

Apache Web Server

Il primo sito italiano completamente dedicato al Web Server Apache
Piattaforma Win32

Lezioni pratiche sulla programmazione in Perl a cura di **Giordano Costantini**

N° Lezione	Titolo	Data
Lezione 1	Introduzione	11/06/2000
Lezione 2	Avvio alla programmazione	11/06/2000
Lezione 3	Variabili scalari	12/06/2000
Lezione 4	Costruttori e significati	12/06/2000
Lezione 5	Numeri	12/06/2000
Lezione 6	Stringhe	13/06/2000
Lezione 7	Valori indefiniti	14/06/2000
Lezione 8	Operazioni tra Numeri e Stringhe	14/06/2000
Lezione 9	Privilegi degli operatori	14/06/2000
Lezione 10	Interagire con il programma	15/06/2000
Esercizi	Soluzione degli esercizi dati	15/06/2000
Lezione 11	Variabili array	16/06/2000
Lezione 12	Assegnazione dei valori	17/06/2000
Lezione 13	Operazioni degli array	17/06/2000
Lezione 14	Funzioni foreach	19/06/2000
Lezione 15	Variabili hash	20/06/2000
Lezione 16	Istruzioni decisionali (if, else, elsif, unless)	21/06/2000
Lezione 17	Istruzioni decisionali (?, &&,)	21/06/2000
Lezione 18	Istruzioni iterative (while ed until)	22/06/2000
Lezione 19	Istruzioni iterative (do, for, foreach)	22/06/2000
Lezione 20	Espressioni Regolari 'Prima parte'	23/06/2000
Lezione 21	Espressioni Regolari 'Seconda parte'	25/06/2000
Lezione 22	Espressioni Regolari 'Terza parte'	26/06/2000
Lezione 23	Espressioni Regolari 'Ultima parte'	29/06/2000
Lezione 24	Interagire con i file	30/06/2000
Lezione 25	Interagire con i file (ultima parte)	02/07/2000
Lezione 26	DBM File	10/07/2000

Lezioni pratiche sulla programmazione in Perl

Lezione 1

Introduzione

Con questa lezione diamo inizio, al nostro corso introduttivo alla programmazione in Perl. Al termine di questo corso sarete in grado di poter sviluppare da soli i vostri programmini CGI, da implementare nelle vostre pagine Web, ma sarete altresì in condizione di poter scrivere qualsiasi programma anche per la gestione del vostro sistema operativo.

Il Perl è un linguaggio di programmazione semplice e potente, a differenza di altri linguaggi dove tutto è troppo complesso, grazie al Perl risulterà facile realizzare ciò che è semplice e possibile realizzare applicazioni complesse.

Il geniale, ma oltretutto generoso ideatore di questo linguaggio, Larry Wall recitava spesso questa frase: "Il programmatore Perl si riconosce dal sorriso che porta stampato in faccia", questo perché la sintassi del Perl corrisponde con semplicità a quello che la natura umana e il senso comune suggeriscono: è imperativo quando si devono ordinare comandi, orientato agli oggetti quando si devono affrontare strutture articolate in gerarchia e moderatamente criptico quando si devono specificare espressioni regolari.

Perl, non è solo un formidabile linguaggio nato nel lontano 1987 dal genio di Larry Wall, ma è anche l'espressione concreta della condivisione delle conoscenze, all'epoca contava appena una decina di seguaci in tutto il mondo, oggi invece se ne stimano più di un milione, con un incremento stimato sui diecimila novizi al mese.

Se anche tu, desideri addentrarti in questo fantastico mondo, non devi far altro che scaricare dalla rete l'interprete Perl, offerto gratuitamente sia in formato sorgente sia precompilato per i più diffusi sistemi operativi, e tutti possono contribuire ad estenderlo e a perfezionarlo con suggerimenti e/o sviluppi concreti.

Alcuni indirizzi utili:

<http://www.perl.com>

<http://www.perl.org>

<http://www.cpan.org>

Lezioni pratiche sulla programmazione in Perl

Lezione 2

Avvio alla programmazione

In questo corso, almeno per il momento, non ci occuperemo dello sviluppo dell'interprete Perl, ne tanto meno della compilazione dei sorgenti, ma ammetteremo che ogni uno di voi abbia installato sulla propria macchina l'interprete Perl servendosi degli eseguibili scaricabili gratuitamente dal sito ufficiale <http://www.cpan.org/ports/index.html>. Nel nostro caso ci preoccuperemo di prelevare l'eseguibile Perl più aggiornato per sistemi Win32 e le librerie libwin32 più recenti in formato zip <http://www.activestate.com/PPMPackages/5.005/zips>.

Restando in linea con lo sviluppo del sito, questa guida si occuperà principalmente di fornirvi le basi di programmazione per l'ambiente Windows, ma alla fine di questa serie di lezioni vi risulterà facile ed immediato sviluppare del software Perl per qualsiasi piattaforma.

D'ora in avanti, presumeremo che abbiate installato la vostra copia Perl in C:\perl e che il suo interprete si trovi in C:\perl\bin\perl.exe, quindi tutti gli esempi riporteranno il suddetto percorso.

Iniziamo subito in modo molto educato a scrivere il nostro primo script o programma (chiamatelo come volete), aprite il vostro editor di testi, preferibilmente BloccoNote di Windows o edit del Dos, ed iniziate a digitare le seguenti stringhe:

```
#!C:/perl/bin/perl.exe  
print "Buon giorno Perl \n";  
print "il mio nome è Giordano\n";  
print "un domani faremo gradi cose insieme\n";
```

Questo primo esempio di programmazione è molto semplice, come altrettanto semplice è la filosofia del linguaggio, generalmente tutti gli script Perl iniziano con un commento speciale che indichi al sistema dove trovare l'interprete Perl, nel nostro caso C:\perl\bin\perl.exe, questo commento viene considerato speciale, perchè contrassegnato da un punto esclamativo, ed è l'unico commento che il sistema potrà riconoscere, mentre, vengono considerati commenti normali, tutte quelle stringhe contrassegnate dal carattere #, quest'ultime vengono ignorate dal sistema ed hanno il solo scopo di promemoria per il programmatore, l'esempio successivo avrà infatti lo stesso risultato di quello sopra descritto.

```
#!C:/perl/bin/perl.exe  
# questa prima stringa indica il percorso dell'interprete  
print "Buon giorno Perl \n";  
# tramite il comando print ordiniamo al nostro programma  
# di stampare ciò che è contenuto tra i doppi apici  
# i caratteri \n individuano un comando speciale  
# che obbliga il programma ad andare a capo  
# mentre il punto e virgola chiude il comando  
print "il mio nome è Giordano \n";  
# Ovviamente potrete modificare il nome inserendo il vostro  
print "un domani faremo gradi cose insieme. \n";  
# Potrete anche decidere di modificare tutte le scritte  
# ed aggiungerne delle altre.
```

Una volta copiato, o modificato lo script, salvatelo in una nuova cartella che utilizzeremo per tutti gli altri esempi col nome di `saluto.pl`, fate attenzione a selezionare tutti i tipi di file prima di salvare ed ad assicurarvi che il vostro sistema riconosca la nuova estensione `.pl`, altrimenti avrete come risultato un file `saluto.pl.txt`, che non vi serve a nulla, per andare sul sicuro, è meglio utilizzare i doppi apici in questo modo: salva con nome `"saluto.pl"`, in questo modo saremo certi che al nostro file non si aggiungerà nessun'altra estensione estranea.

Ora non ci resta che visualizzare il risultato del nostro primo esempio, spostatevi quindi nella cartella dove avete deciso di salvare il vostro script, ed avviatelo con un doppio click del mouse, se il sistema non ha associato alcun programma a tale estensione, lo potrete fare voi scegliendo `perl.exe` reperibile nella cartella `c:\perl\bin\`.

A questo punto vi si dovrebbe aprire una finestra del Dos con il seguente risultato:

```
Buon giorno Perl  
il mio nome è Giordano  
un domani faremo grandi cose insieme
```

Ma voi non farete sicuramente in tempo a leggerlo, in quanto il programma non farà altro che stampare sul monitor, tramite il comando `print`, le righe da voi impostate, e se non avrete commesso errori si arresterà, chiudendosi automaticamente, se invece è vostra intenzione visualizzare con calma il risultato del vostro script, dovrete aprire una finestra del Dos e spostarvi nella directory dell'eseguibile Perl in questo modo: `cd C:\perl\bin`, cliccate su invio e scrivete il seguente comando:

`perl c:\provePerl\saluto.pl`

dove provePerl è la cartella contenente il mio primo script.

Se tutto è stato fatto correttamente, dovrebbe apparirvi in modo stabile il risultato descritto in precedenza.

Lezioni pratiche sulla programmazione in Perl

Lezione 3

Variabili scalari

Come secondo esempio, cercheremo di estendere il semplice programma `saluto.pl`, sviluppato nella lezione precedente, introducendo il concetto di variabili scalari, inizieremo pertanto con scrivere un nuovo script, forse un po' più articolato del precedente, che di lo stesso risultato, ormai credo sia ovvio che voi sappiate già che dobbiate aprire il vostro editor di testi, quindi non mi ripeterò più.

```
#!C:\perl\bin\perl.exe
# specifichiamo il valore della nostra variabile
# scalare utilizzando un nome a piacere
# preceduto dal simbolo del $, risulta facile da ricordare
# in quanto simile alla S iniziale di scalare
$a = 'buon giorno Perl';
$b = 'il mio nome è giordano';
$c = 'un domani faremo grandi cose insieme';

print "$a \n";
print "$b \n";
print "$c \n";
```

Salvatelo col nome `ciao.pl` ed avviatelo per la verifica, utilizzando il procedimento illustrato nella lezione precedente, se tutto è andato bene e non avete commesso gravi errori di sintassi, il risultato dovrebbe essere molto simile all'esempio precedente.

In questo esempio, abbiamo utilizzato tre variabili scalari contenenti delle stringhe di testo, le variabili sono locazioni di memoria richiamabili per mezzo del loro nome, nel nostro caso `$a`, `$b` e `$c`.

Una delle particolarità del Perl consiste nel distinguere solamente due tipi di variabili, variabili scalari, che possono contenere sia un numero, un carattere o un stringa; e variabili lista contenente, in qualsiasi ordine, un insieme di valori. Senza dare quindi importanza al tipo di contenuto della variabile, il Perl, a differenza di altri linguaggi più caotici, che fanno differenza tra carattere, intero, a virgola mobile stringa ecc. tende sempre più ad assomigliare alla lingua parlata. Per creare una nuova variabile basta utilizzare il carattere `$` seguito da un nome a vostra scelta non più lungo di 256 caratteri, inoltre nel linguaggio Perl i nomi sono tutti case sensitive, distinguendo pertanto le lettere minuscole dalle maiuscole, per esempio la variabile scalare `$A` corrisponde ad una variabile diversa da `$a`, almeno fin quando non specificheremo il contrario mediante un comando d'uguaglianza.

Ora volendo estendere il nostro script potremmo esercitarci a stampare il carattere `$` senza che l'interprete lo tratti come uno scalare, quindi avviate il vostro editor di testi ed aprite il file `ciao.pl`, modificandolo con l'aggiunta delle seguenti stringhe:

```
$denaro = '0.00';
print "è un piacere lavorare con te, specialmente per il fatto che mi sei costato solo $denaro \$ \n";
```

Grazie all'uso del simbolo `\` riusciamo ad ordinare al Perl di non considerare il simbolo `$` come variabile ma solo come semplice carattere, la stessa cosa vale per il carattere speciale `@` che in linguaggio Perl indica un array, se volessimo quindi far stampare il nostro indirizzo dovremmo scrivere `mionome\@dominio.it`.

Lo script finale per la verifica, che salverete col nome di `ciao1.pl` è il seguente:

```
#!C:\perl\bin\perl.exe
# specifichiamo il valore della nostra variabile
# scalare utilizzando un nome a piacere
# preceduto dal simbolo del $, risulta facile da ricordare
# in quanto simile alla S iniziale di scalare
$a = 'buon giorno Perl';
$b = 'il mio nome è giordano';
$c = 'un domani faremo grandi cose insieme';
$denaro = '0.00';

print "$a \n";
print "$b \n";
print "$c \n";
print "è un piacere lavorare con te, specialmente per il fatto che mi sei costato solo $denaro \$ \n";
```

Lezioni pratiche sulla programmazione in Perl

Lezione 4

Costruttori escape e significati

Costrutto	Significato
<code>\n</code>	Newline
<code>\r</code>	Return
<code>\t</code>	Tab
<code>\f</code>	Formfeed
<code>\b</code>	Backspace
<code>\a</code>	Bell (beep)
<code>\e</code>	Escape
<code>\007</code>	Ottale ASCII (007 è Bell)
<code>\x8a</code>	Esadecimale ASCII
<code>\cM</code>	Carattere di controllo (CTRL-M)
<code>\\</code>	Backslash
<code>\"</code>	Doppio apice
<code>\l</code>	Prossima lettera minuscola
<code>\L</code>	Tutte le nuove lettere in minuscolo fino a <code>\E</code>
<code>\u</code>	Prossima lettera maiuscola
<code>\U</code>	Tutte le nuove lettere in maiuscole fino a <code>\E</code>
<code>\Q</code>	Preponi Backslash a caratteri non alfanumerici fino a <code>\E</code>
<code>\E</code>	Termina il comando di <code>\L</code> , <code>\U</code> e <code>\Q</code>

Lezioni pratiche sulla programmazione in Perl

Lezione 5

Numeri

Nel linguaggio Perl i numeri sono rappresentati per mezzo di simboli, di naturale e facile interpretazione, detti anche più in generale letterati. Di seguito sono riportati alcuni esempi di numeri:

```
18
# numero intero positivo
1_000_000
# un milione, si noti come il carattere sottolineea venga ignorato dal Perl
-31
# numero intero negativo
7.023
# numero in virgola mobile
7.023e4
# vale 7,023 x 104
01215
# numero ottale intero positivo
0xff812
# numero esadecimale intero positivo
```

Una delle comode caratteristiche del Perl è che il programmatore non deve preoccuparsi di valori interi, decimali a singola o a doppi precisione, in quanto l'interprete Perl tratta tutti i valori numerici allo stesso modo utilizzando la doppia precisione. Per quel che concerne invece i numeri ottali ed esadecimali, il Perl impiega le stesse identiche convenzioni del linguaggio C, pertanto verranno considerati numeri ottali tutti quelli che iniziano con il numero 0 e composti da cifre da 0 a 7, mentre verranno considerati esadecimali tutti quelli che iniziano con la sequenza 0x e sono composti da simboli tra 0-9, tra a-f e tra 0f).

Operazioni numeriche

Le operazioni numeriche gestite dal Perl corrispondono a tutte le operazioni più tradizionali: somma sottrazione, moltiplicazione, divisione, resto della divisione, potenze. Facciamo adesso alcuni esempi specificando due scalari numerici numero1 e numero2:

```
$numero1 = 20;
$numero2 = 4;
$a = $numero1 + $numero2;
# addizione vale 24
$b = $numero1 - $numero2;
# sottrazione vale 16
$c = $numero1 * $numero2;
# moltiplicazione vale 80
$d = $numero1 / $numero2;
# divisione vale 5
$e = $numero1 % $numero2;
# resto della divisione vale 0
$f = $numero1 ** 2;
# potenza vale 202
$g = $numero1 ** $numero2;
# potenza vale 204
```

In questo primo esempio abbiamo ricavato altri scalari numeri secondari, mediante le più comuni operazioni, vengono detti scalari secondari, in quanto dipendono dal valore dei primi due, a questo punto possiamo proseguire nella scrittura del nostro script aggiungendo le seguenti stringhe:

```
print "\$numero1 vale $numero1\n";
print "\$numero2 vale $numero2\n";
print "-----\n";
print "la somma tra $numero1 e $numero2 corrisponde a $a \n";
print "la sottrazione tra $numero1 - $numero2 corrisponde a $b \n";
print "il prodotto tra $numero1 e $numero2 corrisponde a $c \n";
print "la divisione tra $numero1 e $numero2 corrisponde a $d \n";
print "il resto della divisione tra $numero1 e $numero2 corrisponde a $e \n";
print "il valore di $numero1 elevato alla potenza di 2 corrisponde a $f \n";
print "il valore di $numero1 elevato alla potenza di $numero2 corrisponde a $g \n";
```

Salvate il tutto col nome di operazioni.pl, ed avviate il programma per la verifica. Il risultato dovrebbe essere il seguente:

```
$numero1 vale 20
$numero2 vale 4
-----
la somma tra 20 e 4 corrisponde a 24
la sottrazione tra 20 e 4 corrisponde a 16
il prodotto tra 20 e 4 corrisponde a 80
la divisione tra 20 e 4 corrisponde a 5
il resto della divisione tra 20 e 4 corrisponde a 0
il valore di 20 elevato alla potenza di 2 vale 400
il valore di 20 elevato alla potenza di 4 vale 160000
```

Altre operazioni utilizzate frequentemente sono quelle di autoincremento "++" e di autodecremento "--", Con la prima si incrementa di uno il valore della variabile a cui si applica e con la seconda a la si decrementa. Alcuni esempi:

```
$a++
# valore di $a prima di effettuare l'incremento
++$a
# valore di $a dopo aver effettuato l'incremento
$a--
# valore di $a prima di effettuare il decremento
--$a
# valore di $a dopo aver effettuato il decremento
```

Operatori di confronto tra numeri

Anche gli operatori di confronto sono gli stessi che normalmente ci aspetteremo da valori numerici e cioè:

```
minore <
minore o uguale <=
uguale ==
diverso !=
maggiore >
maggiore o uguale >=
```

Facciamo un esempio:

```
$a=20;
$b=13;

$a < $b
# espressione falsa
$a <= $b
# espressione falsa
$a == $b
# espressione falsa
$a > $b
# espressione vera
$a >= $b
# espressione vera
```

Il concetto tra vero e falso in Perl viene considerato in modo pragmatico, proprio come il senso comune suggerisce. Provate ad esercitarvi con queste poche nozioni a scrivere qualche semplice script come quello dell'esempio.

Lezioni pratiche sulla programmazione in Perl

Lezione 6

Stringhe

Nel linguaggio Perl viene definita stringa un valore composto da una sequenza di caratteri, senza alcun logico criterio d'ordine. Per sequenza di caratteri intenderemo tutto l'insieme ASCII più gli escape (\n, per esempio).

Anche se le stringhe sono formate da più caratteri esse possono essere considerate come un unico valore, analogamente al concetto d'insieme, nella teoria degli insiemi, dove pur avendo più elementi corrisponde a un tutto unico, gli Array in Perl possono contenere tanti elementi, ma nel loro insieme questi elementi definiscono il valore di un array.

La semplicità del linguaggio porta a distinguere due soli tipi di stringhe, quelle con interpolazione delle variabili e interpretazione degli escape e quelle senza interpolazione ed interpretazione. Si dicono stringhe interpolate, e di conseguenza interpretate, quelle delimitate in un comando tra i doppi apici, mentre non sono interpolate né interpretate quelle tra apici singoli, ad esempio:

```
$a = ' felici e contenti';  
# $a definisce la nostra variabile stringa  
print "E vissero $a \n";  
# chiediamo al programma di stampare  
# la stringa in modo interpolato  
print 'E vissero $a \n';  
# chiediamo al programma di stampare  
# la stringa in modo non interpolato
```

Se provate ora ad eseguire il programma, avrete come risultato per la prima stringa "E vissero felici e contenti" con un ritorno a capo dato dall'interpretazione dell'escape \n, mentre nella seconda stringa otterrete "E vissero \$a \n" senza nessuna manipolazione del contenuto del comando print.

Quindi nel Perl l'impiego dei doppi apici risulta un qualcosa in più rispetto ad una semplice convenzione, e lo dimostra anche il fatto che esistono diversi modi per rappresentare gli apici:

```
qq/ciao mamma/  
# corrisponde a "ciao mamma"  
q/ciao mamma/  
# corrisponde a 'ciao mamma'  
qq{ciao papà}  
# corrisponde a "ciao papà"  
q{ciao papà}  
# corrisponde a 'ciao papà'
```

Il Perl, comunque, riconosce un caso particolare di stringa dove la differenza tra doppi apici e singoli è del tutto irrilevante, e cioè il caso in cui il valore espresso dalla stringa sia nullo, stringa nulla, di seguito sono riportati gli esempi per rappresentarle:

```
" "  
' '  
qq//  
q//  
qq{}  
q{}
```

La stringa nulla gioca comunque un ruolo molto importante nella definizione di concetto tra vero e falso, come vedremo tra poco.

Operazioni con stringhe

Le operazioni più importanti, che si ottengono con le stringhe sono la concatenazione e la ripetizione, un esempio di concatenazione può essere il seguente:

```
$a = 'Costantini';
$b = 'Giordano';

$c = $a . $b;
# vale CosatantiniGiordano
# se vogliamo inserire uno spazio
# tra le variabili $a e $b potremmo
# aggiungere una stringa nulla
$d = $a . " " . $b;
# vale Costantini Giordano
# specificare il valore di $a e $b è stata solo
# mia scelta per rendere più comprensibile l'esempio
# ma avremmo potuto anche farne almeno scrivendo
# direttamente quanto segue:
$e = "Costantini" . " " . "Giordano";
# avremmo potuto inoltre ricavare lo spazio
# distanziatore tra i due valo in questo modo
$f = "Costantini " . "Giordano";
```

La ripetizione delle stringhe corrisponde ad una operazione molto originale, ecco alcuni esempi:

```
$a = "Ciao " x 3;
# mediante l'impiego di questa operazione
# sarà possibile specificare il valore "ciao ciao ciao"
# allo scalare $a, il carattere "x" indica d'effettuare la
# ripetizione ed il numero tre scelto a piacere indica
# il numero di ripetizioni da effettuare
$b = 'ciao' x 2;
# non vi è alcuna differenza tra gli apici doppi e quelli singoli
$c = "-" x 80;
$d = "3" x (2*3);
# vale 333333
$e = "3 " x (2*3);
# vale 3 3 3 3 3 3
$f = (3-2) x 4;
# vale 1111
```

Operatori di confronto tra stringhe

Gli operatori di confronto tra stringhe sono equivalenti a quelli impiegati per i valori numerici, con l'unica differenza che al posto dei normali operatori di confronto simbolici, utilizzati per i valori numerici, useremo i seguenti:

```
minore lt
minore o uguale le
uguale eq
diverso ne
maggiore gt
maggiore o uguale ge
```

Facciamo un esempio:

```
$a=franco;  
$b=bollo;  
$c=$a . $b;
```

```
$a lt $b  
# espressione falsa  
$a le $b  
# espressione falsa  
$a == $b  
# errore di sintassi  
$a eq $b  
# espressione falsa  
$a gt $b  
# espressione vera  
$a gl $b  
# espressione vera  
$c eq "francobollo"  
# espressione vera
```

Vero e falso

Per quanto riguarda il discorso sulle stringhe, si considerano false la stringhe nulle ed ovviamente vere la stringhe non nulle, come le più normali e semplici logiche di pensiero, anche per il Perl si considera vero tutto ciò che è vero e falso ciò che non appare vero, si considera falsa anche la stringa '0', attenzione però a non confondere con '00' che viene considerata vera come una sequenza di due caratteri.

In sostanza il Perl non ha valori booleani (vero o falso), ma solo stringhe nulle, stringhe "0", e il numero 0 vengono valutati false tutto il resto risulta vero.

Provate ad esercitarvi con queste poche nozioni a scrivere qualche semplice script come quello dell'esempio.

Lezioni pratiche sulla programmazione in Perl

Lezione 7

Valori indefiniti undef

Oltre ai più classici valori numerici e stringa, il linguaggio Perl riconosce un altro valore speciale e cioè il valore indefinito undef, attribuibile dall'interprete sia come variabile numerica uguale a 0, sia come stringa nulla. Ad esempio:

```
$a = 7 * undef;
# in questo caso la variabile undef
# verrà considerata come una
# variabile numerica uguale a 0
print " $a \n";
# il risultato sarà uguale a zero
print "<----->\n";
$b = "tempo:";
print "$b undef\n";
# In questo caso invece la variabile
# undef sarà considerata come
# una variabile stringa ed
# il risultato sarà uguale a
# "tempo: undef"
print "<----->\n";
$c = undef;
print "$b . $c \n";
# il risultato sarà ancora uguale
# a "tempo: null"
```

Salvatelo ed eseguitelo per la prova di verifica.

Lezioni pratiche sulla programmazione in Perl

Lezione 8

Operazioni tra Numeri e Stringhe

Nelle operazioni tra valori numerici e stringhe, il linguaggio Perl si comporta nel modo più naturale possibile, senza creare spiacevoli disagi al programmatore, pertanto i principi fondamentali da seguire sono due, e cioè:

- Se si usa un valore numerico in una stringa, il valore numerico viene convertito anch'esso in stringa, i cui caratteri corrispondono alle cifre del valore numerico.
- Se si usa un valore stringa nel contesto numerico, il valore stringa verrà convertito in un valore numerico, in base ai caratteri numerici contenuti all'interno della stessa, diversamente si avrà un valore numerico uguale a zero.

Vediamo ora alcuni esempi

```
$a = "11 " . "maggio " . "2000";  
# in questo caso i valori numerici  
# 11 e 2000 vengono addizionati al  
# valore stringa "maggio" mediante  
# l'operatore di addizione stringhe  
# per risultato si otterrà che anche  
# i valori 11 e 2000 saranno considerati  
# come stringhe  
print "$a \n";  
# vale 11 maggio 2000  
$b = "11 " + "maggio " + "2000";  
# in questo caso invece abbiamo usato l'operatore  
# addizione numerico + costringendo pertanto  
# l'interprete a considerare la stringa maggio  
# come numero, in questo caso uguale a 0  
print " $b \n";  
# vale 2011  
$c = 15000 - "2500 lire";  
print " $c \n";  
# vale 12500
```

Provate quindi a salvare ed ad eseguire lo script, per la prova di verifica.

Lezioni pratiche sulla programmazione in Perl

Lezione 9

Privilegi degli operatori

Alla pari dei più diffusi linguaggi di programmazione anche il Perl ha le sue brave regole di precedenza e di associatività degli operatori all'interno di espressioni più complesse, contenenti un insieme di operatori ed operandi, al fine di produrre un nuovo valore, abbiamo già visto alcuni semplici esempi tramite la lezione sulle operazioni, ora vedremo di illustrare situazioni più complesse.

Cominciamo quindi a stabilire la precedenza tra quali operatori hanno il privilegio di essere valutati prima di altri. Come ormai ovvio il Perl fa di tutto per rendere le cose più normali e naturali possibili, risolvendo qualunque ambiguità su cosa debba essere eseguito prima, in base alle più normali regole della matematica e dell'algebra, facciamo un esempio per capirci ulteriormente meglio:

```
$a = 3;
$b = $a**2/2;
# l'espressione $b vale 32/2
# quindi uguale a 4,5
# e non 32/2 con un valore errato di 3
# nel qual caso avremmo dovuto
# scrivere $a**(2/2);
print "$b \n";
# vale 4,5
$c = $a**2*2;
print "$c \n";
# vale 18
```

L'associatività risolve invece un altro problema d'ambiguità inerente alle modalità d'incorporazione degli operatori con ugual livello di priorità. L'associazione tra operatori può essere sia destra, sia sinistra che addirittura nulla, vediamo ora un esempio:

```
$a = 3;
$b = $a**2**2;
print "$b \n";
# in questo caso l'associazione tra operatori
# è rivolta a destra, in quanto risulta di maggiore
# priorità risolvere prima tutte le operazioni di potenza
# quindi il risultato sarà dato da $a**(2**2) uguale a 81
$c = 6/3*2;
# pur non essendoci alcun conflitto tra la divisione e la
# moltiplicazione assoceremo, per semplicità, a sinistra
# i privilegi dell'operazione in questo modo (6/3)*2
print "$c \n";
# vale 4
```

Ovviamente si consiglia sempre l'uso delle parentesi per risolvere l'espressioni più complesse, ed anche per facilitarne la leggibilità dello script stesso ad altri.

Per esercizio provate a calcolare la seguente espressione

$$s = \frac{Q^2(0,32+1.25^S)S}{Q^2(0,32)^4+(1,25+S)^{S/Q}}$$

s= valore dell'espressione da calcolare

Q= 1.845

S= 5.35

Lezioni pratiche sulla programmazione in Perl

Lezione 10

Interagire con il programma

Fino ad oggi ci siamo limitati a scrivere dei piccoli programmi Perl statici, che si limitavano a stampare sul video l'impostazioni da noi predefinite. Adesso cercheremo di ampliare le nostre conoscenze in modo da creare uno script in grado di poter interagire con il suo esecutore, grazie all'impiego di un operatore speciale detto parentesi angolari, capace di poter leggere i dati immessi dalla tastiera, rappresentato dalla sigla <STDIN>. Ecco, come ormai solito, un esempio pratico:

```
print "Inserisci il tuo nome!\n";
$name = <STDIN>;
# ti da' la possibilità di inserire
# il tuo valore da tastiera
print "Inserisci la tua età!\n";
$eta = <STDIN>;
chomp ($name);
chomp ($eta);
# rimuove il carattere escape \n
# dalle stringhe ottenuto con la pressione d'invio
print "Il tuo nome è $name e hai $eta anni";
```

Ora se andate ad eseguire il programma da una finestra del Dos, l'esecuzione si bloccherà in attesa di un vostro input chiedendovi appunto di inserire il vostro nome, dopo aver immesso il vostro nome, sarà necessario cliccare su invio, per permettere al programma di continuare la sua esecuzione, con la pressione d'invio però avrete creato anche un carattere escape di più che toglieremo mediante il comando chomp, dopo di che allo stesso modo vi verrà chiesto d'inserire la vostra età, ed infine otterrete per risultato una stringa del tipo "Il tuo nome è Giordano e hai 24 anni". Fate attenzione a non confondere chomp con chop, nella scrittura dei vostri script, in quanto se pur in questo caso non succede nulla, in altri potrebbe portare dei risultati spiacevoli. Ad esempio:

```
$a = "Giordano\n";
chomp ($a);
chomp ($a);
# il risultato sarà sempre il medesimo
# in quanto il comando chomp si limita
# a cancellare il carattere escape \n
# mentre
chop ($a);
chop ($a);
# il comando chop si occupa di cancellare
# l'ultimo carattere della stringa sia esso
# escape che alfanumerico ottenendo prima
# Giordano e poi Giordan
```

Per esercizio, provate a riscrivere l'espressione di ieri, dando però la possibilità di inserire il valore di Q e S da tastiera, ed in fine in base alle nozioni raccolte fino ad oggi stampare un numero stringhe N, inserito da tastiera, con il risultato dell'espressione.

Lezioni pratiche sulla programmazione in Perl

Soluzione esercizio

Soluzione dell'esercizio "Lezione 9"

Ovviamente le seguenti procedure, non sono le uniche esistenti, ma valgono per coloro che hanno trovato delle difficoltà. Fatta eccezione di casi specifici, nei quali vi viene chiesto di adottare un particolare procedimento, l'importante è ottenere il risultato e non come lo si ottiene.

```
print "Calcoliamo il valore della";
print "\nseguente espressione\n";
print "\n";
print "s =(Q**2*(0.32+1.25**S)*S)/(Q**2*0.32**4+(1.25+S)**(S/Q))";
print "\n\n";
$b = "1.845";
print "dove Q vale $b ";
$d = "5.35";
print "e S vale $d\n\n";
$f = $b**2*(0.32+1.25**$d)*$d;
$g = $b**2*0.32**4+(1.25+$d)**($d/$b);
$h = $f/$g;
$e = "L'espressione s vale $h";
print "$e\n";
```

Soluzione dell'esercizio dato il 15/06/2000

```
print "Calcoliamo il valore della";
print "\nseguente espressione\n";
print "\n";
print "s =(Q**2*(0.32+1.25**S)*S)/(Q**2*0.32**4+(1.25+S)**(S/Q))";
print "\n\n";
print "Inserisci il valore di Q!\n";
$b = <STDIN>;
print "Inserisci anche il valore di S!\n";
$d = <STDIN>;
chomp ($b);
chomp ($d);
print "\nBenone, ora Q vale $b ";
print "e S vale $d\n\n";
$f = $b**2*(0.32+1.25**$d)*$d;
$g = $b**2*0.32**4+(1.25+$d)**($d/$b);
$h = $f/$g;
print "Specifica il numero di volte che\n";
print "vuoi far visualizzare il valore di s\n";
$N = <STDIN>;
chomp ($N);
print "\n";
print "eccoti il risultato\n";
$e = "L'espressione s vale $h\n";
print "$e x $N";
```

Lezioni pratiche sulla programmazione in Perl

Lezione 11

Variabili array

Come abbiamo accennato in precedenza, il linguaggio Perl, distingue il tipo di variabili limitandosi a due soli gruppi, e cioè le variabili scalari e le variabili array.

Si definisce un array una sequenza ordinata di valori scalari, in grado di distinguere un unico valore. Come di consueto, nel linguaggio Perl, anche l'impiego di array risulta semplice e flessibile, risolvendo agevolmente le situazioni più ostiche. Vediamo quindi un esempio pratico per rendere meglio l'idea:

```
@collaboratori=('Fabio', 'Davide', 'Alessandro', 'Salvatore');
$c=@collaboratori;
print "Io, Giordano ho $c collaboratori, ed insieme cerchiamo di dare corpo al nostro sito\n";
print "i miei $c collaboratori si chiamano @collaboratori\n";
print "ma mentre io, $collaboratori[0], $collaboratori[1] e $collaboratori[2] ci diamo da fare\n";
print "$collaboratori[3], non si è ancora fatto sentire";
```

Bene, come potrete notare, ci è bastato creare un array di valori per poterci sbizzarrire nella creazione di testi stravaganti, ognuno di voi potrà scegliere i valori che vorrà ed in qualsiasi numero. Se prestate ulteriormente attenzione all'esempio noterete che con la definizione del nostro array abbiamo creato anche un numero di variabili scalari pari al numero dei valori specificati all'interno dell'array, richiamabili in qualsiasi momento dal nostro script, fate molta attenzione a iniziare il conteggio da 0 e non da 1. Vediamo a titolo di esempio, come far stampare su stringhe differenti i valori di ogni scalare definito in un array:

```
@a=('qui','quo','qua');
# definiamo il valore dell'array
print "$a[0]\n";
# chiediamo di stampare il primo scalare
# definito dall'array @a chiamando lo scalare
# con lo stesso nome e cioè $a
print "$a[1]\n";
print "$a[2]\n";
print "$a[3]\n";
# questo richiamo non darà alcun errore,
# in quanto l'array @a non definisce un quarto elemento
# per risultato otterremo solo un ritorno accapo
# ottenuto dall'escape \n, mentre $a[3] verrà semplicemente
# ignorato, (questa sì che è flessibilità!!!)
```

Torniamo adesso all'esempio dei miei collaboratori, ed ampliamo il programma nel seguente modo:

```
@collaboratori=('Fabio', 'Davide', 'Alessandro', 'Salvatore');
$c=@collaboratori;
print "Io, Giordano ho $c collaboratori, ed insieme cerchiamo di dare corpo al nostro sito\n";
print "i miei $c collaboratori si chiamano @collaboratori\n";
print "ma mentre io, $collaboratori[0], $collaboratori[1] e $collaboratori[2] ci diamo da fare\n";
print "$collaboratori[3], non si è ancora fatto sentire\n";
@a=@collaboratori;
push(@a,'Giordano');
$a=@a;
print "In tutto siamo $a e ci chiamiamo @a\n";
```

Come potete vedere le cose che si possono fare con le array sono infinite, ma infinitamente semplici, come ad esempio uguagliare un array ad un altro, rompere il contenuto di un array tramite il comando push per aggiungere altri elementi, ecc ecc.

Per oggi abbiamo dato una bella infarinatura sugli array, nella prossima lezione vedremo di addentrarci più nel dettaglio.

Lezioni pratiche sulla programmazione in Perl

Lezione 12

Assegnazione dei valori

Nella precedente lezione, abbiamo visto in modo sommario come sia possibile ed semplice utilizzare le variabili array nel linguaggio Perl scripting, vediamo ora di approfondire il discorso, illustrando più nel dettaglio le funzionalità dell'array. Come abbiamo già accennato un array altro non è che un valore di una variabile contenente a sua volta altre variabili, sia esse scalari sia anche array stesse, non è necessario inoltre specificare la variabile array, per applicare in uno script una sua variabile interna, vediamo quindi degli esempi.

```
@abc=(' a',' b',' c');  
$abc="a b c";
```

Queste due variabili pur avendo lo stesso nome, definiscono due valori completamente differenti tra loro e senza alcun tipo di legame interno, mentre se avessimo scritto quanto segue:

```
@abc=(' a',' b',' c');  
$abc[0]="a b c";
```

avremmo avuto per risultato un nuovo array uguale a "@abc=(' a b c',' b',' c');".

Potremmo inoltre definire un array vuoto e decidere in seguito di aggiungere i suoi valori interni, richiamandoli con nuovi scalari, (si ricorda ancora una volta, che come per il linguaggio C, C++, Java, ecc., il conteggio dei valori interni ad un array parte da zero anziché da uno), vediamo quindi un altro esempio:

```
@a = ();  
$a[0] = "a1 ";  
$a[1] = "a2 ";  
$a[2] = "a3 ";  
$a[3] = "a4 ";  
$a[7] = "a8 ";
```

Come potrete notare, non esiste limite al numero di valori interni ad un array, come anche non esiste alcuna regola nel creare in un particolare ordine, i nuovi scalari. In questo caso infatti ci siamo limitati a definire il valore dei nuovi scalari partendo dal primo elemento, indicizzato col numero 0, fino al quarto, indicizzato col numero 3, dopo di che abbiamo fatto un bel salto per arrivare all'ottavo elemento, indicizzandolo col numero 7, ignorando il quinto sesto e settimo elemento.

Ora se ci accingiamo a completare il nostro programma chiedendoli di stampare il valore di @a, il nostro caro interprete Perl senza fare troppe domande, si limiterà ad eseguire alla lettera i nostri comandi, proprio come noi vogliamo. Completiamo l'esempio nel seguente modo:

```
print @a;  
print "\n";  
# con queste prime stringhe chiediamo  
# all'interprete di stampare il valore di @a  
# mentre con l'escape \n chiediamo di tornare a capo  
print "@a\n";  
# ora invece abbiamo chiesto di stampare la variabile @a  
print "$a[0]\n";  
print "$a[1]\n";  
print "$a[2]\n";  
print "$a[3]\n";  
print "$a[4]\n";  
print "$a[5]\n";  
print "$a[6]\n";  
print "$a[7]\n";  
# ora invece abbiamo chiesto di stampare  
# ad uno ad uno tutti gli scalari compresi in @a
```

Eseguendo questo programma vi accorgete ulteriormente della flessibilità del linguaggio, nel valutare le variabili non definite dal programmatore. Ma le meraviglie non finiscono qui, proviamo quindi ad ottenere qualcosa di più complicato, e cioè cerchiamo di ottenere il valore di un array, pur non specificandolo nel nostro programma, ma creando solo degli indici scalari, non ci avete capito niente vero? poco male, vediamo subito un esempio molto semplice:

```
$a[0]= "33 ";
$a[1]= "trentini ";
$a[2]= "entrarono ";
$a[3]= "a ";
$a[4]= "Trento ";
print @a;
```

Come potrete notare ci siamo limitati soltanto a definire un indice di scalari associati al nome "a", ora pur non avendo specificato una variabile array per "a", eseguendo il nostro script l'interprete sarà in grado di determinare il valore della variabile array @a.(che dite..? non è fantastico?) , altre piccole comodità del nostro linguaggio, riguardano gli array contenenti solo stringhe, e quelli contenenti un elenco continuo di numeri, ad esempio:

```
@g=("Filippo","e","panaro.");
# equivale al modo più sbrigativo
# alla seguente espressione
@h=qw(Filippo e panaro.);
print "@g\n";
print "@h\n";
@i=(4,5,6,7,8,9,10,11,12,13,14);
# equivale al modo più sbrigativo
# alla seguente espressione
@l=(4..14);
print "@i\n";
print "@l\n";
# ma ancora, è anche possibile definire una
# variabile array di array nel seguente modo
@m=(@g,@i);
print "@m\n";
```

Lezioni pratiche sulla programmazione in Perl

Lezione 13

Operazioni degli array

Come abbiamo visto gli array altro non sono che un insieme di dati o meglio valori, dotati di una grande flessibilità sintattica e manipolabili a piacimento, ma per semplificare ulteriormente il loro impiego ed utilità, il Perl aggiunge un insieme di operazioni esclusive per l'array.

Vediamo alcuni esempi:

Push e pop

Come ormai saprete gli array sono composti da una lista ordinata, in un certo criterio, di valori indicizzati tra 0 ed n valori -1, generalmente considerata una pila di dati (stack), parliamo adesso degli operatori che agiscono sull'ultimo elemento della pila, chiamata in gergo coda della lista, essi sono **push** per aggiungere valori in coda all'array e **pop** per togliere valori all'array, ecco come poter implementare tali operatori:

```
@abc=(12,23,34,45,56);
push (@abc,67,78,89);
print "@abc\n";
# l'array @abc ora vale
# (12,23,34,45,56,67,78,89)
pop(@abc);
print "@abc\n";
# ora l'array @abc ora vale
# (12,23,34,45,56,67,78)
# non è detto che si debba perdere
# l'ultimo valore di un array, ma è
# possibile salvarlo in un nuovo scalare
# per un successivo impiego
$a=pop(@abc);
print "@abc\n";
print "$a\n";
# ovviamente tali operatori si possono
# applicare anche ad array nulli, senza
# creare alcun disagio all'interprete
# che si limiterà a restituirci un valore null
@ab=( );
$b=pop @ab;
print "@ab\n";
print "$b\n";
push (@ab,"ciao", "mondo")
print "@ab\n";
```

Shift e unshift

Prendiamo ora in considerazione la testa della nostra pila di valori, e cioè la parte sinistra del nostro array, mediante l'operatore **shift** è possibile togliere il primo elemento della pila, mentre con l'operatore **unshift** è possibile aggiungere nuovi valori in testa alla pila, facciamo alcuni esempi:

```
@az=(1,2,3,4,5);
shift @az;
print "@az\n";
# varrà (2,3,4,5)
unshift(@az,7,9,6,1,0,-5);
print "@az\n";
# varrà (7,9,6,1,0,-5,2,3,4,5)
# ovviamente anche in questo caso
# è possibile salvare il valore di testa
# che intendiamo eliminare dall'array
# per un successivo impiego
$a=shift @az;
print "$a\n";
# $a varrà 7
```

Sort e reverse

Il linguaggio Perl ci permette di stabilire l'ordine degli elementi interni di un array, in modo da soddisfare l'esigenza di un ordinamento ascendente o discendente, sia per valori numerici, sia per valori stringa, gli operatori impiegati per ottenere tale risultato saranno la funzione **sort** e **reverse**. Ecco vi un esempio:

```
@a=(1,6,8,3,'3,45','6,12345',0,'0,0000012',2,4,76,5,7);
# ora se vogliamo rendere la stessa stringa in un modo
# più ordinato dovremmo creare un secondo array con
# la funzione sort
@b=sort (@a);
print "@a\n";
# l'array @a rimarrà invariato e varrà
# 1 6 8 3 3,45 6,12345 0 0,0000012 2 4 76 5 8 7
print "@b\n";
# l'array @b varrà
# 0 0,0000012 1 2 3 3,45 4 5 6 6,12345 7 76 8
# ora è anche possibile specificare l'ordine inverso
# tramite la funzione reverse, che però andrà applicata
# all'array @b e non a @a
@c=reverse (@b);
print "@c\n";
# varrà "8 76 7 6,12345 6 5 4 3,45 3 2 1 0,0000012 0"
```

Ovviamente tali funzioni valgono anche per i numeri negativi, ma il loro ordinamento sarà un po' differente di come ci aspetteremo, quindi fate molta attenzione nel loro impiego, vediamo un esempio:

```
@a=(-34,-54,-2,-12,-3,-1);
@b=sort (@a);
print "@b\n";
# varrà -1 -12 -2 -3 -34 -54
```

In ultimo vediamo un esempio di ordinamento di valori stringhe interne ad un array:

```
@a=(casa,mare,'città',persone,internet,amici,in,internet);
@b=sort (@a);
print "@b\n";
# varrà "amici casa città in internet internet mare persone"
# seguendo un ordine alfabetico, nel caso di array contenenti
# valori stringa e valori numerici, i valori numerici avranno
# la precedenza sulle stringhe, come anche i valori numerici
# negativi avranno la precedenza su quelli positivi
```

Lezioni pratiche sulla programmazione in Perl

Lezione 14

Istruzioni foreach

Concludiamo il discorso sugli array, introducendo una funzione molto potente, ma allo stesso tempo molto semplice da implementare nei nostri script, e cioè la funzione foreach, in pratica questa istruzione non fa altro che creare un blocco di istruzioni interne ad un programma, un po' come si fa per le funzioni if che vedremo in futuro. Per adesso accontentiamoci di vedere alcuni esempi:

```
@settimana=('lunedì','martedì','mercoledì','giovedì','venerdì','sabato');
$festa=domenica;
foreach $festa (@settimana)
{
    print "$festa\n giorno feriale\n"
}
print "$festa\n giorno di riposo";
```

Con questo semplice esempio è possibile processare ogni singolo elemento di @settimana applicando la funzione foreach alla variabile scalare \$festa la quale si occuperà di processare i valori dell'array al solo interno del blocco foreach, rappresentato tra le parentesi graffe, e non anche all'esterno, dove la variabile \$festa riprende il suo valore predefinito. Il risultato dovrebbe assomigliare a questo:

```
lunedì
 giorno feriale
martedì
 giorno feriale
mercoledì
 giorno feriale
giovedì
 giorno feriale
venerdì
 giorno feriale
sabato
 giorno feriale
domenica
 giorno di riposo
```

Vediamo ancora qualche altro esempio pratico:

```
#Esempio 1
@num=(2,3,4);
foreach $num(@num)
{
    $num **=2
    # eleva al quadrato ogni elemento dell'array
}
print "@num\n";
# @num varrà "4 9 16"
```

```
#Esempio 2
@num=(2,3,4);
foreach $num(@num)
{
    $num *=2
    # moltiplica per 2 ogni elemento dell'array
}
print "@num\n";
# @num varrà "4 6 8"
```

```
#Esempio 3
@num=(2,3,4);
foreach $num(@num)
{
    $num /=2
```

```

    # divide per 2 ogni elemento dell'array
}
print "@num\n";
# @num varrà "1 1,5 4"

#Esempio 4
@num=(2,3,4);
foreach $num(@num)
{
    $num +=2
    # addiziona 2 ad ogni elemento dell'array
}
print "@num\n";
# @num varrà "4 5 6"

#Esempio 5
@num=(2,3,4);
foreach $num(@num)
{
    $num -=2
    # sottrae 2 ad ogni elemento dell'array
}
print "@num\n";
# @num varrà "0 1 2"

#Esempio 6
@num=(2,3,4);
foreach $num(@num)
{
    $num %=2
    # individua il resto nella divisione per 2 di ogni elemento dell'array
}
print "@num\n";
# @num varrà "0 1 0"

#Esempio 7
@num=(2,3,4);
foreach $num(@num)
{
    $num **=-2
    # eleva alla potenza di -2 ogni elemento dell'array
}
print "@num\n";
# @num varrà "0.25 0.111111111111111 0.0625"

#Esempio 8
@num=(2,3,4);
$num = 0;
foreach $numeri(@num)
{
    $num = $numeri + $num
}
print "$num\n";
# $num vale 9

```

Vista così questa istruzione non rappresenta un gran ché ma vedremo in futuro la sua utilità nel proseguire delle lezioni. Questa lezione chiude il nostro ciclo sulle array, provate ora con le informazioni acquisite a risolvere i seguenti esercizi:

Trovate almeno tre modi distinti di calcolare la somma numerica degli elementi interni a @somma avente i seguenti valori: "33 trentini, 118, 916, perl5,00xx, ore 13.00, fine"

Scrivere un programma che chieda all'utente di inserire da tastiera tre valori per l'array @a e due valori per l'array @b, e poi calcoli la somma degli array, attacchi @a in testa a @b ed ordini i valori in modo ascendente.

Lezioni pratiche sulla programmazione in Perl

Lezione 15

Variabili array associati, hash

Con quest'ultima lezioni sulle variabili hash, o hash e basta, concluderemo il discorso sulle variabili plurali. A differenza dei normali array, le variabili hash sono un insieme ordinato di valori associati in coppie chiave/valore, in modo da creare una vera e propria tabella di dati, ideale per la rappresentazione di qualsiasi relazione binaria.

Per meglio comprendere il discorso farò un esempio in tema degli europei di quest'anno in base alla situazione attuale (l'Italia deve ancora giocare l'ultima partita di una qualificazione scontata), iniziamo con l'impostare 4 variabili hash, uno per ogni girone di qualificazione, ogni hash per essere considerata tale necessita di iniziare con il carattere %, con un po' di fantasia è possibile ricordare tale carattere, considerando i due cerchietti divisi dallo slash, proprio come se fosse la relazione chiave/valore dell'hash, ecco come procedere:

```
%girA=('Portogallo',6,'Inghilterra ',3,'Germania',1,'Romania ',1);
# Primo girone, come potete vedere per ogni squadra o impostato
# il suo punteggio
%girB=('Italia ',6,'Belgio ',3,'Turchia',1,'Svezia ',1);
%girC=('Jugoslavia',6,'Norvegia ',3,'Spagna',3,'Slovenia ',1);
%girD=('Francia',6,'Olanda ',6,'Rep. Ceca ',0,'Danimarca ',0);
```

Questo modo di rappresentare le variabili hash se pur valido e perfettamente funzionante, è ormai obsoleto e dalla versione 5 del Perl si è adottato un altro stile molto più ordinato ed anche più leggibile, e cioè:

```
%girA=(
    Portogallo =>6,
    Inghilterra => 3,
    Germania => 1,
    Romania => 1
);
%girB=(
    Italia => 6,
    Belgio => 3,
    Turchia => 1,
    Svezia => 1);
%girC=(
    Jugoslavia => 6,
    Norvegia => 3,
    Spagna => 3,
    Slovenia =>1
);
%girD=(
    Francia => 6,
    Olanda => 6,
    Rep. Ceca => 0,
    Danimarca =>0
);
```

Questo procedimento, come potete notare è molto più ordinato e leggibile rispetto al primo ed inoltre è stato introdotto il simbolo "=>" che ne evidenzia il concetto di relazione tra chiave e valore, ricordate che nell'insiemi hash le chiavi vengono considerate sempre variabili stringhe pur contenendo numeri.

Come ormai di consueto, la flessibilità e semplicità del linguaggio ci permette di poter effettuare diverse operazioni proprio come abbiamo fatto con gli array, senza incappare in particolari disagi, ad esempio potremmo copiare un vecchio array %a in uno nuovo %b (%a=%b) oppure trasformare gli elementi di un hash in un array (%a=@b). Ma vediamo alcuni esempi più nel dettaglio:

Accesso ad una tabella hash

Accedere ad una tabella hash corrisponde nel ritrovare un valore all'interno della tabella per mezzo di una chiave, continuando ad utilizzare l'esempio precedente (meglio se il secondo metodo), potremmo effettuare un operazione di ricerca. Ad esempio, data la chiave Italia cerchiamo di ottenere il suo corrispettivo valore associato, nel nostro caso il punteggio delle partite, in questo modo:

```
$puntIT=$girB{Italia} ;
# $puntIT corrisponde alla variabile a cui vogliamo
# associare il valore della chiave, nel nostro caso
```

```

# il valore corrisponde al punteggio ottenuto dalla squadra
# e la chiave è il nome della squadra
print $puntIT;
# in questo modo è possibile stampare il punteggio dell'Italia
# ma stasera ha vincere!! nessun problema
# basta aggiornare il nostro script in questo modo
$girB{Italia} = 9;
# in questo modo data la chiave possiamo stabilire il valore
$puntIT=$girB{Italia} ;
# associa nuovamente il punteggio
print $puntIT;
# ristampa il punteggio

# purtroppo ogni gara ha i suoi sconfitti, che non hanno più
# ragione di continuare ad esistere all'interno delle nostre
# tabelle, per aggiornare il nostro script elimineremo le squadre
# che usciranno fuori, usando la funzione delete
# come è ormai ovvio nel girone D le squadre Rep.Ceca e Danimarca risultano
fuori
# quindi le cancelleremo in questo modo
delete($girD{Rep. Ceca});
delete($girD{Danimarca});
# provvedete voi ad aggiornare le classifiche
# come per gli array è possibile visualizzare il valore delle
# hash tramite la funzione print (print "%girD\n");

```

Operazioni con hash

Il Perl mette a disposizione dei programmatori le seguenti operazioni e funzioni, al fine di manipolare una tabella hash e cioè:

- **estrazione delle chiavi**
- **estrazione dei valori**
- **estrazioni di una coppia**
- **cancellazione di una coppia**
- **esistenza di una coppia**

Ma vediamo nel dettaglio

La funzione da impiegare al fine di estrarre le chiavi da una variabile hash, e *keys*, vediamo un esempio:

```

%settimana=(
    Lun=>feriale,
    Mar=>feriale,
    Mer=>feriale,
    Gio=>feriale,
    Ven=>feriale,
    Sab=>feriale,
    Dom=>festivo
);
@sigle=keys(%settimana);
print "@sigle\n";
# sarà uguale ad un elenco dei giorni della settimana
# ma non conserverà l'ordine con le quali sono state elencate
# la funzione keys è usata molto con l'istruzione foreach
# ecco un esempio che vi stamperà l'elenco dei valori dei gg
foreach $chiave (keys(%settimana))
{
    print "$settimana{$chiave}\n";
}

```

La funzione per estrarre i valori interni ad un hash, anziché le le chiavi, è *values*, vediamo un esempio:

```

%settimana=(
    Lun=>feriale,
    Mar=>feriale,
    Mer=>feriale,

```

```

        Gio=>feriale,
        Ven=>feriale,
        Sab=>feriale,
        Dom=>festivo
    );
    @day=values(%settimana);
    print "@day\n";

```

Per estrarre dall'hash una coppia chiave/valore, è possibile far uso della funzione *each*, se tale funzione viene invocata ripetutamente per lo stesso hash, estrae una coppia di valori per volta senza però avere un controllo sulla coppia da voler estrarre, fino ad esaurire tutte le coppie disponibili, continuando a basarci sul hash %settimana riportato sopra, illustriamo il funzionamento di each mediante l'istruzione while

```

%settimana=(
    Lun=>feriale,
    Mar=>feriale,
    Mer=>feriale,
    Gio=>feriale,
    Ven=>feriale,
    Sab=>feriale,
    Dom=>festivo
);
while(($chiave, $valore)=each(%settimana))
{
    print "Il giorno di $chiave è $valore\n";
}
# questo comando stamperà un elenco di chiavi/valori tipo
# "Il giorno di Dom è festivo" fino ad esaurire le chiavi, mentre
# con la semplice richiesta di stampa o di associazione ad una variabile
# array o scalare otterremo un richiamo casuale ad una coppia valore/chiave
print each(%settimana);print "\n";
@a = each(%settimana);print "\n";
print @a;print "\n";

```

La funzione *delete* per la cancellazione di una coppia chiave/valore l'abbiamo già vista nell'esempio sugli europei. Per verificare l'esistenza di una chiave all'interno di un hash si usa la funzione *exists*. Riprendiamo l'hash %settimana per il nostro esempio.

```

%settimana=(
    Lun=>feriale,
    Mar=>feriale,
    Mer=>feriale,
    Gio=>feriale,
    Ven=>feriale,
    Sab=>feriale,
    Dom=>festivo
);
print "esiste Mar\n" if exists $settimana{Mar};

```

Per esercizio provate a mantenere costantemente aggiornate le tabelle degli europei2000 riunendole ma mano in un unico hash che rimarrà ovviamente con un'unica coppia, speriamo l'Italia. Provate inoltre a scrivere un programmino in grado di raccogliere le coppie chiavi/valori da tastiera, almeno 6 coppie, che chieda all'utente se vuole visualizzare l'intera tabella, solo le chiavi, solo i valori, o tutti e tre.

Lezioni pratiche sulla programmazione in Perl

Lezione 16

Istruzioni decisionali

Mediante l'introduzione delle istruzioni decisionali, daremo finalmente inizio alla programmazione vera e propria in Perl, fino ad oggi ci siamo limitati ad illustrare le basi minime del linguaggio, dando l'idea di cosa fosse una variabile e quali potevano essere i suoi operatori, finalmente siamo arrivati a potervi presentare qualcosa di più complesso, ma solo nei risultati e non anche nella sintassi del linguaggio. Riprendendo in esame tutte le lezioni illustrate in precedenza vedremo come poter implementare uno script con le istruzioni decisionali più frequenti e cioè **if**, **else** e **elsif**, vediamo subito un esempio semplice semplice:

```
print "Quanti punti ha totalizzato l'Italia, nel primo girone di
qualificazione?\n";
$punti=<STDIN>;
chomp($punti);
if ($punti == 9)
{
    print " con $punti punti siamo dei campioni\n"
}
elsif ($punti >= 6)
{
    print "con $punti punti non siamo affatto male\n"
}
elsif ($punti > 3)
{
    print "con $punti punti dobbiamo migliorare molto\n"
}
else
{
    print "con $punti punti facciamo proprio schifo\n"
}
```

L'istruzione **if** corrisponde ad una espressione booleana tra vero e falso, nel nostro programma chiediamo di inserire il valore numerico della variabile \$punti, che indichi il punteggio attuale dell'Italia per la qualificazione ai quarti di finale, dopo di ch  chiediamo al programma di confrontare il punteggio dato con alcune ipotesi, decidendo di associare un commento a seconda del punteggio, infatti mediante l'istruzione decisionale **if**, uguale al nostro italiano "se", associamo il commento "...siamo dei campioni", se il punteggio   uguale a 9, cio  il massimo, tramite invece l'istruzione **elsif**, che corrisponde ad una via di mezzo tra **if** ed **else**, in italiano altrimenti, serve per abbreviare e rendere pi  leggibile lo script avendo comunque la stessa funzione di if, infatti anche in questo caso associamo il commento "...non siamo affatto male" nel caso in cui il punteggio dovesse essere maggiore o uguale a 6, prima di illustrarvi il procedimento pi  complesso di ottenere il medesimo risultato, senza per  l'uso di **elsif**, finiamo di spiegare il significato di **else**, in pratica lo script verifica se l'istruzione if ed elsif siano vere, se tra tali istruzioni se ne trova una che soddisfi il valore booleano vero, allora verr  stampato il commento associato nel blocco tra parentesi graffe (indispensabili anche nel caso in cui non volessimo riportare nulla al loro interno) altrimenti verr  stampato il valore booleano falso interpretato da **else**, nel nostro caso se avessimo inserito un punteggio inferiore o uguale a tre, non avremmo soddisfatto alcuna istruzione vera, e avremmo ottenuto il commento "...facciamo proprio schifo". Vediamo ora il modo pi  complesso e disordinato di ottenere lo stesso risultato:

```
print "Quanti punti ha totalizzato l'Italia,";
print " nel primo girone di qualificazione?\n";
$punti=<STDIN>;
chomp($punti);
if ($punti == 9)
{
    print " con $punti punti siamo dei campioni\n"
}
else
{
    if ($punti >= 6)
    {
        print "con $punti punti non siamo affatto male\n"
    }
    else
    {
        if ($punti > 3)
        {
```

```

        print "con $punti punti dobbiamo migliorare molto\n"
    }
    else
    {
        print "con $punti punti facciamo proprio schifo\n"
    }
}
}

```

Come potrete notare risulta molto più leggibile e semplice utilizzare **elsif**, al posto di nuovi blocchi nidificati di istruzioni **else**, qualche altra condizione decisionale in più e ci saremmo persi tra le parentesi graffe. Poco più sopra avevo affermato dell'importanza delle parentesi graffe che definiscono i blocchi d'istruzioni, anche nei casi in cui volessimo definire una sola istruzione, se non addirittura nessuna, ripeto questo, per coloro che sono abituati a linguaggi tipo il C il C++ Java ecc. dove le parentesi graffe possono anche essere omesse nel caso di una sola istruzione oppure nessuna. Ma se ciò può farvi apparire il Perl più sofisticato rispetto agli altri linguaggi, ecco che lo stesso ci viene incontro con una espressione abbreviata nel caso in cui abbiamo una singola istruzione decisionale. Nell'esempio precedente per indicare che un punteggio pari a 9 doveva restituirci l'istruzione "...siamo dei campioni" abbiamo usato il seguente procedimento

```

print "Quanti punti ha totalizzato l'Italia,";
print " nel primo girone di qualificazione?\n";
$punti=<STDIN>;
chomp($punti);
if ($punti == 9)
{
    print " con $punti punti siamo dei campioni\n"
}

```

ora mediante l'impiego di un'espressione abbreviata, il nostro script apparirà così:

```

print "Quanti punti ha totalizzato l'Italia,";
print " nel primo girone di qualificazione?\n";
$punti=<STDIN>;
chomp($punti);
    print "con $punti punti siamo dei campioni\n" if $punti == 9;

```

Come potete vedere risulta ancora più semplice e più leggibile, in pratica dice: stampa "bla bla bla" se lo scalare Tizio soddisfa, con un valore booleano vero, il confronto con il valore Caio, se invece il confronto risulta falso il programma ignorerà tale istruzione andando avanti, nel nostro caso non va da nessuna parte e finisce lì. Oltre all'istruzione **if** ve ne sono molte altre da poter implementare come ad esempio l'istruzione contraria **unless**, in italiano "se non è", infatti mentre **if** si usa per dire "se a = b, fai questo" la funzione unless ragiona in questo modo: "se la condizione a = b è falsa, fai questo. Ad esempio:

```

print "Quanti punti ha totalizzato l'Italia,";
print " nel primo girone di qualificazione?\n";
$punti=<STDIN>;
chomp($punti);
unless ($punti < 0)
{
    print "Avere $punti punti è sempre meglio di 0\n"
}

```

Anche con **unless** si può utilizzare **else**, ma non anche **elsif**, vediamone ora l'espressione abbreviata:

```

print "Quanti punti ha totalizzato l'Italia,";
print " nel primo girone di qualificazione?\n";
$punti=<STDIN>;
chomp($punti);
    print "Avere $punti punti è sempre meglio di 0\n" unless $punti < 0;

```

Praticamente non si fa altro che chiedere di stampare un "bla bla bla" solo nel caso il confronto sia falso, anziché vero come per l'istruzione **if**.

Lezioni pratiche sulla programmazione in Perl

Lezione 17

Istruzioni decisionali

Oltre alle più utilizzate istruzioni decisionali *if* ed *else*, il Perl mette a disposizione dei programmatori, altre funzioni per ottenere il medesimo risultato, o quantomeno simile, vediamo alcuni esempi con gli operatori ternari "?", operatori AND "&&" e gli operatori OR "||".

```
$altezza="1.86";  
($altezza > "1.55") ? (print "Sei alto!") : (print "Sei basso!");
```

In pratica l'operatore ternario non fa altro che rendere abbreviata l'istruzione *if* ed *else* standard

```
$altezza="1.86";  
if ($altezza gt "1.55")  
{  
    print ("Sei alto!")  
}  
else  
{  
    print ("Sei basso!")  
}
```

Come potete notare l'impiego dell'operatore ternario "?" è più adatto sia per semplicità, sia per comprensibilità, certo quest'esempio non da' molta alternativa sul risultato quindi vediamo subito di scriverne uno un po' più interattivo:

```
print (Inserisci la tua altezza in metri);  
$altezza =<STDIN>;  
chomp($altezza);  
($altezza > 1.55) ? (print "Con l'altezza di $altezza, puoi dire di essere alto!") :  
(print "Con l'altezza di $altezza, non puoi dire di essere alto!");
```

Come potete vedere l'istruzione ternaria non fa altro che verificare la veridicità del confronto tra lo scalare altezza da noi introdotto ed un valore di confronto predefinito per la verifica, dopo di che se il confronto risulta vero ci stamperà la prima espressione dopo il Carattere ternario "?" altrimenti ci stamperà la seconda espressione posizionata dopo il carattere ":".

Anche se non è indispensabile far uso delle parentesi tonde per definire l'espressioni, se ne consiglia ugualmente l'impiego sia per migliorarne la leggibilità sia per non incappare in qualche errore di precedenza ed associatività tra gli operatori.

Vediamo ora un esempio mediante l'impiego dell'operatore AND "&&":

```
$altezza="1.86";  
($altezza > "1.55") && (print "Sei alto!");
```

Lo stesso programma equivale alla all'impiego dell'istruzione decisionale *if*:

```
$altezza="1.86";  
if ($altezza gt "1.55")  
{  
    print ("Sei alto!")  
}
```

In pratica non si fa altro che confrontare la condizione di confronto, e solo nel caso risulti vera, il programma restituirà l'espressione alla destra dell'operatore AND, altrimenti tale espressione verrà completamente ignorata.

La funzione dell'operatore OR "||" è simile al quella di AND, soltanto che ragiona al contrario, (un po' come fa *unless* per *if*) ad esempio:

```
$altezza="1.86";  
($altezza < "1.55") || (print "Sei alto!");
```

Se il confronto risulta vero, l'interprete ignorerà l'espressione posta alla destra dell'operatore OR, mentre nel caso risulti falsa restituirà l'espressione, in sostanza il nostro programma dice: se \$altezza non è minore di 1,55, sei alto!
Ovviamente i due operatori AND ed OR possono essere associati all'interno dello stesso script, ad esempio potremmo scrivere:

```
$altezza="1.86";  
($altezza > "1.55") && (print "Sei alto!");  
($altezza > "1.55") || (print "Sei basso!");
```

Certamente questo non è il modo più naturale per scrivere un programma di confronto, per il quale sarebbe stato più opportuno utilizzare l'operatore ternario "?", ma vale per dare un'idea del suo utilizzo.

Per esercizio provate a scrivere un programmino che vi chieda d'inserire la vostra età, e che vi chieda anche d'inserire l'età con la quale ci si possa ritenere una persona adulta, che useremo come valore di confronto, per verificare se, chi inserisce il valore di confronto si ritiene adulto o meno, in base alla sua età.

Lezioni pratiche sulla programmazione in Perl

Lezione 18

Istruzioni iterative

Si definiscono istruzioni iterative, tutte quelle istruzioni che vengono eseguite ripetutamente all'interno di un blocco fino al raggiungimento della condizione. Nel linguaggio Perl le istruzioni iterative o ripetitive (come volete) sono le seguenti:

- **while e until**
- **do e until**
- **for**
- **foreach**

While

Vediamo subito qualche esempio semplice semplice, d'istruzione ripetitiva **while**, maggiormente impiegata in Perl.
esempio1

```
$a=3;
while ($a < 9){
    $a++;
}
print "$a\n";
```

Questo semplice esempio, non fa' altro che controllare che il valore di \$a sia minore di 9, se tale istruzione risulta vera, incrementa ripetutamente di uno il valore di \$a sino a quando non raggiunge il valore 9. Infatti se chiediamo di stampare \$a dopo il blocco d'istruzione otterremo un bel 9. Se invece \$a fosse stato maggiore di 9, ad esempio 13, l'istruzione **while** sarebbe stata falsa e quindi ignorata.

esempio2

```
$a=7;
while ($a > 3){
    $a--;
}
print "$a\n";
```

Questo semplice esempio, a differenza del primo controlla che il valore \$a sia maggiore di 3, se tale espressione risulta vera decrementa ripetutamente di uno il valore di \$a fino ad ottenere il valore 3.

esempio 3

```
$a=3;
while ($a > 0){
    $a++;
}
print "$a\n";
```

Prestate una particolare attenzione a questo esempio, in quanto, se pur sintatticamente corretto, vi porterà nella trappola di una istruzione ciclica infinita, senza via di uscita dal blocco, se non con l'arresto manuale dell'esecuzione.

Molto simile a **while** è l'istruzione ripetitiva **until**, in pratica non fa altro che verificare la seguente ipotesi: fin tanto la condizione non risulta vera fai questo e/o quello. Vediamone un esempio:

esempio4

```
$a=3;
until ($a > 10){
    print "$a non soddisfa ancora la condizione\n";
    $a++;
}
print "$a soddisfa la condizione\n";
```

Lascio a voi il piacere di scoprire il risultato di tale script.

Come per **if** anche per le istruzioni **while** ed **until**, il Perl ha predisposto una forma abbreviata vediamo alcuni esempi: esempio2

```
$a=7;
while ($a > 3){
    print "$a è ancora maggiore di 3\n";
    $a--;
}
```

questa è la forma normale, mentre la forma abbreviata di while sarà: esempio5

```
$a=7;
print "$a è ancora maggiore di 3\n" while $a-- >3;
```

Considerando l'esempio4 e vediamone la forma abbreviata: esempio6

```
$a=3;
print "$a non soddisfa ancora la condizione\n" until $a++ > 10;
print "$a soddisfa la condizione\n";
```

All'interno del ciclo **while** vi possono essere anche dei modificatori di percorso, che si occupano di alterare la normale esecuzione al verificarsi di una determinata circostanza. Gli operatori di cui parliamo sono:

- **next**: interrompe l'esecuzione del gruppo di istruzioni e riprende dalla valutazione della condizione
- **last**: esce definitivamente dal ciclo **while** senza curarsi del gruppo di istruzioni
- **redo**: ripete il ciclo, senza valutare e verificare nuovamente l'espressione della condizione, e senza curarsi del gruppo di istruzioni

Vediamo alcuni esempi esempio7

```
$a=3;
until ($a > 10){
    if ($a == 9){
        next;
    }
    print "$a non soddisfa ancora la condizione\n";
    $a++;
}
print "$a soddisfa la condizione\n";
```

In questo caso si chiede di incrementare di uno il valore di \$a fino a quando l'espressione **until** risulta vera, nel caso in cui il valore di \$a dovesse uguagliarsi a nove, il ciclo va nuovamente ripetuto, senza considerare le istruzioni che seguono next, nel nostro caso il programma andrà in panne rimanendo in esecuzione, abbastanza simile è anche la funzione di **redo**, mentre con **last** abbiamo un modo elegante per terminare ed uscire dal ciclo, vediamone un esempio. esempio8

```
$a=3;
until ($a > 10){
    if ($a == 9){
        last;
    }
    print "$a non soddisfa ancora la condizione\n";
    $a++;
}
print "$a soddisfa la condizione\n";
```

In questo caso il ciclo tenterà di ripetersi fino ad ottenere un valore maggiore di 10, ma dato che viene incrementato di uno ad ogni susseguirsi del ciclo quando arriverà a valere 9, il programma interromperà il ciclo interno al blocco ed uscendone con grazia.

Ad esempio in circostanze tipo l'esempio3, il quale genera un ciclo infinito, potremmo utilizzare la funzione di last per bloccare il programma nel momento in cui dovesse raggiungere un valore predefinito.

esempio9

```
$a=3;
while ($a > 0){
  if ($a == 30){
    last;
  }
  print "$a\n";
  $a++;
}
```

Provate a sostituire 30 con un bel numerone ed otterrete un bell'effetto matrix.

Lezioni pratiche sulla programmazione in Perl

Lezione 19

Istruzioni iterative

Nella precedente lezione abbiamo visto dei semplici esempi d'istruzioni cicliche mediante l'impiego di *while*, in questa lezione concluderemo il discorso sulle istruzioni iterative illustrando il comportamento delle istruzioni **do**, **for** e **foreach**.

Istruzioni **do..while** e **do..until**

La funzione di **do** si integra all'istruzione *while*, eseguendo prima le istruzioni ripetitive di blocco, e poi valuta il valore dell'espressione *while*, vediamo un esempio:

esempio1

```
$a=1;
do{
    print "$a\n";
    $a++;
}
while ($a < 30);
```

In questo caso, l'istruzione **do** effettua ripetitivamente un incremento di uno sullo scalare *\$a* e si arresta nel momento in cui soddisfa la condizione *while*, questo significa che qualunque sia l'espressione da soddisfare il ciclo viene avviato almeno una volta, se ad esempio la condizione *while* fosse stata *\$a<1* anziché *<30* al termine del programma avremmo avuto *\$a=2*, scrivete l'esempio successivo per verifica:

esempio2

```
$a=1;
do{
    print "$a\n";
    $a++;
}
while ($a < 1);
print '$a' . " vale $a\n";
```

Ricordatevi di utilizzare il punto e virgola ";" per la condizione *while*, in quanto non essendo seguita da un blocco d'iterazione viene interpretato come un comune comando. Ovviamente dato che *until* è simile a *while*, è anche possibile impiegare l'istruzione **do** aggiungendola ad *until*, vediamo un esempio.

esempio3

```
$a=1;
do{
    print "$a\n";
    $a++;
}
until ($a < 30);
print '$a' . " vale $a\n";
```

L'unica differenza tra **do..while** e **do..until** è che ragionano in modo opposto.

Istruzione **for**

Per la gioia di tutti i programmatori C, Java e simili, vediamo subito come impiegare l'istruzione **for**, in modo da semplificare notevolmente la forma sintattica dell'istruzioni *while*, in un esempio pratico

esempio4

```
for ($a=0;$a<10;$a++){
    print '$a' . "$a non soddisfa la condizione " . '$a<' . "10\n";
}
```

In questo esempio, mediante l'uso della sola istruzione **for**, abbiamo definito il valore iniziale di *\$a*, posta una condizione di verifica, ed imposto l'istruzione da effettuare nell'eventualità che la condizione non fosse soddisfatta.

Istruzione foreach

Anche l'istruzione **foreach**, già illustrata nelle precedenti lezioni, costituisce un'istruzione ripetitiva, qui ne vediamo un ulteriore esempio

esempio5

```
$a=131;
# poniamo $a uguale ad un valore a caso 13
# allo stesso modo specifichiamo il valore di $b
$b=3;
@a=(2,3,4,5,6,7);
# definiamo un array @a, come sappiamo pur
# avendo lo stesso nome di $a, non esiste
# alcuna relazione tra i due
foreach $a (@a)
# però all'interno di un'istruzione foreach
# possiamo associare lo scalare $a all'array
# @a senza variare il valore dello stesso al
# di fuori del blocco, sarebbe stata la stessa
# se avessimo utilizzato lo scalare $b, ma
# anche $$ o $pinco_pallino, ma forse non avremo
# reso l'idea, dell'indipendenza
{
  $a=$a+$b;
  # non facciamo altro che incrementare di tre
  # ogni valore di @a
  print "$a\n";
  # chiediamo di stampare in sequenza tutti
  # gli elementi di @a incrementati di $b
}
print "-----\n$a\n";
# se chiedete di stampare $a fuori dal blocco
# foreach il suo valore tornerà ad essere di 131
```

esempio6 (molto simile all'esempio5)

```
$a=131;
$b=3;
@a=(2,3,4,5,6,7);
print '@a vale ' . "@a" prima dell'istruzione foreach\n";
foreach $a (@a)
{
  $a **=$b;
  # eleviamo alla potenza di $b ogni valore
  # di @a
  print "$a\n";
}
print "-----\n$a\n";
# è vero che $a al di fuori del blocco torna
# a valere il suo valore normale
# ma @a, una volta terminato il blocco rimarrà
# con il suo valore incrementato
print '@a vale ' . "@a" dopo dell'istruzione foreach\n";
```

Lezioni pratiche sulla programmazione in Perl

Lezione 20

Espressioni regolari

Ogni linguaggio di programmazione ha i suoi bravi punti di forza rispetto ad altri che lo fanno preferire in particolari situazioni, ebbene oggi conosceremo le peculiarità del Perl nella gestione delle espressioni regolari. Sicuramente vi sarà capitato qualche volta di voler ricercare all'interno del vostro sistema un determinato file o cartella grazie alla comoda utility in 'start'>'trova'>'file o cartelle', bene quella si può definire un'espressione regolare. Definire un'espressione regolare in uno script Perl equivale ad inserire nello stesso un micro-linguaggio con specifiche proprie. Vediamone subito qualche esempio semplice semplice:

```
$a="Ma quanto mi piace programmare in Perl";
if($a =~ m/piace/){
# per scrivere il carattere '~' si consiglia di
# tenere premuto il tasto ALT mentre premete in
# sequenza i numeri 1 2 e 6 dal tastierino numerico
print "Ho trovato \'piace\'\n"
}
else{
print "Non ho trovato \'piace\'\n"
}
```

In questo semplice script, non abbiamo fatto altro che chiedere al programma di analizzare la stringa definita dallo scalare \$a, dicendogli di verificare l'esistenza della parola "piace", tramite la funzione if inoltre, li chiediamo di stampare una espressione se la ricerca ottiene un esito positivo cioè vero, mentre con else li chiediamo di stampare una espressione nel caso in cui la ricerca dovesse fallire ed ottenere quindi una istruzione falsa. Questo è il succo del programma, vediamo invece d'interpretare la funzione dei suoi componenti:

\$a distingue uno scalare, già ampiamente illustrato nel corso delle lezioni
if verifica se la condizione è vera
else entra in gioco quando if risulta falsa.

(\$a =~ m/piace/) questa è la condizione da verificare e corrisponde alla nostra espressione regolare distinta dall'operatore "=~" alla destra della variabile da analizzare e da comando m (match = uguale) posto a sinistra dell'espressione regolare, delimitata tra le due barre, inoltre non è indispensabile specificare l'operatore m in quanto viene impostato di default dall'interprete se non ne trova di diversi. Vediamo ora di realizzare un esempio più interattivo:

```
$a="Ma quanto mi piace programmare in Perl";
print "In questo programma data una variabile \"$a\n";
print "uguale a ".$a."Ma quanto mi piace programmare in Perl\n";
print "vi chiediamo di inserire un nome da cercare\n";
$b=<STDIN>;
chomp ($b);
if($a =~ m/$b/){
print "Ho trovato ".$b."\n"
}
else{
print "Non ho trovato ".$b."\n"
}
```

Se siete stati in grado di comprendere la funzione dei due esempi, avrete già compreso più del 60% su come definire in modo grossolano le espressioni regolari, un restante 30% cercheremo d'illustrarvelo nelle istruzioni successive e in fine l'ultimo 10% dovrete mettercelo voi con la pratica e l'esperienza in modo da raffinare le conoscenze acquisite. Vediamo ora di illustrarvi qualche peculiarità delle espressioni regolari, cominciando dai tipi di caratteri speciali, per avere una idea di caratteri speciali andate a rivedere i costruttori nella lezione [quattro](#), bene quelli sono alcuni dei caratteri speciali più classici, altri meno classici ma più caratteristici nelle espressioni regolari sono:

- o **\d** cioè una qualunque cifra tra 0 e 9;
- o **\D** cioè un qualunque carattere che non sia una cifra tra 0 e 9;
- o **\w** cioè un qualunque carattere tra a e z, A e Z, 0 e 9;
- o **\W** cioè un qualunque carattere che non sia compreso tra a e z, A e Z, 0 e 9;
- o **\s** cioè un generico carattere di spazio (vengono considerati caratteri spaziatrici anche i costruttori \n, \r, \t, \f)
- o **\S** cioè un generico carattere che non sia di spazio
- o **.** Il carattere punto "." indica un qualunque simbolo nell'insieme dei carattere

Vediamo ora alcuni esempi

```
$a="Ma quanto mi piace programmare in Perl5.00";
if($a =~ /Perl\d/){
print "Soddisfatta la prima\n"
}
if($a =~ /in\sPerl\d/){
print "Soddisfatta la seconda\n"
}
if($a =~ /in\wPerl/){
print "Soddisfatta la terza\n"
}
```

L'espressione `/Perl\d/` sarà soddisfatta da qualunque parola Perl seguita da una cifra qualsiasi, quindi `Perl5.00` soddisfa la nostra espressione, mentre se la posto di "Ma quanto mi piace programmare in Perl5.00" avremmo scritto "Ma quanto mi piace programmare in Perl 5.00" l'espressione `/Perl\d/` non sarebbe stata soddisfatta, anche la seconda espressione `/in\sPerl\d/` sarà soddisfatta, in quanto tra 'in' e 'Perl' esiste un carattere di spazio `\s` e dopo Perl esiste un carattere cifra. In fine l'ultima espressione regolare `/in\wPerl/` non può essere soddisfatta in quanto tra la parola 'in' e 'Perl' non esiste alcun carattere definito da `\w`.

```
$a="Ma quanto mi piace programmare in Perl5.00";
if($a =~ /.){
print "Ho trovato1\n"
}
```

Questa espressione è sempre soddisfatta, in quanto il carattere `.` individua un qualunque carattere della stringa, vale come il nostro asterisco `*` nell'utility di ricerca file e cartelle, cerca `*.jpg` corrisponde a cercare tutti i file con estensione `.jpe`

Molto spesso al posto dei caratteri speciali `\w` e `\d` si usa inserire la classe equivalente tra parentesi quadre, come `[0-9]` per il carattere `\d` oppure `[a-z,A-Z,0-9]` il carattere `\w` all'interno delle classi individua un intervallo, risulta preferibile l'uso delle classi in quanto si ha un maggior controllo dei caratteri, infatti se volessimo verificare l'esistenza di un carattere cifra tra 3 e 6 l'impiego di `\d` non ci sarebbe di grande aiuto, rispondendo sempre vero, mentre la classe `[3-0]` risolverebbe egregiamente i nostri problemi. Questo ci fa comprendere che abbiamo la possibilità di specificare la nostra classe man mano che ci occorre senza dipendere da classi standard, `[03-69a-grst@#?!$£]` anche questa è una classe corretta e non fa altro che verificare l'esistenza di un carattere 0 oppure tra 3 e 6 poi il 9, poi tra 'a' e 'g' ecc. ecc.

Abbiamo inoltre detto che il carattere `-` viene utilizzato per individuare un intervallo, ma se volessimo individuare anche l'esistenza di esso all'interno della classe basterà scrivere in questo modo `"\-"` ad esempio `[a-zA-Z\-,]`, ricordate però che il Perl non è molto pignolo nei dettagli, almeno ché non glielo diciamo noi, quindi analizzando il contenuto della classe e la stringa in cui ricercare, esso si blocca appena verifica l'esistenza del primo carattere contenuto nella classe.

Tra le varie funzionalità delle classi è anche possibile specificare i caratteri che non ci devono essere in una stringa mediante il carattere `^`, vediamone un esempio:

```
$a="Apache3000 è un bel sito";
if($a =~ /Apache[^3]/){
print "tutto ok, niente 3\n"
}
else{
print "peccato ho trovato il 3\n"
}
```

Questo programma verifica all'interno di `$a`, che dopo la parola Apache non vi sia un numero uguale a 3, nel nostro caso invece c'è, e quindi non potrà essere soddisfatta.

Lezioni pratiche sulla programmazione in Perl

Lezione 21

Moltiplicatori di caratteri

Nelle espressioni regolari formate da sequenze di caratteri può capitare che uno stesso carattere o una parola siano ripetute più volte in una stringa. Per questo motivo il micro-linguaggio delle espressioni regolari prevede l'impiego di moltiplicatori. Se ad esempio si volessero rappresentare tutte le sequenze di numeri interi con un numero di cifre compreso tra un minimo di uno e un massimo di cinque, potremmo usare la semplice espressione:

```
/\d{1,5}/
```

Il moltiplicatore è specificato per mezzo di parentesi graffe (ecco un altro carattere speciale che necessita del carattere di '/' per rappresentare se stesso) e da una coppia di valori

```
{min,max}
```

Il primo, min, indica il numero di evenienze minime richiesto per il carattere (o parola) che si trova a sinistra della parentesi graffa aperta, "{"; il secondo, max, indica il massimo numero di evenienze richiesto per il carattere che si trova a destra della parentesi graffa chiusa, "}". Vediamone alcuni esempi:

```
/cavallo {1,3}\w/      # Verifica "cavallo pazzo", "cavallo
pazzo",
                        # ma non verifica "cavallone". Il carattere che
                        # viene esaminato è quello di spazio vuoto

/(cavallo){1,3}\w/    # Verifica "cavallone", ma non verifica
                        # "cavallo pazzo". Il carattere che
                        # viene esaminato è la sequenza di caratteri
                        # compresi all'interno delle parentesi tonde
```

Naturalmente Perl è un linguaggio pratico e le scorciatoie sono ammesse e soprattutto incoraggiate. Ecco quindi alcune interessanti varianti:

```
{min,}    il numero di occorrenze del carattere deve essere almeno min
{tot}     verifica la corrispondenza del carattere tot volte
*         equivalente a {0,}, cioè da zero occorrenze in su
+         equivalente a {1,}, almeno una occorrenza
?         equivalente a {0, 1}, una occorrenza al massimo
```

Avidità del linguaggio

Pur troppo le funzionalità delle espressioni regolari sono abbastanza imprecise, imputando la colpa all'ingordigia del linguaggio, vediamo di capirci meglio con un esempio:

```
/a{2,6}/ # dovrebbe verificare l'esistenza di una sequenza di caratteri
          # tra 'aa' e 'aaaaaa' ed in effetti è quello che fa, ma non è
affatto  # preciso in quanto verificherebbe anche 'aaaaaaaaa', pur
dovendo  # fermarsi ad un massimo di 6, effettivamente il programma è
stato    # precisissimo, fermando la verifica a 6, diciamo più tosto
          # che siamo stati noi poco chiari ad istruire il programma
```

Analizziamo l'espressione `/a{ 2, 6 }/`. Il programma potrebbe decidere di partire a verificare una sequenza minime di "a", in questo caso due caratteri "a", "aa", ma è ingordo e dalla specifica del moltiplicatore sa che può "esagerare" e partire da una sequenza di sei caratteri "a", "aaaaa". Supponiamo allora di verificare l'espressione con la stringa "aaaaaaaaa" (10 volte "a"). Il programma inizia la ricerca della verifica con una sequenza massimale di sei caratteri "a", "aaaaa", si muove a partire dall'inizio stringa da sinistra a destra e, guarda caso, seguendo questo cammino incontra le prime sei "a" su un totale di dieci. Dunque il programma, avendo trovato la corrispondenza che cercava, si interrompe. In altre parole, il secondo gruppo di quattro caratteri 'a' che forma la stringa è semplicemente ignorato. In pratica l'istruzione `/a{ 2`

,6)/ è una direttiva indirizzata al programma che potrebbe essere così descritta: "inizia vedendo se ce la fai a trovare una sequenza di sei "a". Se non la trovi prova allora con cinque. Se non ce la fai nemmeno con cinque prova con quattro e poi con tre e due. Ma se non ce la fai con due allora arrenditi e ritorna un valore falso". Vediamo un altro esempio che potrà risultare esplicativo

```
/. *bollo/
```

Essa rappresenta un numero qualsiasi di caratteri, eventualmente zero, seguiti dalla sequenza "bollo". Vediamo ora come si comporta il programma per verificare questa espressione regolare. Per prima cosa si colloca sulla prima posizione della stringa. Poi guarda l'espressione regolare e vede che inizia con ".*" cioè un numero qualsiasi di caratteri a piacere escluso \n. Siccome il motore è ingordo, quel "numero qualsiasi di caratteri" per lui è tutti i caratteri possibili della stringa e quindi si posiziona in fondo alla stringa. Essendosi posizionato in fondo alla stringa è chiaro che non riuscirà a verificare l'espressione: non c'è ovviamente spazio in fondo alla stringa per "bollo". Allora decide d'indietreggiare di un passo posizionandosi sul penultimo carattere della stringa. Anche in questo caso non c'è posto per "bollo", al massimo ci sarebbe spazio per il primo carattere, "b", continuando così a indietreggiare finché o trova "bollo", ma ho lo trova nella posizione più a destra possibile nella stringa oppure non lo trova affatto. Vediamo un ultimo esempio suggerito da Randal Schwartz, colui che è per Larry ciò che fu frate Leone per S. Francesco d'Assisi:

```
$_ = 'a xxx c xxxxxxxx c xxx d';  
/a.*c.*d/
```

I caratteri "a", "c" e "d", inframmezzati da qualche altro carattere, ci sono nella stringa e dunque la corrispondenza avverrà. Ma ci sono due "c" all'interno della stringa, una più a sinistra e una più a destra, e il motore verificherà quella più a destra, questo perché per prima cosa il programma si colloca sulla posizione iniziale della stringa. Analizza l'espressione regolare da sinistra a destra e vede che le cose vanno bene: la stringa comincia con "a" e così anche l'espressione regolare. Allora il motore si sposta un po' più a destra nella espressione regolare e trova ".*", ma dato che è ghiotto, prende più caratteri possibili e arriva in fondo alla stringa. Ma poiché con tutto quest'impeto non soddisfa l'espressione torna indietro finché non trova la 'c' più a destra, dopo di che prosegue nella espressione regolare e trova nuovamente ".*" dopo il carattere 'c'. Inizia di nuovo a fare l'ingordo, ma stavolta gli basta indietreggiare di un passo per trovare il carattere 'd' che soddisfa e completa l'espressione. Semplice, no?

Inerzia del linguaggio

Il Perl è linguaggio troppo flessibile per non avere introdotto anche la possibilità di un comportamento definibile "inerte", opposto dell'ingordigia: quando il programma si trova in presenza di un moltiplicatore prova sempre con il minimo numero di occorrenze. Il comportamento pigro è richiesto esplicitamente al programma per mezzo del simbolo '?', ottengono così le varianti pigre dei moltiplicatori che già conosciamo:

```
{min, max}? - prima prova min e poi incrementa fino a max  
{n, }?     - prima prova n e poi incrementa  
{n}?      - uguale alla versione ingorda  
*?        - prima prova zero e poi incrementa  
+?        - prima prova 1 e poi incrementa  
??        - prima prova 0 e poi 1
```

Riprendiamo allora l'espressione di "abel" in modo pigro:

```
/. *?ino/
```

Nella specifica della espressione regolare trova all'inizio ".*", cioè un numero qualsiasi di caratteri qualsiasi escluso "\n". Ma poiché viene chiesto al programma di non aggitarsi e di essere pigro, '?', proverà prima a vedere se con zero caratteri davanti ad "ino" si trova qualcosa a partire dall'inizio della stringa. Se non trova nulla allora prova a vedere se dopo il secondo carattere della stringa si trova "ino" e continua così finché o trova "ino" nella stringa o fallisce per non averlo trovato. Proviamo anche la variante pigra dell'esempio di Randal:

```
$_ = 'a xxx c xxxxxxxx c xxx d';  
/a.*?c.*?d/
```

Avremo che il carattere 'c' della espressione regolare verificherà questa volta il primo carattere 'c' della stringa. Il comportamento pigro è senz'altro meno efficiente di quello ingordo, ma risulta spesso più naturale da usare, e consigliato a coloro che si trovano ancora alle prime armi con le espressioni regolari.

Memorizzazione delle corrispondenze

Le parentesi tonde, oltre ad avere la funzione di racchiudere una sequenza di caratteri da verificare, come avevamo detto in precedenza, hanno anche un'altra, più importante, funzione, e cioè quella di memorizzare le porzioni di stringa in cui è avvenuta la corrispondenza. Vediamone un esempio:

```
print "Inserisci una data valida del tipo\n";
print "XX\XX\XXXX e premi invio\n\n";
chomp($data=<STDIN>);
if($data =~ /(..)\/(..)\/(....)/)
{
    print "gg: $1\n";
    print "mm: $2\n";
    print "aa: $3\n";
}
else
{
    print "Formato data errato";
}
```

Se la stringa in \$data verifica l'espressione regolare ovvero se l'utente inserisce una data valida, ad esempio "25/06/2000", allora il programma memorizzerà in \$1 il contenuto delle prime parentesi, in \$2 il contenuto delle seconde parentesi, che corrisponde al mese, e in \$3 il contenuto delle terze parentesi, corrispondente all'anno. In generale il programma memorizza le porzioni di stringa che corrispondono ai caratteri posti tra parentesi nella espressione regolare secondo l'ordine delle parentesi, ovvero la porzione di stringa che corrisponde al primo gruppo di parentesi tonde è memorizzata in \$ 1, la porzione di stringa che corrisponde al secondo gruppo di parentesi tonde è memorizzata in \$2 e così via per tutti i gruppi di parentesi presenti nella espressione regolare. Se però la corrispondenza tra stringa ed espressione regolare non è verificata, le variabili \$1, \$2, ... non sono valorizzate e anzi non sono nemmeno definite. Le variabili \$ 1, \$2, ... sono variabili a sola lettura e, quindi, non possono essere usate per assegnazioni. Questo semplice meccanismo della memorizzazione ci consente di effettuare un passo in avanti notevole. Infatti sinora avevamo usato le espressioni regolari solo per dire se una stringa verificasse o meno una certa espressione. Fin qui ci siamo accontentati di un sì o di un no, ma è possibile anche sapere come e dove è stata verificata una certa espressione regolare. Questo significa che d'ora in poi saremo in grado di effettuare il riconoscimento di sequenze di caratteri, dette anche pattern. Se ad esempio volessimo estrarre il contenuto di un tag HTML, tipicamente delimitato da parentesi angolari come <HTML> potremmo usare questo semplice programma:

```
chomp($tag_html = <STDIN>);
if($tag_html =~ /<(.)>/)
{
    print "Tag: $1\n";
}
```

Questa espressione verifica un set di caratteri che inizia con "<", ha poi una sequenza di caratteri qualsiasi, ".", di almeno un carattere, "+", e infine con ">". Inoltre la sequenza di caratteri tra parentesi angolari è memorizzata in \$1. Prima di concludere questo paragrafo, vediamo altri simboli speciali che memorizzano anch'essi la porzione di stringa tra parentesi tonde. Essi sono "\ 1 ", "\ 2 ", ... per il primo, il secondo ecc. insieme di parentesi. Essi possono venire usati direttamente all'interno della specifica di una espressione regolare:

```
/vado(.+?)al\1mare/
```

In questa espressione il separatore tra "dire" e "fare", rappresentato dalla sottoespressione "(.+?)", viene memorizzato in "\1 e, dunque anche tra "fare" e "baciare" si attende un separatore identico a quello tra "dire" e "fare". Esempi di stringhe che verificano questa espressione sono:

```
vado-al-mare
vado al mare
```

mentre stringhe che non la verificano sono:

```
vado-al mare
vado al   mare
```

Si osservi la differenza tra \$ 1, \$ 2, ... e \1, \2, ... : le prime sono variabili scalari che possono essere usate anche al di fuori delle espressioni regolari in sola lettura, le seconde invece sono istruzioni nel linguaggio delle istruzioni delle espressioni regolari.

Lezioni pratiche sulla programmazione in Perl

Lezione 22

Abbiamo visto che le parentesi tonde oltre a creare un insieme di elementi, hanno anche la funzione, non sempre necessaria, di memorizzare in variabili speciali la sezioni di stringa che verifica la corrispondenza tra parentesi. Vediamo, quindi, un esempio di come sia possibile impiegare le parentesi senza richiamare la memorizzazione:

```
/(?:aeiou)+/
```

questa espressione verifica almeno una sequenza di caratteri "aeiou", oppure "aeiouaeiou", "aeiouaeiouaeiouaeiou", e via dicendo, senza però sprecare inutilmente memoria per la memorizzazione delle stringhe. Per ostacolare la memorizzazione quindi basta inserire nelle parentesi, la seguente istruzione "?: " a destra della sequenza di caratteri.

Altre variabili speciali

Sempre a proposito di verifica di corrispondenza, ci sono altre tre variabili speciali che servono a indicare quale sezione della stringa l'espressione regolare ha verificato. Esse sono:

```
$& contiene la sezione di stringa che verifica l'espressione
$`  contiene la sezione di stringa che precede quella che verifica l'espressione
$'  contiene la sezione di stringa che segue quella che verifica l'espressione
```

Vediamo un esempio.

```
$a="Italia batte in rete all'Olanda";
$a=~ /rete/;
print "esiste: $& \n";
print "prima: `$` \n";
print "dopo: $(' \n";
```

Asserzioni

Le asserzioni sono direttive particolari delle espressioni regolari che non rappresentano un carattere o una sequenza di caratteri, ma servono ad identificare particolari sezioni all'interno di una stringa, come l'inizio, la fine o i confini di una parola. Vediamo il solito esempio:

```
/^Italia/
```

verifica tutte le stringhe che iniziano con le parole "Italia", come: 'Italia batte in rete all'Olanda', mentre fallirà con stringhe del tipo: 'L'Olanda è stata battuta dall'Italia'. Vediamo un altro esempio di asserzione

```
/Italia$/
```

verifica stringhe che terminano con la parola "Italia", dunque verifica: 'L'Olanda è stata battuta dall'Italia' ma non verifica 'Italia batte in rete all'Olanda'. Le asserzioni più comunemente usate sono le seguenti:

```
^  indica l'inizio della stringa
$  indica la fine della stringa
\b  indica il confine di una parola ("b" sta per boundary)
\B  indica ogni punto che non sia confine di parola
```

Approfondiamo il concetto di confine di una parola. Si definisce confine di una parola il punto che ha da una parte un \w, cioè uno dei caratteri parola, e dall'altra un \W, cioè un carattere qualsiasi che non sia tra quelli usati per una parola. I confini di parola sono detti pure ancora delle parole, anche se noi continueremo a preferire il più accademico asserzioni. Vediamo alcuni confini di parole, rappresentati graficamente con carattere colorato di blu:

un classico esempio di confine

Si noti come questo confine sia posto nel punto tra il carattere spazio e il carattere "d":

altro esempio.

Qui il confine è posto tra il carattere "o" e il punto "."; ma il punto non fa parte dei caratteri ammessi per una parola. Vediamo ora alcuni esempi di espressioni regolari che usano l'asserzione "\b":

```
/\bmare\b/
```

verifica "andiamo al mare", "un giorno al mare", ma non verifica "maremoto" né "altamarea". Vediamo infine altre due asserzioni che hanno la vocazione di guardare in avanti:

```
?= lookahead positivo  
?! lookahead negativo
```

dove i puntini ... stanno per una qualsiasi subespressione nella sintassi delle espressioni regolari. Un esempio servirà a chiarire il concetto più di mille parole di spiegazione:

```
/Italia(?:\s+vincente)/
```

Questa espressione regolare verifica la stringa "Italia" solo se questa è seguita da almeno uno spazio, e dopo lo spazio si trova la stringa "vincente". Il lookahead sia positivo sia negativo è un'asserzione che deve essere verificata nella posizione in cui è inserita nella espressione regolare. Ma attenzione, le parentesi tonde del lookahead non hanno memoria, non hanno nulla a che vedere con \$ 1, \$ 2,... e simili, e anche se la subespressione del lookahead è verificata non fa parte della porzione di stringa che è verificata con successo contenuta in \$&. Il lookahead negativo rappresenta un'asserzione negativa:

```
/\W+(?!\.)/
```

verifica una sequenza di almeno un carattere parola purché questo non sia seguito da una cifra.

Alternative

L'ultimo costrutto per specificare una espressione regolare, che completa la nostra rassegna, è l'operatore '|' che consente di articolare una espressione in più alternative:

```
/Tizio|Caio|Sempronio/
```

Questa espressione verifica uno qualsiasi nome ma, anche, l'espressione regolare è scandita da sinistra a destra e dunque prima viene provato "Tizio", poi "Caio" e infine "Sempronio".

Precedenze

Abbiamo fin qui illustrato come specificare una espressione regolare in Perl introducendo gradualmente i caratteri, i moltiplicatori, il raggruppamento in parentesi, le asserzioni e infine le alternative. Ognuno di questi costrutti sintattici può essere equiparato a una specie di operatore per la costruzione di espressioni regolari e, come per ogni insieme di operatori, è corretto stabilire delle regole di precedenza. Ad esempio nella espressione:

```
/a|b+/
```

è il moltiplicatore "+" che si applica alle due alternative "a|b" o il moltiplicatoresi applica solo a 'b'? Dobbiamo ammettere che intuitivamente il moltiplicatore "+" ha la precedenza sull'alternativa, ma non sempre l'intuito è d'aiuto. Si riporta allora una tabella delle precedenze a partire dai costrutti con precedenza maggiore verso quelli a precedenza minore.

Parentesi	() (?:)
Moltiplicatori	? + * {m,n} {n,} {n} ?? +? *?
Sequenza e asserzioni	Abc ^ \$ \b \B (?:) (?!)
Alternativa	

Interpolazione di variabili

Le espressioni regolari sono una parte integrante del Perl come qualsiasi altro costrutto di linguaggio. L'operatore `m/ ... /` non è una eccezione a questa regola e una caratteristica che lo rende un vero costrutto di linguaggio è l'interpolazione delle variabili. Quest'ultima funziona nello stesso identico modo che abbiamo visto per l'operatore doppio apice:

```
$a = 'Caio';
$_ = 'Caio, Sempronio e Pincopallino';
m/^\$a/;
```

L'espressione è verificata con successo poiché è equivalente a `/^Caio/` in virtù della interpolazione di variabili. Volendo, nella variabile interpolata si possono inserire pezzi di espressione regolare:

```
$a = '^Caio|Pincopallino';
$_ = 'Caio, Sempronio e Pincopallino';
m/^\$a/;
```

Anche in questo caso l'espressione regolare è verificata.

Commenti

Per le espressioni regolari complesse sono in genere utili dei commenti che ne favoriscano la leggibilità e la riusabilità, anche per l'autore stesso della espressione. Perl prevede che i commenti possano essere inseriti in una espressione regolare con la sintassi

```
(?# questo è un commento)
```

I commenti sono scartati immediatamente dal motore delle espressioni regolari. Per la verità questo tipo di commento non è molto usato in pratica perché invece di chiarire ingarbuglia l'espressione. Più spesso i commenti vengono aggiunti sfruttando l'operatore di matching, `m/ ... /`, il quale mette a disposizione una opzione che consente di inserire commenti leggibili:

```
$a="Ormai è estate e la domenica si va a mare";
if(
$a=~/          # espressione commentata
domenica      # grazie all'impiego
/x           # dell'operatore /x
)
{print "trovato\n";
}
else{
print "non trovato\n";
}
```

Per mezzo dell'opzione `/x` si può dare alla espressione un formato più leggibile e si possono aggiungere commenti con il consueto `"#"`. Di fatto quest'ultimo è il vero e unico modo per rappresentare in modo leggibile espressioni regolari di media-alta complessità.

Ignora "case"

Ignorare il "case" significa non considerare se quel carattere sia maiuscolo o minuscolo. Vediamo come ottenere l'effetto di ignorare il "case":

```
$a="Ormai è estate e la domenica si va a mare";
if(
$a=~/DoMEEnICa/i)
{print "trovato\n";
}
else{
print "non trovato\n";
}
```

L'opzione /i rende indipendente dal "case" i caratteri che si trovano nella espressione. Sempre a proposito di maiuscolo e minuscolo, ci sono altri simboli delle espressioni regolari che consentono di attivare rappresentazioni dei caratteri in maiuscolo e minuscolo all'interno di una espressione. Questi sono:

```
\l poni in minuscolo il prossimo carattere  
\u poni in maiuscolo il prossimo carattere  
\L tutto minuscolo fino al prossimo \E  
\U tutto maiuscolo fino al prossimo \E
```

Lezioni pratiche sulla programmazione in Perl

Lezione 23

Matching globale

Come abbiamo già precedentemente accennato l'espressione regolare si riferisce unicamente al primo elemento o parte di stringa che verifica la corrispondenza, quindi se la nostra stringa fosse: "fabio fabio fabio fabio fabio fabio fabio" e decidessimo di verificare l'esistenza della sequenza di caratteri "fabio" con l'espressione `/(?!.fabio)/`, il programma si arresta al primo set di caratteri 'fabio' e non va in cerca di altre eventuali corrispondenze della espressione. Questo comportamento di default può essere modificato per mezzo della opzione `/g`, dove "g" sta per valutazione globale. Vediamo alcuni esempi:

```
$a = "fabio fabio fabio fabio";
while($a =~ /^(\w+)\s(\w+)\s(\w+)/g) {
    print "$1\n" if defined($1);
    print "$2\n" if defined($2);
    print "$3\n" if defined($3);
}
print "\n=====>\n";
if ($a =~ /^(fabio)\s(fabio)\s(fabio)/g){
    print "$1\n";
    print "$3\n";
    print "$3\n";
}
print "\n=====>\n";
if($a =~ /^(((fabio))\s)/){
    print "$1\n" if defined($1);
    print "$2\n" if defined($2);
    print "$3\n" if defined($3);
}
print "\n=====>\n";
print "\nFine :)";
```

La stringa `$a` contiene parole separate da uno spazio vuoto. L'espressione regolare viene applicata a ogni iterazione del ciclo `while` mantenendo memoria della posizione della stringa dove si è verificata l'ultima corrispondenza. Spieghiamo con calma la prima parte dell'esempio. Alla prima iterazione del ciclo viene verificata la prima delle alternative:

```
^(\w+)\s
```

e in tal modo tra parentesi rimane qualsiasi sequenza di caratteri diversa da 0 quindi "fabio" va benissimo ed andrà a valorizzare la variabile `$1`. Il contenuto di `$1` è visualizzato ma non quello di `$2`, `$3` e `$4` perché non sono definite (si noti l'uso della funzione Perl `defined` che ritorna "vero" se una variabile è definita). Il motore allora aggiorna un indice alla posizione dove si è fermata la corrispondenza, ovvero nel punto dopo il primo spazio e prima di "due". Si riparte con la seconda iterazione e, poiché la posizione non è più all'inizio della stringa, è verificata la seconda e la terza alternativa.

```
(\w+)\s(\w+)
```

In questo caso sono definite `$2` e `$3` che contengono rispettivamente "fabio" e "fabio". L'indice è ora aggiornato nella posizione tra lo spazio che segue "tre" e l'inizio di "quattro".

Sostituzione

Un operatore con una sintassi simile a `m/.../` è l'operatore di sostituzione `s/.../.../`. Questo operatore serve per effettuare una o più sostituzioni all'interno di una stringa sfruttando la potenza delle espressioni regolari. La sua sintassi generale è:

```
s/espressione-regolare/sostituzione/
```

Vediamo un semplice esempio:

```
$a = "Io e Fabio siamo amici da sempre";
print "$a\n";
$a =~ s/Io e Fabio siamo/Fabio e Rocker non sono/;
print "$a\n";
$a =~ s/$a/Ma e come se lo fossero!//;
```

```

print "$a\n";
$a =~ s/!//. La stessa cosa vale anche con Alessandro./;
print "$a\n";
$a =~ s/$a/Salvatore invece è un po' fetente./;
print "$a\n";
print "\nFine :)";

```

Vediamo un altro esempio:

```

$a= "La farfalla";
print "$a\n";
$a =~ s/[a]/\@/g;
print "$a\n";

```

Si ottiene "L@ f@r@f@ll@": tutte le "a" di \$a sono sostituite con "@".

L'operatore di traduzione

L'operazione di traduzione tr risolve facilmente casi difficili da affrontare con la sostituzione quali ad esempio sostituire tutte le "a" di una stringa con le "lo" e tutte le "o" con delle "i".

L'operatore tr non usa espressioni regolari e dunque potrebbe sembrare fuori luogo in questo capitolo. In realtà è un utile complemento all'uso delle espressioni regolari.

La sintassi generale dell'operatore tr è:

```
tr/caratteri-da-cercare/caratteri-da-sostituire/
```

Vediamo alcuni esempi.

```
tr/A-Z/a-z/
```

trasforma tutti i caratteri maiuscoli in corrispondenti caratteri minuscoli.

```

$a= "La farfalla";
print "$a\n";
$b = $a =~ tr/a-z/A-Z/;
print "$a\n";
$a =~ s/[A]/\@/g;
print "$a\n";
print "-----\n\n";
$c= "La farfalla e l'elefante";
print "$c\n";
$d = $c =~ tr/a,e/e,a/;
print "$c\n";
print "\nFine";

```

L'operatore split

Un classico operatore Perl simile a una funzione, che usa tra i suoi parametri una espressione regolare è split. Jeffrey Friedl definisce brillantemente questa funzione come "il contrario dell'operatore m/ ... /g valutato in contesto lista". Infatti, come l'operatore m/ ... /g valutato in contesto lista ritorna un array delle corrispondenze, l'operatore split ritorna in un array il testo che sta a sinistra e destra delle corrispondenze. Vediamo un esempio.

```

$a = "uno due tre quattro cinque";
@numeri = split(/\s+/, $a);
print "@numeri";

```

L'operatore salva in un array il testo separato da spazi. Dunque l'array @numeri conterrà il valore lista qw (uno due tre quattro cinque). Come si può notare, gli spazi, /\s+/, sono le corrispondenze verificate dalle espressioni regolari, e le stringhe "uno", "due", "tre", "quattro" e "cinque", elementi di @numeri, sono ciò che sta intorno alle corrispondenze.

L'operatore split ha la seguente sintassi generale:

```
split( espressione_regolare, variabile_target, numero_elementi);
```

dove:

espressione_regolare è l'espressione che se verificata crea le divisioni, se non specificata l'espressione di default è `^s+` la sequenza di spazi

variabile- target è il nome della variabile su cui applicare split se non specificata split viene applicato a `$_`

numero-elementi è il numero massimo di elementi in cui si vuole spezzare la stringa.

Con queste regole il nostro semplice esempio si riduce al banale:

```
$_ = "uno due tre quattro cinque";  
@numeri = split;  
print "@numeri";
```

Invece nel programma:

```
$testo = 'uno:due:tre:quattro:';  
@numeri = split(/:/,$testo,2);
```

la specifica di split scompone \$testo in due parti. Queste parti, ora elementi di @numeri, sono:

```
'uno' e 'due:tre:quattro:'
```

Un'altra caratteristica di split è quella degli elementi vuoti. Vediamo un esempio chiarificatore.

```
$tipi= 'tizio:caio::sempronio:';  
@ele = split(/:/,$tipi);
```

dopo split:

```
$ele[0] conterrà "tizio"  
$ele[1] conterrà "caio"  
$ele[2] conterrà la stringa vuota, perché tra due corrispondenze di ":" non  
c'è nulla  
$ele[3] conterrà sempronio"  
$ele[4] non è definito perché gli elementi vuoti in coda alla stringa sono  
ignorati
```

La funzione join

Se l'operatore split serviva per spezzare una stringa in un array di parti componenti la stringa, la funzione join serve per rincollare gli elementi di un array con un simbolo che fa da colla. Vediamone alcuni esempi:

```
$a='-';  
@b=('uno', 'due', 'tre');  
$incollaggio=join($a, @b);  
$a='-';  
@b=qw(uno due tre);  
$incollaggio=join($a, @b);  
@b=qw(uno due tre);  
$incollaggio=join('colla', @b);
```

Esercizi

Contare quante vocali ci sono in una stringa alfanumerica.

Contare le parole di una stringa alfanumerica.

Data la stringa: <finché la barca va' lasciala andare> trasformarla in <la barca lasciala andare finché va'>

Lezioni pratiche sulla programmazione in Perl

Lezione 24

File

Oramai avremmo dovuto raggiungere un buon grado di preparazione e dovremmo essere in grado di poter programmare delle applicazioni abbastanza interessanti, vediamo quindi come applicare le nostre capacità all'integrazione con file o cartelle. Partiamo subito con un esempio semplice semplice per introdurre i numerosi costrutti, per la manipolazione dei file, che il Perl mette a disposizione del programmatore. Tutto ciò che ci occorre sono due file di testo che chiameremo uno.txt e due.txt. nel file uno.txt inserirete a piacere alcune frasi del tipo:

```
Ma come son contento di poter
interagire con i file esterni, ora sì
che il Perl comincia a piacermi
sul serio.
```

mentre il file due.txt potete lasciarlo benissimo in bianco. Vediamo ora di scrivere un programma capace di leggere il file uo..txt e copiarne il contenuto in due.txt. Chiameremo tale file copia.pl

```
$uno='c:\prove\uno.txt';
$due='c:\prove\due.txt';
# come prima cosa individuimo la posizione
# dei file, nel nostro caso ammetteremo di
# aver salvato i file di testo nella cartella
# c:\prove, se il programma copia.pl si trova
# anch'esso nella stessa cartella basta
# indicare soltanto il nome del file
open(a, "<$uno") or die "Non apre uno.txt";
# con l'istruzione open chiediamo di aprire
# il file valorizzato in $uno che identificheremo
# un nome a piacere, in questo caso 'a'
# il carattere '<' indica che il file è di
# sola lettura, ma è anche possibile ometterlo
# in quanto se non specificato, il programma
# lo inserisce di default, gli operatori
# 'or' e 'die' sono molto importanti e ci
# permettono di uscire dal programma nel caso
# non si riesca ad individuare il file da
# cercare, restituendoci una frase a piacere.
open(b, ">$due") or die "Non apre due.txt";
# questa istruzione è simile alla precedente
# ma il carattere '>' indica che il file
# è di sola scrittura
while($riga=<a>){
print b $riga;
}
# con l'istruzione while, chiediamo
# di stampare tutte le stringhe contenute
# in 'a' in 'b', dove 'a' identifica $uno,
# e 'b' identifica $due
close(a);
close(b);
# Fatte le nostre modifiche chiediamo di
# chiudere i file in modo da salvare le
# modifiche, per verificare che i file
# vengano chiusi correttamente, sarebbe
# opportuno come abbiamo fatto per open
# utilizzare le funzioni 'or' e 'die'
```

Come potrete notare se andate ad eseguire il programma la funzione print del ciclo while non corrisponde più al video ma al file

due.txt, se andate a verificare troverete in esso il contenuto di uno.txt. Bello vero..? ..ma possiamo fare di più, immaginiamo di voler codificare un file di testo in modo da non farlo leggere ad occhi indiscreti, riprendiamo l'esempio precedente ed aggiungiamo al ciclo while una istruzione di traduzione:

```

$uno='c:\prove\uno.txt';
$due='c:\prove\due.txt';
open(a, "<$uno") or die "Non apre uno.txt";
open(b, ">$due") or die "Non apre due.txt";
while($riga=<a>){
$riga =~ tr/[a-z]/[zvutrsrqponmlihgfedcba]/;
# in questo modo potremmo modificare l'insieme
# dei carattere alfabetici dalla 'a' alla 'z',
# nell'insieme di caratteri opposti, cioè dalla
# 'z' alla 'a', provate a leggere il file due.txt
# se ci riuscite, scegliete quindi un ordine a
# piacere, e vedrete che sarà un buon metodo di
# codifica, per la decodifica, scrivete il programma
# inverso e cioè legge due.txt e scrive che so tre.txt
# utilizzate l'istruzione traduci in senso opposto
# $riga =~ tr/[zvutrsrqponmlihgfedcba]/[a-z]/;
# e come per magia eccovi restituito nuovamente
# il documento in forma comprensibile, Che dite ora...
# ...è bello?
print b $riga;
}
close(a);
close(b);

```

Con questo primo esempio sull'uso dei file abbiamo potuto anticipare molte cose sia sul loro utilizzo, sulla semplicità, ma soprattutto le potenzialità. Vediamo ora di prendere in esame alcuni dei principali concetti legati alla gestione dei file:

L'apertura di un file avviene per mezzo della funzione open. Esistono molti modi in cui un file può essere aperto, i principali sono i seguenti.

Sola lettura

```
open( A, '<nome_file' ) oppure open( A, 'nome_file' )
```

Il file può essere letto ma non modificato.

Sola scrittura

```
open(A, '>nome_file')
```

Il file è creato ex novo e può essere solo modificato, ma non letto.

Append

```
open(A, '>>nome-file')
```

Se il file non esiste viene creato, se esiste o se viene creato si possono aggiungere righe in coda al file, ma comunque non può essere letto.

Lettura e scrittura

```
open(A, '+<nome file')
```

La modalità di apertura è la stessa di lettura ma si può anche scrivere.

Scrittura e lettura

```
open(A, '+>nome-file')
```

La modalità di apertura è la stessa di scrittura ma si può anche leggere.

Append e lettura

```
open(A, '+>>nome-file')
```

La modalità di apertura è la stessa di append ma si può anche leggere.

Per poter scrivere su un file aperto in lettura e scrittura, o leggere su uno aperto in scrittura e lettura, occorre muovere il cursore del file per posizionarsi in fondo e in cima rispettivamente. Il cursore di un file è un indice della posizione attuale all'interno del file, che viene automaticamente aggiornato con la lettura, o esplicitamente spostato con la funzione seek. La funzione seek ha la seguente sintassi:

```
seek(descrittore, numero-di_byte, riferimento)
```

Dove:

descrittore è il descrittore del file.

numero-di-byte numero di byte di cui ci si vuole spostare nel file a partire dal riferimento.

riferimento vale 0 per inizio file, 1 se siamo nel mezzo del file (nel qual caso numero_di_byte può essere espresso da un numero negativo), 2 per la fine del file.

Vediamo un esempio.

```
$uno='c:\prove\uno.txt';  
open(a,'<uno.txt') or die "non riesco ad aprire il file";  
seek(a,0,2); # posiziona il cursore in fondo al file  
print a "\n     Giordano Costantini"; #firma il file  
close(a);
```

Il file uno.txt è aperto in lettura e scrittura. Se il file non esiste allora è eseguita la funzione 'die' . Una procedura analoga può essere seguita per leggere un file aperto in scrittura e lettura:

```
$aaa='c:\prove\tre.txt';  
open(a, '+>tre.txt') or die "non trovo il file";  
print a "Come son contento\n";  
print a "finalmente sto imparando molte cose\n";  
print a "sul Perl e la sua semplicità d'impiego";  
seek(a, 0, 0); # posiziona il cursore all'inizio del file  
print while<a>; # visualizza il file  
close(a);
```

Dopo aver ricreato il file e aggiunte tre righe. Il cursore si posiziona in fondo al file, ma per mezzo di ' seek(D, 0, 0);' si riposiziona all'inizio e poi si rilegge tutto.

Lezioni pratiche sulla programmazione in Perl

Lezione 25

Funzione Globbing

Oggi introdurremo qualche stuzzicante funzione del linguaggio Perl e l'interazione con i file esterni, cominciando con l'illustrare il modo di fare il globbing dei file, vediamone subito un esempio pratico:

```
@a = <*.txt>;  
print "@a\n";
```

con questa istruzione nell'array @a sono memorizzati tutti i nomi di file .txt della directory corrente. L'operazione di globbing può essere effettuata anche in un contesto scalare, in modo da estrarre un nome di file alla volta:

```
print while<*.txt>;
```

Come potete notare, se andate ad eseguire il programma, vi saranno restituite tutti i nomi dei file, proprio come se fosse un mini motore di ricerca, potrete inoltre utilizzare tale memorizzazione di array per il completamento di script più sofisticati man mano che ne sentirete l'esigenza. Prestate comunque un po' di attenzione alle seguenti particolarità: l'uso del metacarattere '*' nella descrizione dell'insieme di file non ha nulla a che vedere con l'analogo simbolo impiegato nelle espressioni regolari;

l'operatore angolare quando è applicato a qualcosa di diverso da un descrittore di file esegue il globbing e, di fatto, si trasforma in un operatore diverso da quello che sappiamo utilizzare per leggere le righe di un file.

Soprattutto per quest'ultimo motivo è stato introdotto un operatore specifico di globbing detto glob:

```
@a = glob("*.txt");
```

oppure

```
print while glob "*.txt";
```

L'operatore glob effettua anche interpolazione di variabili:

```
$a="*.txt ";  
@a =glob $a;  
print "@a\n";
```

File binari

I file binari sono quelli che possono contenere caratteri ASCII non "printable", ovvero possono contenere qualsiasi combinazione di caratteri ASCII da 0 a 255. Questa definizione di file binario è artificiosa e i sistemi operativi più moderni come concezione di file system non distinguono tra file binari e non binari. Altri sistemi operativi (Windows 95/98/NT) invece effettuano questa distinzione e richiedono dunque un trattamento speciale per i file binari.

Ad esempio, se si volesse eseguire una lettura su un file binario, come una immagine, sotto Windows dovremmo seguire la seguente procedura

```
open(GIF, 'pippo.gif') or die "Non riesco ad aprire pippo";  
binmode(GIF);  
read(GIF, $buf, 1024);
```

Con questa procedura il file binario viene aperto con open secondo le modalità che conosciamo, successivamente il descrittore creato con la open viene passato alla funzione binmode la cui sintassi generale è 'binmode(Descrittore)' la quale avverte il sistema operativo dicendogli che quel particolare descrittore è associato a un file binario. Essendo un file binario dobbiamo usare poi la funzione read invece del classico operatore angolare. Infatti l'operatore angolare prende una stringa di caratteri fino al prossimo \n, includendo anche \n. In un file binario il carattere \n potrebbe anche non essere presente e comunque si ha bisogno di leggerlo a gruppi di byte la funzione read serve proprio a questo. Vediamone ora più in dettaglio la sintassi generale.

```
read(descrittore, variabile-buffer, numero_byte);
```

dove, il 'descrittore' è il descrittore di file da cui leggere, la 'variabile - buffer' è la variabile a cui assegnare i prossimi byte letti, mentre 'numero-byte' stabilisce quanti byte alla volta leggeremo con la byte.

Vediamo di illustrare un esempio su come poter creare una copia, di 100 byte per volta di un file binario, per questo esempio supporremo di avere nella nostra directory C:\prove un file del tipo binario, ad esempio ligabue.mp3 e di voler copiare il brano in un nuovo file Battiato.mp3, vediamo ora come impostare il programma:

```
open(a, 'c:/prove/ligabue.mp3') or die 'Non riesco ad aprire il file';
open(b, '>c:/prove/Battiato.mp3');
binmode(a);
binmode(b);
while(read(a, $buf, 100)){
print b $buf;
}
```

Si osservi come la funzione read nella condizione del while termini il ciclo quando si è giunti alla fine del file. Si osservi inoltre l'uso della print. La funzione di scrittura reciproca della read è semplicemente la print. Quest'ultima dovendo scrivere quello che è memorizzato nel buffer, sa quanti byte dovrà scrivere e quindi non ha bisogno che sia specificato il numero di byte, come avviene invece per la read.

Test sui file

Il Perl mette a disposizione una serie di operatori da applicare o al nome di un file o a un descrittore di file allo scopo di verificare se un certo fatto è vero o falso. Con l'esempio

```
open(a, 'c:/prove/ligabue.mp3') or die 'Non riesco ad aprire il file';
if(-e a){
print "Il file esiste\n";
}
else{
print "Non esiste il file\n";
}
```

Oppure

```
if(-e "c:/prove/ligabue.mp3"){
print "Il file esiste\n";
}
else{
print "Non esiste il file\n";
}
```

dimostriamo l'uso dell'operatore -e, il quale verifica se il file esista o meno.

I test sui file ritornano il valore numerico 1 se la verifica ha esito positivo e " ", nullo, se ha esito negativo. Nel caso in cui non venga applicato un argomento, automaticamente l'operatore assume come argomento il contenuto di \$_, con l'eccezione dell'operatore -t che assume come default STDIN.

Riportiamo in Tabella gli operatori di test indicando quelli che sono o meno disponibili sotto Windows.

Operatore di test	Significato	Windows (s/n)
-r	File leggibile da uid/gid effettivo	s
-w	File scrivibile da uid/gid effettivo	s
-x	File eseguibile da uid/gid effettivo	s
-o	File posseduto da uid effettivo	n
-R	File leggibile da uid/gid reale	n
-W	File scrivibile da uid/gid reale	n
-X	File eseguibile da uid/gid reale	n
-O	File posseduto da uid reale	n
-e	File esiste	s
-z	File ha dimensione zero	s
-s	Dimensione del file	s
-f	File normale	s
-d	Directory	s
-l	Link simbolico	n
-p	Pipe con nome	n
-S	Socket	n
-b	File speciale block	n
-c	File speciale carattere	n
-t	Descrittore aperto su tty	n
-u	Il bit setuid è alto	n
-g	Il bit setgid è alto	n
-k	Il bit sticky è alto	n
-T	File di testo	s
-B	File binario	s
-M	Età in giorni dall'ultima modifica al file	s
-A	Età in giorni dall'ultimo accesso	s
-C	Età in giorni dall'ultima modifica di inode	s

L'età di un file è sempre calcolata prendendo come riferimento l'istante in cui è stato lanciato il programma che contiene il test sull'età. Questo significa che se il programma che contiene il test verifica l'età in giorni di una modifica che è avvenuta dopo che è stato lanciato il programma, può ritornare anche un valore negativo.

Con questi operatori di test si possono sapere tante cose di un file, ma non tutto. Se si vuole proprio il quadro completo conviene usare la funzione stat.

La funzione stat si applica nello stesso modo in cui si applicano gli operatori di test, ma ritorna ben 13 argomenti:

```
@a =stat "C:\prove\Battiato.mp3";  
print "@a\n";
```

In questo modo è possibile sapere vita morte e miracoli del file processato. Il significato dei valori restituiti corrisponde ai seguenti significati

Valore	Descrizione Unix	Descrizione Win32
\$a[0]	Numero dispositivo	numero dei drive
\$a[1]	Numero di inode	zero
\$a[2]	Modo dei file: read/write/execute	
\$a[3]	Numero di link al file	1
\$a[4]	User ID	0
\$a[5]	Group ID	0
\$a[6]	Identificatore dispositivo	numero drive
\$a[7]	Dimensione file in byte	Dimensione file in byte
\$a[8]	Ultimo istante di accesso	Ultimo istante di accesso
\$a[9]	Ultimo istante di modifica	Ultimo istante di modifica
\$a[10]	Ultimo istante di creazione	Ultimo istante di creazione
\$a[11]	Dimensione dei blocco dei disco	0
\$a[12]	Numero di blocchi per file	0

Gestione dei file

Un file, oltre a essere aperto, chiuso, letto, scritto e verificato, può subire altri maltrattamenti. Per esempio può essere:

- rimosso;
- rinominato;
- modificato nei permessi.

Rimozione di un file

Per rimuovere un file da Perl si può usare la funzione unlink:

```
unlink "C:\prove\Battiato.mp3"; #Cancella il file processato  
unlink <*.txt>; # eliminati tutti i file *.txt
```

Come si deduce dagli esempi, si può rimuovere uno o più file e anche rimuovere file facendo globbing. La funzione unlink ritorna il numero di file che sono stati rimossi con successo.

Rinomina di un file

Per cambiare nome a un file da Perl si può usare la funzione rename. La sintassi generale della funzione rename è

```
rename(vecchio-nome_file, nuovo_nome_file);
```

dove per il nome del file, vecchio e nuovo, si seguono le convenzioni che abbiamo già visto; ad esempio

```
rename('uno.txt', 'ciao.html');
```

ribattezza uno.txt in ciao.html. Il file ciao.html si trova nella directory corrente;

```
rename('c:/a/ciao.html','c:/a/b/uno.txt');
```

ribattezza ciao.html in uno.txt. Spostandolo nella sottodirectory b di a, che deve già esistere, altrimenti tale processo fallirà.

Se la funzione rename porta a termine la sua funzione allora ritorna il valore 1, ma se fallisce ritorna la stringa nulla " ". Se poi il nuovo nome di file corrisponde al nome di un altro file, con rename quest'ultimo file viene sovrascritto e quindi distrutto. Superfluo dunque anche questa volta raccomandare la massima cautela (e magari il test -e).

Modifica dei permessi

I permessi stabiliscono chi è che può fare che cosa su un file. In Perl per modificare i permessi si usa la funzione chmod. La funzione chmod è sintatticamente analoga alla funzione unlink,

```
chmod(mode, lista file);
```

Per calcolare i valori mode da implementare su i file si devono seguire le seguenti istruzioni ritornando agli operatori di test precedentemente indicati, si associa ad ogni operatore un valore nel caso non si volesse associare nessun operatore '-' il suo valore è uguale a zero, vediamo ora quali sono i valori: r=4,w=2,x=1 e -=0.

Ora dobbiamo stabilire i permessi da indicare per l'*user* cioè te, il *group*, il gruppo particolareggiato e tutti gli altri *others*.

Vediamo quindi un esempio

user			group			others		
r	w	x	r	-	-	r	-	-
4	2	1	4	0	0	4	0	0
7			4			4		

in questo caso quindi il chmod dei permessi vale 744, altri valori tipici di mode sono:

666 per indicare permesso read/write

444 per indicare permesso di sola lettura

Ad esempio.

```
chmod(744, 'uno.txt', 'due.txt', 'tre.txt');
```

La funzione chmod ritorna il numero di volte che è riuscita a cambiare i permessi, quindi nell'esempio può ritornare 0, 1, 2 o 3. Se non è specificata alcuna lista di file prende come nome di file il contenuto di \$_

Directory

Le directory sono file particolari che hanno come contenuto il nome di altri file e dunque anche di directory. Anche le directory possono essere manipolate, vediamo come:

- cambiare directory;
- aprire una directory;
- chiudere una directory;
- leggere una directory;
- creare una directory;
- rimuovere una directory.

Cambio directory

Per modificare la directory corrente in una directory diversa da quella in cui è lanciato lo script (il processo) Perl usa la funzione chdir:

```
chdir(nuova_directory);
```

Se riesce a cambiare directory ritorna vero, altrimenti falso. Se non si indica l'argomento nuova-directory allora la directory corrente diventa la home directory.

Aprire una directory

Per aprire una directory si usa la funzione opendir. La sintassi generale è:

```
opendir(descrittore_directory, nome_directory);
```

Ad esempio:

```
opendir(prove, 'c:\prove\');
```

Chiudere una directory

Per chiudere una directory si usa la funzione closedir, la cui sintassi generale è:

```
closedir(descrittore_directory);
```

ad esempio

```
closedir(prove);
```

Come per close, una opendir chiude anche una opendir che la precede.

Leggere una directory

Per leggere una directory si usa la funzione readdir, la cui sintassi generale è:

```
readdir(descrittore_directory)
```

Se la funzione è valutata in un contesto scalare ritorna il prossimo nome di file, ma solo la base del nome senza specificare il percorso; se valutata in contesto lista ritorna, in un ordine apparentemente casuale, i nomi base di tutti i file nella directory:

```
$dir = 'c:\prove\';  
opendir(prove, $dir) or die "Non riesco ad aprire $dir";  
while($file_base = readdir(prove)){  
  print "$file_base\n";  
}  
closedir(prove);
```

legge la directory un nome di file alla volta. Oppure

```
$dir = 'c:\prove\';  
opendir(prove, $dir) or die "Non riesco ad aprire $dir";  
@nomi_file = readdir(prove);  
print "@nomi-file\n";  
closedir(prove);
```

dà lo stesso risultato ma in contesto lista @nomi_file contiene tutti i nomi di file nella directory.

Creare una directory

Per creare una directory si usa la funzione mkdir, la cui sintassi generale è:

```
mkdir(nome_file, modo);
```

Ad esempio:

```
mkdir('prove2', 777);
```

mkdir() ritorna 1 se ha successo e "" se fallisce.

Rimuovere una directory

Per rimuovere una directory si usa la funzione rmdir, la cui sintassi generale è:

```
rmdir(nome_file);
```

rmdir rimuove la directory nome_file se la directory è vuota, ritorna 1 se ha successo e " " se fallisce.

Ad esempio:

```
rmdir('prove2');
```

rimuove la directory prove2 ma solo se prove2 non contiene nulla.

Lezioni pratiche sulla programmazione in Perl

Lezione 26

File DBM

Una delle caratteristiche più interessanti offerte dal Perl scripting è la possibilità di creare e gestire tabelle hash permanenti, ovvero database che risiedono permanentemente su file speciali detti file DBM. I file DBM vengono gestiti allo stesso identico modo con cui vengono manipolate le variabili hash in memoria volante.

L'attento programmatore preferisce impiegare gli hash DBM anche quando si abbia a disposizione un Data Base Management System come Sybase, MySQL, Informix, Oracle o SQL server etc etc, in quanto oltre a essere estremamente semplici da utilizzare, hanno soprattutto prestazioni in termini di tempo d'accesso eccezionali e sono per questo motivo lo strumento ideale per risolvere problemi di prestazione.

Vedimo subito la sintassi su come poter associare un file DBM a una variabile hash:

```
dbmopen(%HASH, "nome-file-dbm", MODO);
```

dove

%HASH è il nome di un hash a vostra scelta
nome-file-dbm è il nome che volete dare al vostro file DBM
mode è un codice numerico, di solito posto a 0644

Per chiudere un file DBM si usa invece:

```
dbmclose(%HASH);
```

Ad esempio:

```
dbmopen(%nomehash, "pippo", 0777) or die "non riesco ad aprire il file";
$nomehash{"Estate"}="Caldo";
$nomehash{"Mare"}='refrigerio';
while(($key, $value)= each(%nomehash)){
print "$key $nomehash{$key}\n";
}
dbmclose(%nomehash);
```

In questo esempio si è definita una nuova variabile hash 'nomehash' e si chiede di salvare i nuovi elementi nel file pippo all'interno della stessa directory del programma Perl che lo invoca, si è inoltre impostato i permessi su 777 ma si sarebbero potuti scegliere anche un 755 o un 666 non sarebbe cambiato nulla di rilevante, ora se andate ad eseguire il programma saranno generate due file binari contenenti le informazioni generate dalla variabile hash e cioè 'pippo.pag' e 'pippo.dir', ovviamente si possono utilizzare tutte le operazioni eseguibili in una normalissima hash, e cioè aggiungere nuove chiavi e valori, cancellare chiavi oppure ricercare un valore in base ad una chiave e viceversa, questo esempio è molto semplice, ma se provate a modificarlo, dando agli utenti esterni la possibilità di poter aggiungere valori mediante la funzione STDIN, potrete notare aprendo il file pippo.pag con NotePad che le nuove chiavi valori vengono aggiunte al database anzicchè sovrascrivere i vecchi dati, pur facendo eseguire diverse volte il programma.

Esercizio

Scrivere un programma che richieda all'utente mediante funzioni STDIN l'inserimento, modifica, cancellazione e ricerca di un file DBM in cui le chiavi sono nomi dei loro amici ed i valori sono i loro indirizzi E-mail.