (GB) che "truck" (USA). In tal caso l'alternanza fa proprio al caso nostro: `van|truck`

Le `RE`, per definizione, restituiscono il *massimo* numero di caratteri conforme al modello proposto. Questo comportamento è definito "*greedy*" (= ghiotto). Per rendersene conto consideriamo il seguente input:

```
catamarano
```

ed il seguente modello:

```
a.*a
```

In pratica si ricerca la stringa di caratteri che inizia e finisce col carattere `a`. Come risultato si ottiene:

catamarano

Per ottenere la stringa minima (comportamento "*lazy*" = [pigro]) si opera il seguente ragionamento, da cui poi far discendere il modello. Si deve ricercare la stringa che inizia col carattere voluto (in questo caso `a)` seguito da un qualsivoglia numero di caratteri purché diversi da `a`. La stringa si conclude quindi col carattere `a`. Il modello è:

```
a[^a]*a
```

L'entità `[^a]` rappresenta 1 qualsiasi carattere "diverso da a" cui viene applicato la ripetizione `*` (un qualsivoglia numero). Se l'input è:

```
catamarano
Andromeda
abbandonare
cassapanca
```

col modello proposto

```
a[^a]*a
```

verrà restituito:

catamarano
abbandonare
cassapanca

Ma al riguardo i RE engine più avanzati, ad es. PERL ma non solo, offrono una soluzione più concisa. Consiste nel piazzare il carattere :?: dopo il moltiplicatore. Quindi per ricercare la stringa *minima* che inizia e termina col carattere a nella stringa catamarano, la RE è a.*?a. Se si preferisce il PERL è possibile effettuare delle prove on line usando uno dei siti che offrono il servizio. La tabella 4 a pag. 11 mette in relazione i comportamenti greedy, lazy ed il capturing.

Accanto alle àncore accennate, esistono pure delle altre, a grana più fine. Permettono la ricerca a livello di "*parola*[1]". Quasi tutti i RE engine adottano \b per indicare sia l'inizio che la fine parola (si parla più propriamente di *confine* di parola). Tcl usa \m. Le estensioni GNU, utilizzate dal comando Unix grep, ammettono, accanto a \b, anche \< (inizio) e \> (fine). Ed ancora, chi ammette \b prevede anche \B che sta a significare il contrario di \b, ossia il fatto di essere all'interno di una parola. Concetto non presente nelle GNU extension. Tcl usa \M.

Pertanto la seguente RE:

```
\bb.*m\>
```

individua l'inizio di una parola (è stata usata la forma \b). La parola inizia col carattere b seguita da un numero qualsiasi di caratteri (.*) seguiti dal carattere m che è pure l'ultimo carattere della parola in quanto seguito dal fine parola (è stata usata la forma \> perché si presuppone l'uso di grep). Se l'input è del tipo:

```
Bim, bum, bam!
```

si ottiene il seguente output (notare con attenzione i colori):

Bim, bum, bam!

[1] per *parola* si intende una successione di lettere, sia maiuscole che minuscole, cifre ed il carattere _ (underscore)

1.3 Il *capturing*

Sinora i caratteri intercettati dalle varie RE sono stati stampati. Ma i RE engines, ovvero il software che le maneggia, permettono la memorizzazione del materiale intercettato e quindi il suo utilizzo. Si parla di "*capturing*" e si ricorre ai metacaratteri () (parentesi tonde), già viste nel grouping. Se si vuol catturare qualcosa, la RE deve essere racchiusa da una coppia di parentesi tonde. Il materiale eventualmente intercettato viene memorizzato in buffer interni. Ce ne sono 9 (nove), ciascuno avente come nome una cifra da 1 e 9. Scorrendo da sinistra verso destra una RE che comprende delle () (parentesi tonde), il materiale intercettato dalla prima coppia viene memorizzato nel buffer 1; quello relativo alla seconda coppia nel 2 e si prosegue avanti così. Per utilizzare il contenuto del buffer su usa la notazione \†, ove † rappresenta il numero del buffer voluto.

Un esempio per fissare le idee. Si vogliano ricercare i caratteri doppi (non necessariamente lettere) presenti in un testo. La RE per intercettare un carattere qualsiasi è costituita dal . (punto). Mettendolo tra parentesi tonde, il carattere intercettato finisce nel buffer n° 1. Allora:

```
(.)\1
```

rappresenta la RE che intercetta qualsiasi carattere doppio perché il punto intercetta qualsiasi carattere e lo memorizza. Nel prosieguo della RE viene fatto uso di quanto memorizzato nel buffer. Ovvero il carattere intercettato dal punto, quale che sia, viene ripetuto una seconda volta. Tecnicamente \1 è noto come "*backreference*" in quanto viene fatto riferimento a quanto memorizzato in precedenza.

Se l'input è:

```
ottimizzare
```

allora, con la RE (.)\1 si ottiene

ottimizzare

Il capturing impegna risorse computazionali. Talvolta le parentesi tonde vengono usare per effettuare il raggruppamento ma non si intende catturare i caratteri intercettati. Anche il questo caso i RE engine più avanzati mettono a disposizione un metodo per evitarlo, a tutto beneficio delle prestazioni. Si tratta di piazzare i due caratteri ?: dopo la parentesi aperta. La tabella 4 a pag. 11 mette in relazione i comportamenti greedy, lazy ed il capturing. Le regular expressions ricercano ipotetiche sequenze di numeri pari e dispari, alternati. In grassetto i caratteri intercettati ed in rosso quelli memorizzati nel buffer interno 1.

input	RE	cattura	risultato
0123456789	([02468][13579]){2,4}	yes; greedy	**01234567**89
0123456789	([02468][13579]){2,4}?	yes; lazy	**0123**456789
0123456789	(?:[02468][13579]){2,4}	no; greedy	**0123456**89
0123456789	(?:[02468][13579]){2,4}?	no; lazy	**0123**456789

Tabella 4: Interrelazioni tra comportamenti e capturing

1.4 Variazioni sul tema

Con le RE si possono fare tante belle cose. Ad esempio individuare le quantità numeriche in un testo. A prima vista sembrerebbe impossibile eseguire un controllo dei numeri, perché mancano del tutto costrutti del tipo IF (presuppone la conoscenza della programmazione strutturata). Ma a ben vedere, con un po' di pazienza, si può realizzare un controllo. Non sarà il massimo dell'eleganza né della performance. Ma in macanza di meglio ...

Supponiamo di voler intercettare un numero, con la condizione che il valore deve essere compreso tra 2 limiti. Il problema si risolve descrivendo in termini di RE tutti i numeri compresi tra i 2 limiti. Vediamo con un esempio pratico. Il limite inferiore sia 87 e quello superiore 315. Si procede in questo modo. I numeri da descrivere vengono raggruppati opportunamente, in modo che sia facile ottenere la relativa RE. Nel caso specifico si procede come indicato nella tabella 5 a pag. 11.

87 ÷ 89	90 ÷ 99	100 ÷ 299	300 ÷ 309	310 ÷ 315
8[7-9]	9[0-9]	[12][0-9]{2}	30[0-9]	31[0-5]

Tabella 5: IF tramite regular expression

Il passo successivo consiste nel legare assieme tutte le varie RE tramite l'operatore di alternanza |. Pertanto, per individuare in un testo un numero compreso tra 87 e 315, estremi inclusi, la RE è:

```
8[7-9]|9[0-9]|[12][0-9]{2}|30[0-9]|31[0-5]
```

A noi il compito di pensarla e scriverla. Al RE engine il compito di interpretarla ed applicarla.

Tra i comandi Unix un posto d'onore spetta all'editor vi. Tra le varie funzioni che ci si aspetta da un editor c'è quella di operare delle variazioni.

`vi` prima di operare una qualsiasi modifica, salva la situazione. Se per caso ci si pente, si può fare retromarcia col comando `u` (= undo) (cfr. la freccia a sinistra degli strumenti di Office). `u`, essendo un comando di modifica, prima di operare, salva la situazione. Se usato di nuovo, ripristina la modifica. Ovvero si comporta come un *toggle*, secondo lo schema

situazione corrente (1)
modifica
nuova situazione (2)
`u` - si ritorna alla situazione 1
`u` - si ritorna alla situazione 2
`u` - si ritorna alla situazione 1
etc.

Il comando `vi` per operare una modifica è:

```
:[range]s/old/new/[qualifier]
```

ove `:` serve per entrare nel cosiddetto "*ex mode*" che permette l'uso di comandi complessi. `range` indica la porzione di file su cui agire; in particolare, col carattere `%` si indica tutto il file. `s` (= substitute) è il comando per la modifica. `old` è la stringa da sostituire con `new`. Le stringhe sono delimitate dal carattere `/`, scelto a piacimento, purché non sia presente nelle stringhe. Tra i qualificatori molto importante è `g` (= global). Se usato vengono sostituite tutte le occorrenze. Se non usato, per ogni riga viene sostituita solo la prima occorrenza.

`vi` possiede al suo interno un `RE` engine che permette la backreference. Nulla vieta quindi ad usare la sintassi delle `RE` per indicare la vecchia e la nuova stringa. Se un file è costituito da righe formate da due campi (ad esempio cognome e nome) separati da un apposito carattere è possibile scambiarli di posto con l'uso del backreferencing. Il comando che opera l'apparente *magia* è:

```
:%s/\([a-Z]\+\)(.)\([a-Z]\+\)/\3\2\1/
```

Le parentesi tonde, quelle che operano il capturing, devono essere precedute dal carattere escape (\) e così pure il carattere di ripetizione (+). La prima coppia di parentesi cattura il cognome mettendolo nel buffer 1. La seconda intercetta il carattere di separazione mettendolo nel secondo buffer. Infine l'ultima coppia cattura il nome e lo mette nel buffer 3. La sostituzione fa ricorso alla backreference, scrivendo prima il nome (buffer 3), quindi il separatore (buffer 2) e poi il cognome (buffer 1). Poiché siamo nell'editor,

è sempre possibile fare retromarcia, col comando `u`. Usandolo come è stato spiegato è possibile passare dalla forma originale (prima il cognome e poi il nome) alla nuova (prima il nome e poi il cognome) e viceversa.

forma prolissa	`:%s/foo/  foo  /g`
forma sintetica	`:%s/foo/  &  /g`

Tabella 6: `vi` e `RE`

Sempre nell'editor `vi` le `RE` possono semplificare a volte il lavoro. Se si deve sostituire una stringa con un'altra che prevede la ripetizione della prima, anziché ribattere i caratteri si può usare il carattere `&` (noto in inglese come *ampersand*). Un esempio, purtroppo scolastico, per chiarire le idee. In un testo si vuole sostituire la stringa `<foo>` (parentesi angolari escluse) con `<␣␣foo␣␣>`. In altre parole la stringa `foo` deve essere preceduta e seguita da un doppio spazio. Le due alternative sono evidenziate nella tabella 6 a pag. 13.