Introduzione alle

Espressioni regolari

Mariano Spadaccini spadacciniweb at gmail.com

Versione 1.5 - 7 giugno 2020

Copyright

Indice

Copyright **2**

Indice **3**

1 Premessa **4**

2 La sintassi e il lessico dei linguaggi di programmazione **4**

3 Espressioni regolari **5**
3.1 Regexp parte I . . . 6
3.1.1 Matching repetitions . . . 9
3.1.2 Principi generali . . . 10
3.1.3 Greedy or not greedy? . . . 10
3.1.4 Il carattere *boundary* . . . 11
3.1.5 Esempi riassuntivi . . . 12
3.2 Regexp parte II . . . 12
3.2.1 Grouping e backreferences . . . 12
3.2.2 I modificatori . . . 13
3.2.3 POSIX . . . 18
3.2.4 Unicode . . . 20
3.3 Regexp parte III - Power tools . . . 20
3.3.1 Embedded modifiers . . . 20
3.3.2 Non-capturing groupings . . . 21
3.3.3 Look-ahead e look-behind . . . 21
3.4 Regexp parte IV - Last but not least . . . 23
3.4.1 Non pensate di sapere a sufficienza, per esempi . . . 23
3.4.2 Un esempio avanzato . . . 24
3.4.3 PCRE . . . 25
3.4.4 `Regexp::Common` . . . 26
3.4.5 kregexpeditor . . . 27

4 Esempi **29**
4.1 Esempi con VIM . . . 29
4.2 Esempi con PHP . . . 30
4.3 Esempi con MySQL . . . 31

Riferimenti bibliografici **33**

1 Premessa

Osserviamo questi gruppi di parole:

`I gruppo`	`II gruppo`	`III gruppo`	`IV gruppo`	`V gruppo`
bicchiere	casa	ab123cd	as 12	abcde
perdere	lasagne	cz32de	b2 a	cde
camminare	Fasano	s2c	cza ee	abc

Ognuno dei gruppi[1] ha qualcosa che li distingue dagli altri. Ad esempio:

- nel `I gruppo` sono presenti termini che hanno in comune le ultime due lettere;
- nel `II gruppo` sono presenti termini che hanno in comune la sottostringa `asa` (inoltre nella medesima posizione, cioè seconda, terza e quarta lettera);
- il `III gruppo` presenta il carattere `2`;
- il `IV gruppo` presenta il carattere `" "` (cioè il carattere *spazio*);
- il `V gruppo` presenta il carattere `c` in tutte le stringhe.

Queste sono solo alcune caratteristiche che distinguono i gruppi tra loro; altre meno evidenti possono essere:

- nel `I gruppo` ci sono almeno sette lettere;
- il `III gruppo` presenta la successione lettere/cifre/lettere.

Le caratteristiche distintive di ogni gruppo sono state sin qui scritte *a parole*, ma nel linguaggio informatico è stato necessario stabilire un criterio razionale e formale a cui è stato assegnato il nome *espressioni regolari*.

2 La sintassi e il lessico dei linguaggi di programmazione

Nella sezione precedente è stato accennato un valido motivo per l'introduzione delle espressioni regolari. Un ulteriore motivo riguarda più da vicino

[1] sarebbe più corretto scrivere *gli elementi del gruppo*; spero che il lettore mi perdoni l'ellissi

tutti i linguaggi di programmazione: i programmatori, nel loro lavoro creativo, devono rispettare sia il lessico sia la sintassi del linguaggio che adottano; nel caso non li rispettassero, il loro lavoro non sarà compilato (o non sarà interpretato) poiché le linee di codice non sarebbero comprensibili.
Più formalmente, gli elementi lessicali di un linguaggio di programmazione (costanti, identificatori, parole chiave, ...) sono parole riconosciute se rispettano il lessico; analogamente, i costrutti utilizzati sono riconosciuti se la sintassi è rispettata. Quando un compilatore (o interprete) cerca di riconoscere se una certa stringa utilizzata appartiene al lessico o, analogamente, se il costrutto utilizzato è sintatticamente corretto, il programma confronterà la stringa o il costrutto con un *pattern*.

3 Espressioni regolari

Come è intuibile dalle sezioni precedenti, un'*espressione regolare* è un'espressione costruita secondo una sintassi predefinita che permette di descrivere un insieme di stringhe. Nonostante il *linguaggio delle espressioni regolari* sia formalmente indipendente dai linguaggi di programmazione, questi ultimi permettono la valutazione delle espressioni regolari implementando un proprio motore: la *potenza* del motore è rappresentata (in prima approssimazione) attraverso l'espressività concessa per costruire le espressioni regolari. Tanti *linguaggi* e *metalinguaggi* possiedono un proprio motore, ed il migliore è quello del Perl[2].
Da tale premessa segue che:

- gli esempi mostrati di seguito saranno legati al linguaggio di programmazione scelto; tale scelta ricadrà nel Perl, escluso avviso contrario[3];
- per motivi didattici, la trattazione delle espressioni regolari sarà divisa in due parti:
  1. la prima (sezione 3.1), più generale, è trasversale a (quasi) tutti i motori;
  2. la seconda (sezione 3.2) riguarda concetti avanzati, ed è solo accennata in quanto una trattazione esaustiva:

[2] se il lettore non fosse in accordo con tale affermazione, lo invito a leggere questo documento, poi a riflettere, ed infine (se ancora non fosse soddisfatto) a scrivermi una e-mail

[3] vedi esempi nell'Appendice

- non matcherebbe[4] con il titolo del documento;
- sarebbe impossibile generalizzarla in quanto i concetti presentati sono applicabili solo con i motori più potenti e, in ogni caso, sono specifici per ognuno di essi (rimando alla documentazione ufficiale).

Introduciamo alcuni termini chiave utilizzati nel seguito:

regexp: contrazione di *regular expression*, sarà utilizzato indistintamente dall'equivalente italiano espressione regolare.

pattern matching: si intende l'operazione tramite la quale si verifica se una stringa (o sottostringa) corrisponde al pattern definito, cioè se tale stringa è costruita con uno schema determinato.

pattern substitution: si intende l'operazione tramite la quale si cerca una sottostringa corrispondente al pattern definito (effettuando quindi il *pattern matching*) e, se la ricerca ha esito positivo, si sostituisce a questa sottostringa una stringa da noi definita.

3.1 Regexp parte I

Nella Tabella 1 sono riportati gli elementi atomici tramite i quali è possibile costruire semplici espressioni regolari.
Generalmente, un'espressione regolare si compone di più termini della Tabella 1, come riportato negli esempi che seguono.

Esempio I

`/..ca/`	verifica nella riga la presenza di due caratteri qualsiasi seguiti dalla stringa `ca`

Esempio II

`/\d\d:\d\d:\d\d/`	verifica nella riga la presenza del formato temporale `hh:mm:ss`

Esempio III

`/[0-9a-fA-F]/`	verifica nella riga la presenza di una cifra esadecimale

[4] il verbo *to match* sarà molto utilizzato nel prosieguo del documento, pertanto ho preferito italianizzarlo con *matchare*; ho anche utilizzato i termini *matcherà*, *matcheranno*, *matcherebbe*, ..., sempre per definire il concetto di *matching*; spero vivamente che questa scelta non pregiudichi (per i lettori) la qualità del documento

`peppe`	la stringa `peppe`
`[abc]`	il carattere `a` o il carattere `b` o il carattere `c` (equivale a `[a-c]`)
`[a-z]`	qualsiasi carattere alfabetico minuscolo
`[^0-9]`	qualsiasi carattere escluse le cifre
`\d`	una cifra qualsiasi (equivale a `[0-9]`)
`\D`	un carattere che non sia una cifra (equivale a `[^0-9]`)
`\w`	un carattere alfanumerico (equivale a `[a-zA-Z0-9_]`)
`\W`	un carattere non alfanumerico (equivale a `[^a-zA-Z0-9_]`)
`.`	qualsiasi carattere escluso il *newline* (`\n`)
`\s`	un carattere di spaziatura (equivale a `[\ \t\r\n\f]`)
`\S`	un carattere non di spaziatura (equivale a `[^\s]`)
`\n \r`	rispettivamente i caratteri *new line* e *return*
`\t \v`	rispettivamente i caratteri *tabulazione* e *tabulazione verticale*
`aa|bb|cc`	la stringa `aa` oppure la stringa `bb` oppure la stringa `cc`
`^regexp`	la stringa inizia con l'espressione regolare `regexp`
`regexp$`	la stringa termina con l'espressione regolare `regexp` oppure `regexp\n`
`(regexp)`	oltre a raggruppare l'espressione regolare `regexp`, il motore bufferizza `regexp` (utile con `$`n e `\`n con n numero)
`$`n `\`n	sottostringa dell'espressione regolare esaminata nel gruppo n-esimo (`\`n utilizzato nella definizione della regexp, `$`n successivamente)
`\. \^`	i caratteri `.` e `^`
`\\| \\`	i caratteri `|` e `\`
`\* \+`	i caratteri `*` e `+`
`\[ \]`	i caratteri `[` e `]`
`\( \) \0`	i caratteri `(`, `)` e *null* (il carattere nullo)

Tabella 1: Termini per la composizione di semplici espressioni regolari

Esempio IV

`/[\\abc]de/` — verifica la presenza nella riga di `\de`, `ade`, `bde` oppure `cde`

Esempio V

`/^item[0-9]/` — verifica la presenza di `item0`, ..., oppure `item9`, sottostringa collocata all'inizio della stringa analizzata

Esempio VI

`/^$/`

non è presente nessun carattere tra l'inizio o la fine della riga (l'eventuale carattere `\n` è ignorato[a])

[a] ricordo che `$` matcha alla fine della stringa, o prima del carattere *newline*. Negli esempi successivi, nell'utilizzare quest'ancora il carattere *newline* sarà ignorato.

Esempio VII

`/^\d$/`

tra l'inizio e la fine della riga è presente una cifra

Esempio VIII

`/^.$/`

tra l'inizio e la fine della riga è presente un carattere escluso `\n`

Esempio IX

`/^\d+$/`

tra l'inizio e la fine della riga sono presenti solo cifre, comunque ne è presente almeno una

Esempio X

`/(a|b)b$/`

la riga termina con `ab` oppure `bb`

Esempio XI

`/\.[^\.]*$/`

la riga termina con il carattere `.` ed una eventuale sottostringa che non né contiene

Esempio XII

`/(\d\d):(\d\d):(\d\d)/`

verifica nella riga la presenza del formato temporale `hh:mm:ss`; sarà poi possibile riferirsi all'ora con `$1`, ai minuti con `$2` ed ai secondi con `$3`

Esempio XIII

`/(\w\w)\s\1/`

verifica nella riga la presenza di due caratteri seguiti da `\s` (*spazio*, *newline*, ...), seguito dai medesimi due caratteri che precedono; sarà poi possibile riferirsi agli stessi due caratteri con `$1`

Esempio XIV

`/(19|20)\d\d/` — verifica nella riga la presenza dell'anno compreso tra il 1900 e il 2099; sarà poi possibile riferirsi alle due cifre iniziali con `$1`

3.1.1 Matching repetitions

Nelle espressioni regolari si introduce il concetto di *ripetizione* attraverso le regole sintetizzate nella Tabella 2.

`x?`	il carattere `x` ripetuto 0 o 1 volta
`x*`	il carattere `x` ripetuto 0 o più volte
`x+`	il carattere `x` ripetuto 1 o più volte
`x{`n`}`	il carattere `x` ripetuto n-volte
`x{`n`,}`	il carattere `x` ripetuto almeno n-volte
`x{`m`,`n`}`	il carattere `x` ripetuto da m a n-volte

Tabella 2: Termini per la valutazione della *ripetizione* in una espressione regolare

Esempio XV

`/[a-z]+\s+\d*/` — verifica nella riga la presenza di una sequenza di caratteri alfabetici minuscoli (costituita da almeno un elemento), uno o più caratteri spaziatori ed una eventuale sequenza di caratteri numerici

Esempio XVI

`/\d{2}|\d{4}/` — verifica nella riga la presenza dell'anno composto da due o quattro cifre

Esempio XVII

`/\d{2}(\d{2})?/` — come nell'esempio precedente, verifica nella riga la presenza dell'anno composto da due o quattro cifre; comunque, sarà poi possibile riferirsi all'eventuale seconda coppia con `$1`

3.1.2 Principi generali

Le regole e gli esempi precedenti introducono nel mondo delle espressioni regolari, realtà nella quale è necessario tenere a mente alcuni semplici principi per evitare errori grossolani.

Principio 0. Una espressione regolare matcherà nella stringa il prima possibile

Principio 1. Nel valutare le alternative (ad esempio `a|b|c`), saranno valutate da sinistra a destra (nell'esempio prima `a`, poi `b` e poi `c`) e sarà utilizzata la prima che matcherà.

Principio 2. I quantificatori `?`, `*`, `+` e `{n,m}` matcheranno con la più grande sottostringa compatibile con l'espressione data.

Di seguito saranno riportati alcuni esempi che mostreranno i principi appena citati; da notare che la struttura generale degli esempi sarà la seguente:

- a sinistra in alto la stringa;

- a sinistra in basso il pattern;

- a destra le sottostringhe che matchano oppure una breve spiegazione.

Esempio XVIII

`"Pasticcio"`	`$1 = 'Pasticc'`
`/^(.+)(c|i)(.*)$/`	`$2 = 'i'`
	`$3 = 'o'`

Esempio XIX

`"Pasticcio"`	`$1 = 'Pasticci'`
`/^(.+)(c|i|o)(.*)$/`	`$2 = 'o'`
	`$3 = ''`

3.1.3 Greedy or not greedy?

Nelle formulazioni di regole è spesso prevista l'esistenza di una o più eccezioni; questa sezione rappresenta l'eccezione al **Principio 2** mostrato precedentemente.
In particolare lo stesso principio evidenzia la caratteristica della *voracità* dei motori: in presenza dei *quantificatori* sarà matchato la più grande sottostringa compatibile con l'espressione regolare (comportamento **greedy**).
Per evitare la voracità, è possibile eseguire un match di tipo **non-greedy** aggiungendo il carattere `?`, come mostrato nella seguente tabella.

`*?`	matcherà 0 o più volte
`+?`	matcherà 1 o più volte
`??`	matcherà 0 o 1 volta
`{n}?`	matcherà esattamente n-volte
`{n,}?`	matcherà almeno n-volte
`{n,m}?`	matcherà almeno n-volte ma non più di m-volte

Tabella 3: Comportamento **non-greedy** dei quantificatori. A differenza del comportamento vorace, quelli mostrati preferiranno la più corta sottostringa compatibile con il pattern

Esempio XX

```
"Pasticcio"
/^(.+?)(c|i)(.*?)$/
```

```
$1 = 'Past'
$2 = 'i'
$3 = 'ccio'
```

Esempio XXI

```
"Pasticcio"
/^(.+?)(c|i|o)(.+?)$/
```

```
$1 = 'Past'
$2 = 'i'
$3 = 'c'
```

3.1.4 Il carattere *boundary*

Il **boundary** (spesso chiamato *word boundary*) è un *carattere di larghezza 0* (talvolta definito *non carattere*) ed è presente tra due caratteri di classe eterogenea come, ad esempio, tra due caratteri in cui il primo è di tipo `\w` (alfanumerico) ed il secondo è di tipo `\W` (non alfanumerico). Nelle espressioni regolari il carattere boundary si esprime con la stringa `\b`; di contro, c'è la stringa `\B`, che equivale alla negazione del carattere boundary.

Esempio XXII

```
"abcdabc!a"
/abc\b/
```

match il secondo gruppo `abc`, non il primo

Esempio XXIII

```
"abcdabc!a"
/abc\B/
```

match il primo gruppo `abc`, non sarebbe matchato il secondo

3.1.5 Esempi riassuntivi

Nel Perl si attiva il motore delle espressioni regolari tramite l'operatore `=~` (oppure `!~` che è la negazione del precedente). Di seguito alcuni esempi.

Esempio XXIV
`"stringa" =~ /regexp/` Esegue il matching di `regexp` nella stringa `stringa`; ad esempio:

`"peppe" =~ /^pe.*e$/` restituisce il valore *True*.

Esempio XXV
`$stringa =~ s/pattern/replacement/` Sostituisce l'espressione regolare `pattern` con la stringa `replacement` da noi definita; ad esempio, il seguente codice:

```
-------------------------------------------------
$stringa = "Preferisco programmare in php";
$stringa =~ s/P|php/Perl/;
print $stringa;
-------------------------------------------------
```

sostituisce, nella stringa `$stringa`, `php` con `Perl` e invia il risultato a video mostrando `Preferisco programmare in Perl`.

3.2 Regexp parte II

Come accennato precedentemente, questa sezione è valida solo per i motori di regexp più evoluti; tutti gli esempi saranno riferiti esclusivamente al motore del Perl.

3.2.1 Grouping e backreferences

I raggruppamenti mostrati negli esempi precedenti, sono tutti raggruppamenti *non nidificati*; a volte la **nidificazione** è utilizzata per riferirsi in maniera flessibile ad un dato raggruppamento, come nell'esempio che segue.

Esempio XXVI

```
/(ab(cd|ef)((gh)|ij))/
 1  2      34
```

la cifra presente sotto le parentesi indica il riferimento ad un determinato raggruppamento

Nonostante la nidificazione, non cambia assolutamente nulla per quanto concerne l'ordinamento dei raggruppamenti: il primo gruppo è quello che ha la parentesi più a sinistra, il secondo è quello appena a destra e così via.

Molto utile nella costruzione delle regexp sono le **backreferences**: similmente alle variabili `$1`, `$2`, ..., nelle regexp è possibile utilizzare le *backreferences* `\1`, `\2`, ...: ovviamente, le backreferences possono essere utilizzate solo nella definizione della regexp, come nell'esempio che segue.

Esempio XXVII

`/(\w\w)\1/`	match, ad esempio, `papa`
`/(\w\w)\w\1/`	match, ad esempio, `calca`

3.2.2 I modificatori

I modificatori permettono all'utente di alterare il comportamento del motore delle regexp; nell'ultima parte di questa sezione sarà mostrato un esempio riassuntivo in cui saranno combinati i modificatori presentati.

Case insensitive

Poiché il motore delle espressioni regolari è *case sensitive*[5], può essere utile in alcuni casi forzare il motore ad essere *case insensitive*[6].
Il modificatore che istruisce il motore all'*insensibilità maiuscole/minuscole* è `//i` ed è illustrato nel seguente esempio:

`/[yY][eE][sS]/` è equivalente a `/yes/i`

Splittare e commentare la regexp

A volte è necessario utilizzare espressioni regolari particolarmente complesse e lunghe da non permettere una facile lettura della stessa al programmatore. Fortunatamente è possibile splittare su più linee una regexp ed inserire dei commenti come nel seguente esempio:

[5] cioè sensibile alla differenza tra minuscolo e maiuscolo
[6] cioè non sensibile alla differenza tra minuscolo e maiuscolo

Esempio XXVIII

```
/^
    [+-]?               # match an optional sign
    (                   # match integers or mantissas
        \d+\.\d+        # mantissa of the form a.b
       |\d+\.           # mantissa of the form a.
       |\.\d+           # mantissa of the form .b
       |\d+             # integer of the form a
    )
    ([eE][+-]?\d+)?     # optionally match an exponent
$/x;
```

in cui è evidente che gli spazi ed i commenti inseriti sono ignorati. Per utilizzare il carattere spazio nella regexp è necessario far precedere allo spazio il carattere backslash o definire un carattere di classe tramite `[ ]`; la medesima accortezza dovrà essere attuata per il carattere di commento `#` potendo quindi utilizzare `\#` oppure `[#]`.
Per completezza, anche se poco utilizzata considerando la scarsa utilità, è possibile inserire in un'unica riga sia il commento sia la definizione della regexp attraverso l'utilizzo della sequenza `(?#testo)`, in cui al posto di testo è possibile inserire un commento. Ad esempio:

Esempio XXIX

```
$x = "ab\ncd";                          // inizializzazione
$x =~ /(a(?#commento)\p{IsAlpha})/;     // $1 conterrà ab
```

in cui è evidente il poco significativo commento. L'espressione `\p{IsAlpha}` sarà spiegata nel seguito (sezione 3.2.4).

Single line or multiline

Come evidenziato nella Tabella 1, il carattere `"."` (come elemento delle espressioni regolari) rappresenta qualsiasi carattere escluso il *newline* (`\n`); tale comportamento è conveniente[7] perché (solitamente) nella composizione delle regexp si ignorano i *newline* presenti nel testo. Talvolta però è utile che il motore delle regexp si comporti in maniera differente; quello del Perl permette di personalizzare il comportamento, cioè ignorare o prestare attenzione al *newline* utilizzando rispettivamente i modificatori `//s` oppure `//m`, i quali abbreviano, rispettivamente, *single line* e *multiline*.
In altri termini, tali modificatori determinano se una stringa sarà trattata come una stringa continua oppure come una successione di linee. In conseguenza della loro introduzione, si avrà l'effetto di modificare l'interpretazio-

[7] se non lo doveste considerare conveniente... sappiate che così va il mondo ;-)

ne della regexp. In particolare muterà il comportamento nell'interpretazione della classe di caratteri definita con `.` e il piazzamento delle ancore `^` e `$`. Di seguito riporto i comportamenti alla luce del concetto di modificatore:

senza modificatori: È il comportamento di default:

- `.` matcherà qualsiasi carattere *escluso* il newline;
- `^` ancorato all'inizio della stringa;
- `$` ancorato alla fine della stringa o prima del newline finale.

modificatore s: La stringa è interpretata come una singola linea:

- `.` matcherà qualsiasi carattere, *anche* il newline;
- `^` ancorato all'inizio della stringa;
- `$` ancorato alla fine della stringa o prima del newline finale.

modificatore m: La stringa è interpretata come una successione di linee:

- `.` matcherà qualsiasi carattere *escluso* il newline;
- `^` ancorato all'inizio della linea;
- `$` ancorato alla fine della linea.

entrambi i modificatori: La stringa è interpretata in modo promiscuo:

- `.` matcherà qualsiasi carattere *anche* il newline (come nel caso `//s`);
- `^` ancorato all'inizio della linea (come nel caso `//m`);
- `$` ancorato alla fine della linea. (come nel caso `//m`).

Alcune esempi riepilogativi.
Ricordo che senza modificatore `\n` non appartiene alla classe `.`.
Esempio XXX

`$x = "ab\ncd";`	`// inizializzazione`
`$x =~ /(.*)/;`	`$1` conterrà `ab`
`$x =~ /^(.*)/;`	`$1` conterrà nuovamente `ab`
`$x =~ /(.*)$/;`	`$1` conterrà `cd`
`$x =~ /^(.*)$/;`	non matcherà

Ricordo che con il modificatore `s`, `\n` appartiene alla classe `.`, la linea coincide con la stringa.

Esempio XXXI

`$x = "ab\ncd";`	`// inizializzazione`
`$x =~ /(.*)/s;`	`$1` conterrà `ab\ncd`

Ricordo che con il modificatore `m`, `\n` non appartiene alla classe `.`, le linee sono separate dal *newline*.

Esempio XXXII

`$x = "ab\ncd";`	`// inizializzazione`
`$x =~ /(.cd)/s;`	`$1` conterrà `\ncd`
`$x =~ /^cd/m;`	matcherà la 2^a linea
`$x =~ /^.cd/m;`	non matcherà: la 2^a linea inizia con `cd`

Esempio XXXIII

`$x = "ab\ncd";`	`// inizializzazione`
`$x =~ /^ab.cd$/s;`	matcherà: con `//s` il 3^o carattere è `\n`
`$x =~ /^ab.cd$/m;`	non matcherà: con `//m` il 3^o carattere non esiste
`$x =~ /^ab.cd$/sm;`	matcherà la 2^a linea

Ricordo che con entrambi i modificatori, `\n` appartiene alla classe `.` (come nel modificatore `s`), per l'utilizzo delle ancore si ricorda che per la loro valutazione le linee sono divise dal *newline* (come nel modificatore `m`).

Esempio XXXIV

`$x = "ab\ncd";`	`// inizializzazione`
`$x =~ /^cd$/s;`	non matcherà: il 1^o carattere è `a`
`$x =~ /^cd$/m;`	matcherà la 2^a linea
`$x =~ /^cd$/sm;`	matcherà la 2^a linea
`$x =~ /.cd$/sm;`	matcherà `\ncd`
`$x =~ /^.cd$/sm;`	non matcherà (`^` ad inizio linee)
`$x =~ /^ab.cd$/sm;`	matcherà

Multiple match

Per capire l'utilità del match multiplo, si illustra il seguente esempio:

```
$x = "12 novembre 1001";      // inizializzazione
```

Si voglia ora sostituire la cifra `1` con la cifra `2`; ovviamente, si può generalizzare l'esempio supponendo che la variabile `$x` non sia nota, e si intuirà

facilmente che è necessario utilizzare un nuovo modificatore per effettuare un *match globale*.

Esempio XXXV

```
$x = "12 novembre 1001";      // inizializzazione
$x =~ s/1/2/g;                // sostituite le cifre 1
                              // con 2
```

Con l'utilizzo di questo modificatore, è possibile utilizzare l'ancora `\G` la quale si posiziona dal punto in cui era il precedente match globale. Un esempio sarà presentato nella sezione 3.4.2.

Altri modificatori

In aggiunta ai modificatori precedentemente menzionati, in questa sezione ne presento altri che ho raramente utilizzato o visto utilizzare, ma saranno riportati per la loro semplicità.

- `//o`, utile nei cicli nel caso in cui il pattern non cambia: poiché il motore rivaluta il pattern ogni volta che lo incontra, se nel ciclo il pattern non cambia è più efficiente istruire il motore a non ripetere la valutazione della regexp. Per chiarezza, si veda l'esempio.

  ```
  ------------------------------------------------
  open FH, '</usr/share/dict/italian';
  $pattern = 'zi[ao]';
  while (<FH>) {
      print if /$pattern/
  }
  ------------------------------------------------
  ```

  valuta se nel dizionario ci sono parole che contengono le stringe `zio` o `zia` e, nel caso positivo, le stampa a video.
  È utile sostituire la linea con la valutazione della regexp con la seguente:

  ```
  ------------------------------------------------
      print if /$pattern/o;
  ------------------------------------------------
  ```

  con cui si otterrà un vantaggio prestazionale: ad esempio, sulla mia macchina, eseguendo gli script ottengo:

  ```
  ------------------------------------------------
  > time prova_senza_o.pl > /dev/null
  real    0m0.362s
  user    0m0.336s
  ```

```
sys     0m0.004s
> time prova_con_o.pl > /dev/null
real    0m0.326s
user    0m0.284s
sys     0m0.016s
-------------------------------------------------
```

Il valore assoluto è solo indicativo per ovvi motivi di fluttuazioni, ma ripetendo l'esperimento la differenza è spesso evidente.

- `//e`, utile nella valutazione di codice Perl nella regexp. Si veda l'esempio seguente e la sezione 3.4.1, al paragrafo *A bit of magic: executing Perl code in a regular expression.*

```
-------------------------------------------------
$x = 'Il gatto sul tetto';
$x =~  s/(.)/$chars{$1}++;$1/eg;
print "frequenza di '$_' is $chars{$_}\n"
    foreach (sort {$chars{$b} <=> $chars{$a}}
                              keys %chars);
-------------------------------------------------
```

Nella seconda linea si è inserito un istruzione Perl (`$chars{$1}++`) la quale memorizza in un hash la frequenza dei caratteri; le linee successive stampando a video l'hash ordinando la stampa in base al valore della frequenza.

3.2.3 POSIX

Per la composizione delle espressioni regolari può essere utilizzato lo stile **POSIX**; in particolare la generica sintassi POSIX per le classi di carattere è

`[:class:]`

in cui `class` è un elemento della Tabella 4 della colonna *POSIX*.
È utile evidenziare l'utilizzo delle parentesi quadre le quali sono parti del costrutto; ad esempio:

`[0-9[:alpha:]]` equivale a `[0-9a-zA-Z]`

È altresì utile sottolineare che, l'utilizzo della parentesi quadre più esterne è necessario per il motore del Perl in quanto (quest'ultimo) prevede l'utilizzo dei caratteri di classe POSIX solo all'interno di una classe di caratteri: se la classe non è definita, il Perl segnala un messaggio di *warning*.

POSIX	*Unicode*	*tradizionale*
`alpha`	`IsAlpha`	`[a-zA-Z]`
`alnum`	`IsAlnum`	`[a-zA-Z0-9]`
`ascii`	`IsASCII`	v. di seguito
`blank`	`IsSpace`	`[ \t]`
`cntrl`	`IsCntrl`	v. di seguito
`digit`	`IsDigit`	`\d`
`graph`	`IsGraph`	v. di seguito
`lower`	`IsLower`	`[a-z]`
`print`	`IsPrint`	v. di seguito
`punct`	`IsPunct`	segno di interpunzione
`space`	`IsSpace`	`[\s\v]`
`upper`	`IsUpper`	`[A-Z0-9]`
`word`	`IsWord`	`\w`
`xdigit`	`IsXDigit`	`[a-eA-E0-9]`

Tabella 4: Classi di carattere *POSIX* e *Unicode*

La maggior parte delle classi definite sono di immediata intuizione, le altre sono di seguito commentate:

ascii: Qualsiasi carattere ASCII.

cntrl: Qualsiasi carattere di controllo; solitamente tali caratteri non producono output ma inviano segnali non visualizzati al terminale: ad esempio, *newline* e *backspace* sono caratteri di controllo; altro esempio, tutti i caratteri che corrispondono nella tabella ASCII ad un valore inferiore al 32 sono classificati come caratteri di controllo.

graph: Unione della classe `alpha`, `digit` e `punct`:

`[[:graph:]]` è equivalente a `[[:alpha:][:digit:][:punct:]]`

print: Unione delle classe `graph` e `space`:

`[[:print:]]` è equivalente a `[[:graph:][:space:]]`

È possibile riferirsi alla negazione delle classi precedenti preponendo il carattere `^` al nome della classe; ad esempio:

`[[:^digit:]]` è equivalente a `\D`

3.2.4 Unicode

Oltre allo stile *POSIX* è utilizzabile lo stile **Unicode**, attualmente uno standard di codifica di molti programmi; la generica sintassi Unicode per le classi di carattere è:

```
\p{class}
```

in cui `class` è un elemento della Tabella 4 nella colonna *Unicode*.
Per la definizione delle classi, si rimanda alla precedente sezione.
È possibile riferirsi alla negazione delle classi con la generica sintassi

```
\P{class}
```

Riprendendo un esempio precedente ed estendendolo alla sintassi Unicode:

`[[:^digit:]]` è equivalente a `\D` ed equivalente a `\P{IsDigit}`

3.3 Regexp parte III - Power tools

In questa sezione saranno riportati alcuni strumenti molto potenti, i quali però hanno utilità esclusivamente in ricerche avanzate e, pertanto, poco utilizzati.

3.3.1 Embedded modifiers

Nella sezione 3.2.2 sono stati introdotti ed è stato mostrato l'utilizzo dei modificatori `//i`, `//x`, `//s`, `//m` e `//g`. I primi quattro modificatori possono essere embeddati nella regexp utilizzando rispettivamente `(?i)`, `(?x)`, `(?s)` e `(?m)`. Si veda l'esempio seguente:

`/yes/i` è equivalente a `/(?i)yes/`

Il vantaggio nell'utilizzo dei modificatori embeddati rispetto ai precedenti è che i primi non hanno il vincolo di globalità[8], cioè possono essere definiti per un gruppo e non per l'intero pattern, cosicché la definizione di un gruppo permette di isolare l'effetto di un modificatore, come nel seguente esempio.

Esempio XXXVI

```
$x = "Answer: Yes";          // inizializzazione
$x =~ /Answer: (?i)yes/;     // matchano entrambe; nella
$x =~ /Answer: ((?i)y)es/;   // riga I case insensitive
                             // "yes", nella riga II "y"
```

[8] si intuisce la precedente esclusione del modificatore `//g`

Un secondo vantaggio è la disabilitazione dei modificatori precedentemente definiti preponendo il carattere `-` al carattere identificativo del modificatore (ad esempio `(?-i)`).
Ulteriore vantaggio è la possibilità di combinare in una singola espressione più modificatori; ad esempio `(?s-i)` abilita il *single line mode* e disabilita il *case insensitive*.

3.3.2 Non-capturing groupings

I raggruppamenti fin qui mostrati hanno due distinte funzioni:

1. raggruppare elementi in una singola unità (*grouping*);
2. memorizzare il raggruppamento per un successivo utilizzo (*backreference* con `$1`, `$2`, ...).

Quest'ultima funzione è parte del meccanismo denominato *capturing groupings*, ma se non è necessaria si ricorre al raggruppamento senza memorizzazione, denominato *non-capturing groupings*, utilizzato tramite l'espressione `(?:regexp)`.
Ovviamente le variabili `$1`, `$2`, ..., non saranno utilizzabili perchè non settate, per cui si intuisce che questo raggruppamento sarà più rapido del precedente nell'esecuzione.
Come nell'esempio seguente, è possibile utilizzare contemporaneamente entrambi i meccanismi:

Esempio XXXVII

```
/([+-]?\d+)(?:\.\d+)?/          # memorizza in $1 solo
                                # il segno e la parte intera
```

3.3.3 Look-ahead e look-behind

Nelle regexp è usuale incontrare le *ancore* quali `^`, `$` e `\b` le quali rispettivamente valutano se ci sono caratteri anteriormente (comportamento *look-ahead*), caratteri posteriormente (comportamento *look-behind*) e caratteri di classe eterogenea (comportamento sia *look-ahead* sia *look-behind*).
In ciascuno dei precedenti casi (ed in generale) le ancore permettono espressività nella regexp *senza la cattura* essendo stringhe o caratteri a *larghezza 0*[9].
Il look-ahead è inserito nel pattern della regexp utilizzando `(?=regexp)` ed il look-behind attraverso `(?<=regexp)` come nell'esempio seguente.

[9] si veda sezione 3.1.4

Esempio XXXVIII

```
$x = "A twister of twists";     // inizializzazione
$x .= "once twisted a twist";   // inizializzazione
@stwords = ($x =~ /(?<=twi)st(?=\w+)/g)
                                // matchano twister,
                                // twists e twisted
```

Per rendere più evidente l'utilizzo e l'utilità del look-behind e look-ahead, si mostrerà un esempio analogo al precedente, in cui sarà evidente l'inutilità di una porzione del pattern.

Esempio XXXIX

```
$x = "A twister of twists";     // inizializzazione
$x .= "once twisted a twist";   // inizializzazione
@stwords = ($x =~ /twi(?<=twi)st(?=\w+)/g)
                                // matchano twister,
                                // twists e twisted
```

Nell'esempio appena mostrato, si può notare l'utilizzo del pattern `/twi(?<=twi)st(?=\w+)/`, il quale sintetizza una stringa formata da:

- la sequenza `twi` (catturata);
- la sequenza `st` (catturata) preceduta dalla sequenza `twi` (non catturata);
- una sequenza `\w+` (non catturata).

È pertanto evidente la differenza tra i due esempi precedenti in cui, attraverso l'utilizzo di due pattern differenti, è mostrata l'essenza del look-ahead e look-behind (si ricorda che l'ancora ha sempre larghezza zero).
Purtroppo nell'utilizzo del look-behind esiste un limite, difatti se nel look-ahead si è utilizzato correttamente un'arbitraria lunghezza (nell'esempio `(?=\w+)`), nel look-behind la lunghezza dovrà essere fissata a priori, pertanto si ricorrerà ad uno schema del tipo `(<=fixed-width)`. Ad esempio, potrà essere utilizzato `(?<=(ab|cd)`, ma non potrà essere utilizzato `(?<=\w*)`.
Anche nel look-ahead e nel look-behind è possibile negare un'espressione utilizzando il punto esclamativo al posto del segno di uguaglianza, come rispettivamente nelle sintassi `(?!regexp)` e `(?<!regexp)`, ricordando che quest'ultima regexp dovrà avere lunghezza fissata.

3.4 Regexp parte IV - Last but not least

3.4.1 Non pensate di sapere a sufficienza, per esempi

Per ignoranza dilagante dell'autore, per pigrizia del lettore[10], per la poco utilità e sicuramente per altre ragioni che al momento ignoro, in questo documento si evitano ulteriori approfondimenti[11].
È comunque vero che è utile sapere che c'è dell'altro e, soltanto per citare qualche elemento, ho preferito riportare il titolo in originale di alcune sezioni inserite nella documentazione del Perl (digitando `perldoc perlretut` nel sistema GNU/Linux):

- *Using independent subexpressions to prevent backtracking*, ad esempio:

  Esempio XL
  ```
  $x = "ab";                  // inizializzazione
  $x =~ /a*ab/;               // matcha
  $x =~ /(?>a*)ab/;           // non matcha
  ```

  Il primo confronto con l'espressione regolare ovviamente matcha; il secondo non matcha in quanto la porzione `(?>a*)` è una sottoespressione indipendente: ciò implica che la sottoespressione non si preoccupa (comportamento indipendente) del resto della regexp matchando e, contestualmente, causando il mancato match dell'intera regexp.

- *Conditional expressions*, permettono di inserire un costrutto condizionale all'interno di una regexp, come `(?(condizione)si-pattern)`[12] e `(?(condizione)si-pattern|no-pattern)`[13], in cui se la condizione espressa da `(condizione)` è verificata si esegue `si-pattern`, altrimenti nella seconda tipologia di espressione condizionale si esegue `no-pattern`. Ad esempio `/[ATGC]+(?(?<=AA)G|C)$/` matcha una sequenza di DNA che termina in `AAG` oppure in `C` (nota: formalmente è più corretta la forma `(?((?<=AA))G|C)`, ma in presenza di look-ahead e look-behind il Perl ci permette la forma più leggibile `(?(?<=AA)G|C)`).

[10] difficilmente sarete arrivati a leggere fino a questa sezione
[11] in questo documento eventuali approfondimenti li considererei poco utili perché

- trasformerebbero tale manuale da livello introduttivo/intermedio a livello intermedio/avanzato, e spero che la la maggior parte dei lettori tecnici italiani preferiscono leggere documentazione approfondita in lingua originale (e quindi evitare l'italiano);
- non credo di aver mai utilizzato ulteriori strutture nelle regexp se non letto per cultura personale qualche pagina di documentazione.

[12] corrisponde al costrutto *if-then*
[13] corrisponde al costrutto *if-then-else*

- *A bit of magic: executing Perl code in a regular expression*, per inserire codice Perl in una regexp; ad esempio

Esempio XLI

```
$x = "abcdef";                  // inizializzazione
$x =~ /abc(?{print "Hi!";})def/;
                                // matcha e scrive "Hi!"
```

È utile segnalare che, se nell'esempio precedente non fosse stata inserita la stringa `def` ma, ad esempio, `[a]ef`, il `print` sarebbe ugualmente stato eseguito; rimando alla documentazione ufficiale per approfondire tale questione e la sua banale spiegazione.

- *Pragmas and debugging*, per il debug delle regexp in Perl... e non solo. Un esempio di debug possiamo ottenerlo su una macchina GNU/Linux con installata la distribuzione Perl digitando:

```
perl -e 'use re "debug"; "abc" =~ /a*b+c/;'
```

3.4.2 Un esempio avanzato

In questa sezione presento un problema pratico presentatomi qualche tempo fa: *"Si voglia aggiungere con una regexp il punto di separazione delle migliaia ad un numero"*.
Se l'utente proverà a risolvere questo problema, difficilmente troverà in breve tempo una soluzione; in ogni caso, qualcuno l'ha risolto per noi[14] tramite questa regexp[15]:

```
s/(^[-+]?\d+?(?=(?>(?:\d{3})+)(?!\d))|\G\d{3}(?=\d))/$1,/g;
```

Scritta in questo modo non è molto leggibile, pertanto la arricchiamo di alcuni commenti:

```
s/(
    ^[-+]?          # inizio del numero
    \d+?            # cifre prima della prima virgola
    (?=             # seguite da  (ma non incluso nel match):
       (?>(?:\d{3})+) # gruppo composto da un multiplo di 3 cifre
       (?!\d)       # un multiplo esatto
```

[14] tratto da `perlfaq 5`, *How can I output my numbers with commas added?*

[15] la soluzione proposta inserisce il carattere `,` al posto del `.` di separazione delle migliaia (schema americano invece di quello italiano), ma tecnicamente non c'è alcuna differenza ad esclusione della banale sostituzione del carattere `,` con `.` alla fine della regexp

```
    )
   |                # oppure
    \G\d{3}         # dopo il precedente gruppo, prendi 3 cifre
    (?=\d)          # ma ci sono altre cifre dopo
)/$1,/xg;
```

3.4.3 PCRE

La libreria PCRE (***P**erl **C**ompatible **R**egular **E**xpressions*) è un insieme di funzioni che implementa un motore per espressioni regolari, il quale utilizza la stessa sintassi e semantica del Perl versione 5[16].
Scendendo più nel dettaglio, PCRE fornisce un'*API* che corrisponde alla definizione della *API POSIX* per la definizone delle regexp.
Come riportato nella *home page* di PCRE[17], PCRE è stato sviluppato originariamente per `Exim MTA`, ma ora è utilizzato da molti progetti open source di grande successo, inclusi `Apache`, `PHP`, `KDE`, `Postfix`, `Analog` e `Nmap`.
Nel mio sistema è installato PCRE, pertanto posso accedere alla documentzione digitando semplicemente

```
------------------------------------------------
> man pcre
------------------------------------------------
```

Possiamo lanciare `pcretest`, come nel seguente caso:

```
------------------------------------------------
    >  pcretest
PCRE version 7.8 2008-09-05

re> /^abc(\d+)a/
data> abc123
No match
data> abc123a
 0: abc123a
 1: 123
data> abc123a1
 0: abc123a
```

[16] nel momento in cui sto scrivendo, utilizzo nella mia distribuzione GNU/Linux Perl versione 5.10.0

[17] `http://www.pcre.org`

```
 1: 123
data> abc10a2
 0: abc10a
 1: 10
data>
-------------------------------------------------
```

quindi dai nostri esempi sembra che la libreria sia robusta. ;-)

3.4.4 Regexp::Common

Questo modulo (presente su CPAN[18]) fornisce una interfaccia per comuni richieste di regexp; di seguito un esempio che permette semplicemente di determinare se una parola è palindroma[19]:

```
-------------------------------------------------
#!/usr/bin/perl -w

use Regexp::Common qw /lingua/;

open FH, '</usr/share/dict/italian';
while (<FH>) {
    /^$RE{lingua}{palindrome}$/    and  print $_;
}
-------------------------------------------------
```

il quale produce in output i 23 palindromi contenuti nel vocabolario[20][21] che ho installato per il controllo sintattico.

```
-------------------------------------------------
FSF
aerea
afa
ala
alla
ama
```

[18] http://search.cpan.org/dist/Regexp-Common/lib/Regexp/Common.pm
[19] lessicalmente indifferente da sinistra o da destra
[20] attualmente contiene 116878 termini
[21] per curiosità, quello inglese contiene 98569 termini di cui 56 palindromi

```
ara
atta
avallava
aveva
ebbe
elle
ere
esse
ingegni
inni
ivi
non
onorarono
oro
oso
osso
otto
-------------------------------------------------
```

Analogamente è possibile capire se siamo in presenza di:

numeri $\longrightarrow$ `$RE{num}`

- **reali** $\longrightarrow$ `$RE{num}{real}`
- **interi** $\longrightarrow$ `$RE{num}{int}`
- **binari con virgola** $\longrightarrow$ `$RE{num}{real}{-base => 2}`
- **binari con separatore di migliaia** $\longrightarrow$
  $\longrightarrow$`$RE{num}{real}{-base => 2}{-sep => ','}{-group => 3}`

delimitatori ad es. una sequenza `/.../` $\longrightarrow$ `$RE{delimited}{-delim=>'/'}`

espressione con parentesi bilanciate $\longrightarrow$ `$RE{balanced}{-parens=>'()'}`

Per una ricca documentazione corredata di esempi efficaci, su sistemi GNU/Linux si consulti `perldoc Regexp::Common`.

3.4.5 kregexpeditor

Quest'editor di espressioni regolari trasforma il pattern inserito (formato ASCII) in schema grafico.

Nonostante questo strumento possa essere utile ai neofiti, il motore non sembra molto robusto; infatti, inserendo un pattern un minimo complesso, il comportamento non appare corretto, evidenziando in basso a destra un *simbolo di warning* (si veda Figura 3).

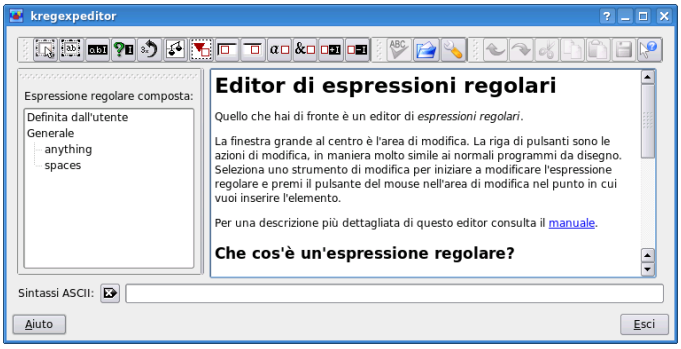

Figura 1: Apertura del programma *kregexpeditor*

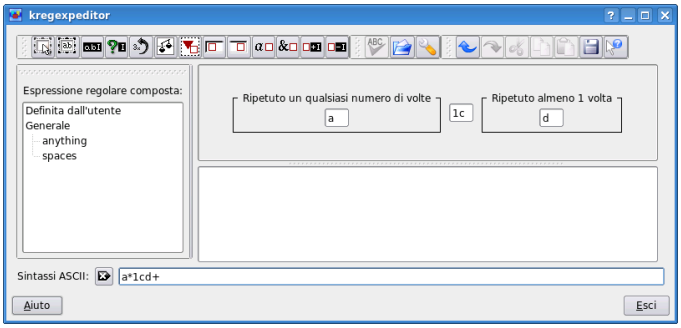

Figura 2: Inserimento del pattern `a*1cd+`

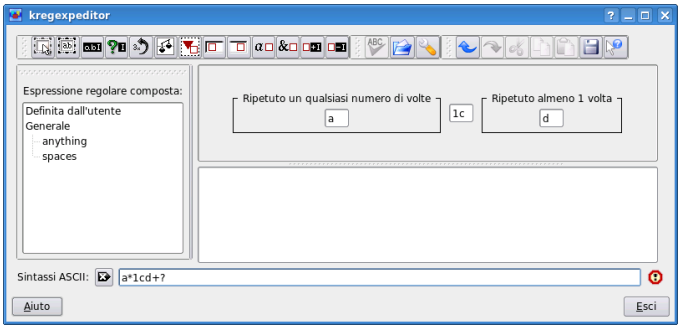

Figura 3: Inserimento del pattern `a*1cd+?`

4 Esempi

4.1 Esempi con VIM

Tramite questo editor testuale, la valutazione delle espressioni regolari avviene in modalità *COMMAND LINE*. Quindi, anche se non esplicitamente ricordato, prima di della digitazione mostrata, è necessario entrare in tale modalità.

Pattern matching

Per eseguire il matching di una espressione regolare, è necessario digitare:

```
:/regexp
```

in cui `regexp` è l'espressione regolare da cercare. Diversamente dal caso precedente, il pattern matching può essere valutato in maniera *case insensitive*:

```
:/regexp/i
```

Pattern substitution

Per eseguire la sostituzione di un stringa, è necessario digitare:

```
:n,ms/pattern/replacement/
```

in cui `pattern` è l'espressione regolare da sostituire, `replacement` è la stringa che sostituirà il pattern, `n` ed `m` sono due numeri interi con `m>n`, in cui `n` indica la riga dalla quale inizia la sostituzione, `m` indica la riga in cui termina la sostituzione.
Analogamente alla sezione precedente, avremmo potuto aggiungere nella definizione dell'espressione regolare la direttiva `i` per effettuare un pattern matching *case insensitive*.
Spesso si esegue una sostituzione *globale* aggiungendo nella definizione dell'espressione regolare la direttiva `g` (*global*):

```
:n,ms/pattern/replacement/g
```

È inutile commentare l'esempio seguente:

```
:n,ms/pattern/replacement/gi
```

facilmente comprensibile combinando i due paragrafi precedenti.

4.2 Esempi con PHP

Pattern matching

`preg_match`: Esegue il matching di un espressione regolare; è utile l'esempio seguente:

```
-------------------------------------------------
$stringa = "Perl e PHP sono due ottimi linguaggi";
$pattern = "/\bPerl\b|\bPHP\b/"
if (preg_match ($pattern, $stringa)) {
   print "Il riconoscimento è avvenuto";
} else {
   print "Testo non riconosciuto";
}
-------------------------------------------------
```

In questo *if-then-else* si controlla che nella stringa `$stringa` siano contenute le sottostringhe `Perl` o `PHP`, precedute e seguite da un boundary; infatti, è eseguito il ramo *then* per la presenza della stringa `Perl` delimitata alle estremità da un carattere boundary.

Diversamente dal caso precedente, il pattern matching può essere valutato in maniera *case insensitive* come nel seguente esempio:

```
-------------------------------------------------
$stringa = "Il PHP è un ottimo linguaggio";
$pattern = "/ php /i"
if (preg_match ($pattern, $stringa)) {
   print "Il riconoscimento è avvenuto"; }
else {
   print "Testo non riconosciuto";
}
-------------------------------------------------
```

che restituirà `Il riconoscimento è avvenuto`.

Pattern substitution

`preg_replace`: Sostituisce una espressione regolare con la stringa da noi definita; ad esempio, il seguente codice:

```
-----------------------------------------------
$stringa = "May 20, 2008";
$pattern = "/\b20/";
$replacement = " 30";
echo preg_replace($pattern, $stringa, $replacement);
-----------------------------------------------
```

mostra a video la stringa `May 30, 2008`.

Analogamente alla sezione precedente, avremmo potuto aggiungere nella definizione dell'espressione regolare la direttiva `i` per effettuare un pattern matching *case insensitive*: in questo caso l'aggiunta sarebbe stata sintatticamente corretta, ma sarebbe anche insignificante[22].
Diversamente dal caso precedente, potremmo eseguire una sostituzione *globale* aggiungendo nella definizione dell'espressione regolare la direttiva `g` (*global*); ad esempio, il seguente codice:

```
-----------------------------------------------
$stringa = "May 20, 2008";
$pattern = "/\b20/g";
$replacement = " 30";
echo preg_replace($pattern, $stringa, $replacement);
-----------------------------------------------
```

mostra a video la stringa `May 30, 3008`.

4.3 Esempi con MySQL

Pattern matching

Di seguito saranno riportati alcuni esempi di pattern matching in cui la risposta `0` oppure `1` evidenzia rispettivamente il non match oppure il match della regexp.

```
-----------------------------------------------
mysql> SELECT 'fo\nfo' REGEXP '^fo$';
0
```

[22] questo perché i caratteri numerici non si distinguono in *maiuscoli* e *minuscoli*

```
mysql> SELECT 'fofo' REGEXP '^fo';
1
mysql> SELECT 'Ban' REGEXP '^Ba*n';
1
mysql> SELECT 'abcde' REGEXP 'a[bcd]{3}e';
1
mysql> SELECT 'aXbc' REGEXP '^[a-dXYZ]+$';
1
mysql> SELECT 'aXbc' REGEXP '^[^a-dXYZ]+$';
0
mysql> SELECT '~' REGEXP '[[.tilde.]]';
1
mysql> SELECT 'justalnums' REGEXP '[[:alnum:]]+';
1
mysql> SELECT '!!' REGEXP '[[:alnum:]]+';
0
------------------------------------------------
```

Il terzultimo esempio evidenzia la possibilità di ricorrere a stringhe al posto di taluni caratteri, mentre gli ultimi due esempi suggeriscono la compatibilità con lo *standard POSIX*.

Riferimenti bibliografici

[1] Per il sistema operativo GNU/Linux rimando alla documentazione fornita nella distribuzione; utilissimi (dal più semplice al più completo/complesso) `perldoc perlrequick`, `perldoc perlre` e `perldoc perlretut`. Ovviamente tale documentazione è disponibile per qualunque distribuzione Perl degna di questo nome ed è quindi accessibile indipendentemente dal sistema operativo che state utilizzando.

[2] Perl Compatible Regular Expressions:
http://www.pcre.org/

[3] `perldoc Regexp::Common` oppure http://search.cpan.org

[4] Definito da se stesso *"The Premier website about Regular Expressions"*:
http://www.regular-expressions.info/

[5] How Regexes work
http://perl.plover.com/Regex/

[6] Weaving Magic With Regular Expressions
http://www.wdvl.com/Authoring/Languages/Perl/Weave/

[7] Moltissimi riferimenti li trovate su:
http://www.google.com