Perl

Nicola Gessa

Introduzione al Perl

- Il Perl (Pratical Estraction Report Language) è stato creato da Larry Wall verso la fine degli anni 80
- E' un linguaggio interpretato mirato alla manipolazione di stringhe e file di testo
- E' uno dei linguaggi più utilizzati nella scrittura di script CGI
- Sue caratteristiche sono compattezza degli script e potenza
- Non è indicato per sviluppare programmi ad elevata velocita di esecuzione

Per iniziare

- Ogni script Perl deve essere lanciato attraverso l'interprete per poter essere eseguito
- Esempio di creazione di uno script:

  creare un file **test.pl** inserendo la riga

  print "Ciao mondo!";
- Per eseguire lo script possiamo fare in due modi:
  - lanciare il comando >**perl test.pl**, oppure
  - sotto Unix inserire nel file di testo come prima riga

    #!usr/local/bin/Perl

    rendere il file eseguibile

    eseguire il file **>test.pl**
- I commenti in Perl iniziano con il carattere **#**

Variabili in Perl

- Non è necessario dichiarare le variabili prima di utilizzarle
- Nella sintassi per l'uso delle variabili il Perl distingue tra 3 tipi di variabili: scalari, array e array associativi.
- Il Perl interpreta le operazioni e i tipi delle variabili in base al contesto in cui sono poste. Ci sono tre contesti principali: stringa, numerico e array.
- Uno scalare è interpretato come TRUE (1) se non è la stringa nulla né 0.
- I **riferimenti** alle variabili scalari iniziano sempre con **$**, anche se si riferiscono a uno scalare componente di un array, gli array iniziano con **@** e gli array associativi con **%**.

Variabili in Perl

Perl è **case sensitive** ($a<>$A)

Variabili scalari: contengono sia stringhe che numeri

$nome = ‘Marco’;

$n=3;

$pigreco=3.14;

Variabili array

@nomi=(‘Marco’,’Michele’,’Paolo’);

@nomi=(Marco,Michele,13);

Il primo elemento di un array ha indice 0

Per accedere agli elementi dell’array si usano le variabili scalari:

$nomi[1] prende il valore Michele

Operazioni e assegnamento

$a=1+2;	addizione
$a=3-4;	sottrazione
$a=3*4;	moltiplicazione
$a=8/2;	divisione
$a= 2**4;	elevamento a potenza
$a=5%2;	modulo
$a++;++$a;	incremento
$a+=$b; $a-= $b; $a.=$b	assegnamento
$a=$b.$c;	concatenazione di stringhe
$a=$b x $c	ripetizione di stringhe (non commutativa:”stringa “ x “numero”)

Interpolazione

Il Perl consente di forzare o meno l'interpretazione dei nomi delle variabili all'interno delle stringhe

- Una coppia di apici singoli viene usata per delimitare una stringa che non deve essere interpretata
- I doppi apici sono utilizzati per delimitare una stringa che deve essere interpretata
- Il carattere di escape per rappresentare caratteri speciali all'interno di stringhe interpolate è '\'

Es.

```
$a=1;
$b=2;
print '$a+$b\n';          stampa> $a+$b\n
print "$a+$b\n";          stampa> 3
print "\$a+\$b\\n";       stampa> $a+$b\n
```

Variabili in Perl - array

- **Assegnamento di array**

  @nomi=('Marco','Sara');

  @nomi2=('Luca',@nomi); nomi2>Luca,Marco,Sara
- **Inserimento di elementi**

  push(@nomi,"Carlo"); nomi>Marco,Sara,Carlo

  push(@nomi2,@nomi);
- **Estrazione**

  $persona=pop(@nomi); $persona='Carlo';nomi>Marco,Sara
- **Calcolo della lunghezza di un array**

  $len = @nomi $len= 2
- **Conversione in stringa**

  $len = "@nomi" $len='Marco Sara'
- **Ultimo indice dell'array**

  $index=$#nomi $index=1

Variabili Perl - uso del contesto

```
$var1 = 1;          $var2 = 2;
$var3 = "pippo";    $var4 = "pluto";

$tot1=$var1+$var2;
$tot2=$var1.$var2;
print "var1+var2 = $tot1\n";        >var1+var2 = 3
print "var1\.var2 = $tot2\n";       >var1.var2 = 12

$tot1=$var3+$var4;
$tot2=$var3.$var4;
print "var3+var4 = $tot1\n";        >var3+var4 = 0
print "var3\.var4 = $tot2\n";       >var3.var4 = pippopluto

$tot1=$var3+1;
print "var3+1 = $tot1\n";           >var3+1 = 1
```

Variabili Perl - uso del contesto

```
@tmp=(Marco,Paolo);
print @tmp;              >MarcoPaolo

print "@tmp";            >Marco Paolo

print @tmp."";           >2
```

Variabili in Perl - array associativi

Mentre un normale array ci consente di “puntare” ai suoi elementi usando numeri interi, un array associativo fa la stessa cosa usando stringhe (chiavi) come indici. Si mantiene quindi una associazione chiave-valore (come in una tabella hash).

%squadre = (‘Inter’, Milano,’Juventus’,Torino,’Spal’,Ferrara);

print “ La Spal è la squadra di **$squadre{Spal}**;

- I valori negli array associativi possono essere associati solo a coppie.
- Usando le funzioni **key()** e **values()** si possono risalire dalle chiavi ai valori.
- Gli array associativi possono essere convertiti in array e viceversa

Variabili in Perl

- La funzione keys restituisce la lista delle chiavi dell'array associativo

```
foreach $squadra (keys %squadre){
        print $squadra;
}
```

- La funzione values restituisce la lista dei valori dell'array associativo

```
foreach $citta(values %squadre){
        print $citta;
}
```

- La funzione each restituisce la coppia

```
while (($squadra, $citta)=each(%squadre)){
        print " La $squadra e' la squadra di $citta\n";
}
```

Strutture di controllo

Operatori di confronto

- Il Perl mette a disposizione 3 tipi di operatori di confronto, ognuno dei quali va applicato su un determinato tipo di dato: numeri, stringhe e file
- Il Perl effettua una conversione di tipo delle variabili prima di effettuare il confronto coerentemente col tipo dei dati conivolti

Operatori logici

Gli operatori logici del Perl per le espressioni booleane sono

&&	**AND**
||	**OR**
!	**NOT**

Operatori di confronto

- Tra numeri:

$a==$b	uguaglianza
$a<$b	minore (maggiore >)
$a<=$b	minore(maggiore >) o uguale
$a!= $b	diverso

- Tra stringhe il minore - maggiore segue l'ordine alfabetico

$a eq $b	uguaglianza
$a lt $b	minore (maggiore gt)
$a le $b	minore(maggiore ge) o uguale
$ ne$b	diverso

Operatori di confronto

- Tra file

  $a = “myfile.txt”

  -r $a leggibile

  -w $a scrivibile

  -d $a è directory

  -f $a file regolare (non device)

  -T $a file di testo

  -e $a esiste

Costrutto if - else

Permette di condizionare il flusso del programma in base a determinate condizioni

- **if** (*condizione*) { *istruzioni*...}

- **if** (*condizione*) { *istruzioni*...}
  else{….}

- **if** (*condizione*) { *istruzioni*...}
  elsif { *istruzioni*...}

istruzione_singola **if** *condizione*

Istruzioni come valori

In Perl l'esecuzione di ogni istruzione restituisce un valore di vero o falso:

print print "pippo"; > pippo1

quindi si possono utilizzare nelle espressioni booleane.

Il Perl interrompe la valutazione di una espressione logica appena è in grado di stabilirne il valore:

A && B l'espressione B verrà valutata solo se A è vera

A || B l'espressione B non verrà valutata se A è vera

Istruzioni di ciclo: while- for

- Il ciclo **while** consente di ripetere un certo blocco di istruizioni finchè una certa condizione è vera

  while(*condizione*){*istruzioni*..}

- Il ciclo **for** consente di ripetere un certo blocco di istruizioni un numero prefissato di volte grazie ad un contatore

  for(*condizione iniziale; condizione finale; incremento*){

  istruzioni….}

- **foreach** consente di inserire in una variabile scalare tutti gli elementi di una lista

  foreach *variabile_scalare* (*array*){*istruzioni*….}

- **next** forza il Perl a saltare l'esecuzione delle restanti operazioni e a riprendere da un nuovo ciclo
- **last** forza l'uscita da un ciclo riprendendo il programma dalla prima istruzione dopo il blocco

Input/Output su file

- I canali standard di comunicazione di default sono 3, a cui sono associati degli handler
  - <STDIN> standard di input
  - <STDOUT> standard output
  - <STDERR> standard error
- Per eseguire operazioni di lettura e scrittura su file si deve aprire un canale di comunicazione dichiarandone l'uso che si intende fare
  - **open**(*handle*,"<*nome*") lettura
  - **open**(*handle*,">*nome*") scrittura
  - **open**(*handle*,">>*nome*") append
- La lettura viene eseguita con l'operatore <..> e la scrittura usando il comando **print** *handle.*
- Infine il comando **close**(*handle*) chiude il file

Input/Output su file - esempi

Es 1:

```
$filename =<STDIN>;
open(FILE,"< $filename");
while(!eof(FILE)){
        $riga=<FILE>;          #leggo una riga dal file
        print $riga;}
```

Es 2:

```
$filename =$ARGV[0];
open(FILE,"< $filename”) || die “errore!\n\n”;
while($riga=<FILE>){
        print $riga;
}
```

Input/Output su file - esempi

Es 4:

```
$filename ="prova";
open(FILE,"> $filename") || die "errore!\n\n";
$riga=<STDIN>;
while($riga ne "\n"){
        print FILE $riga;
        $riga=<STDIN>;}
close FILE;
```

Es 3:

```
$filename =“prova”;
open(FILE,"< $filename");
@lines = <FILE>;
print “@lines”;
close FILE;
```

Variabili speciali

Perl dispone di un insieme ampio di variabili speciali, variabili gestite direttamente dall'interprete per contenere determinati parametri:

- Parametri da riga di comado: sono contenuti nell'array **@ARGV**, **@#** contiene il numero dei parametri
- Variabili d'ambiente: sono contenute nell'array associativo **%ENV**

  %ENV{path}
- Variabili speciali per il pattern matching

  $1,$2,…$&
- Variabili per il controllo dei processi

  $!,$0,$@

Uso delle espressioni regolari

- Una delle caratteristiche del Perl è quella di poter operare in maniera flessibile sulle stringhe di caratteri.
- Perl consente di descrivere in modo generico pattern (espressione regolare) di caratteri, ossia schemi di costruzione di stringhe, per effettuare confronti e sostituzioni.
- Una espressione regolare è una espressione che descrive uno schema di stringa
- Il Perl definisce una sintassi precisa per identificare i possibili componenti di una stringa

Es. **\d\d.+**

descrive una stringa che inizia necessariamente con due cifre e poi prosegue con una parola di una qualunque lunghezza composta da qualunque carattere

33p84_jdf soddisfa questa espressione, **a2y** no

Uso delle espressioni regolari

Nell'uso delle espressioni regolari si può utilizzare un vasto insieme di caratteri speciali, ed è questo che rende le espressioni regolari potenti.

Alcuni caratteri speciali

.	Qualunque carattere eccetto il newline
^	L'inizio di una stringa
$	La fine di una stringa
*****	L'ultimo carattere ripetuto 0 o più volte
+	L'ultimo carattere ripetuto almeno una volta
?	L'ultimo carattere presente 0 o una volta

Le parentesi quadre sono usate per fornire un insieme preciso di caratteri

[abc]	Un carattere che si **a** o **b** o **c**
[a-z]	Tutti i caratteri fra **a** e **z**

Uso delle espressioni regolari

Altri caratteri speciali

\n	newline		
\t	tab		
\d	digit	**\D**	non digit
\w	carattere alfanumerico (uguale all'insieme **[a-zA-Z0-9]**)		
\W	carattere non alfanumerico		
\s	carattere whitespace (space, tab, newline)		

I caratteri speciali possono essere usati facendoli precedere dal carattere di escape "\". Es.

\[	rappresenta la parentesi quadra "**[**"
*****	rappresenta l'asterisco "*****"

Uso delle espressioni regolari

Esempi di espressioni regolari e stringhe che le soddisfano

prova	prova
p.ova	prova, pqova, p ova,p3ova,p:ova….
p\.ova	p.ova
p\d?ova	pova, p1ova, p2ova….
p[123]ova	p1ova,p2ova,p3ova
[a-c][ab]	aa,ab,ba,bb,ca,cb
adc*	ad,adc,adcc,adccc
[ac]*	“qualunque combinazione di a e c”
(pippo|pluto)	pippo, pluto
(1\+)+	1+,1+1+,1+1+1+…..

Espressioni regolari: confronto

- Pattern matching è l'operazione di verifica se una string **o una sottostringa** corrisponde (fa match) con un certo pattern, e quindi rispetta lo schema della espressione regolare
- In Perl l'operatore per tale verifica è

  m/espressione_regolare/

(la m può essere omessa) e il confronto si effettua usando il simbolo **=~**

prova =~ m/p.ova/ è verificata

Es. per controllare l'inserimento di un input possiamo fare

```
$in=<STDIN>
if($in =~/\d+/){
        print "hai inserito delle cifre"}
```

Espressioni regolari:sostituzione

- La sostituzione (pattern substitution) consente di cercare una certa sottostringa all'interno di una stringa e di sostituirla con un altro insieme di caratteri
- In Perl l'operatore per la sostituzione è

  s/*stringa_di_ricerca*/*stringa_da_sostituire*/

  si usando il simbolo **=~** per l'assegnazione
- La sostituzione avviene solo la prima volta che viene rilevata un'occorenza della stringa di ricerca.Per forzare la sostituzione di tutte le occorrenza della stringa di ricerca si aggiunge **g** (globally) per indicare che l'operazione va effettuata su tutte le occorrenze.

  Es. per sostituire in una stringa tutti numeri con delle X

```
$in=<STDIN>;
$in=~ s/\d/X/g;
```

Uso delle espressioni regolari

E’ possibilie nelle E.R. tenere in memoria alcune componenti dell’espressione stessa.

- Ogni termine racchiuso fra parentesi tonde viene memorizzato ed è riferibile usando le variabili speciali \1, \2, \3 ridefinite in seguito nel programma in $1, $2, $3….

Es. per verificare l’inserimento di una stringa di tipo “a=a” si puo’ scrivere

```
$in=<STDIN>;
if($in =~ /^(\w)=\1$/){
        print "hai una equazione\n";
}
else{ print 'non e\' una equazione\n';}
```

Uso delle espressioni regolari

- L'operatore **!~** è usato per verificare una stringa che NON soddisfa una espressione regolare

  Es **("pippo" !~ "pluto")** è vera

- La variabile **$_** è la variabile che contiene la stringa di default per le espressioni regolari quando non viene specificata
- L'opzione **/i** rende l'espressione regolare case insensitive
- E' possibile gestire l''**ingordigia**" del linguaggio nell'analisi delle espressioni regolari.Data la stringa

  AxxxCxxxxxxxxxxCxxxD, si puo' suddividere in due modi

  - **a.*c.*d** A xxxCxxxxxxxxxx C xxx D
  - **a.*?c.*d** A xxx CxxxxxxxxxxCxxx D

Uso dei puntatori

In Perl è possibile definire e utilizzare puntatori alle variabili, cioe' variabili che contengono gli indirizzi di memoria di altre variabili.

Per ricavare l'indirizzo di memoria di una variabile la si fa precedere dall'operatore **'\'**, per risalire alla variabile puntata si utilizza il doppio **$$**

Es.

```
$nome = 'Mario';          #definisce una variabile
$ind = \$nome;            #ora tmp contiene l'indirizzo di $nome

print "indirizzo di \$nome="."$ind\n";      >SCALAR(0x1796fbc)
print "valore di \$nome="."$$ind\n";        >Mario
```

Uso dei puntatori

Posso usare l'operatore -> per risalire al valore della variabile di un lista anonima

Esempio

```
$arrayref = [1, 2, ['a', 'b', 'c']];
# $arrayref contiene l'indirizzo della lista
print "$arrayref->[2][1]\n";          >b
print "$$arrayref[2][1]\n";           >b
```

In perl non c'e' limite alla definizione di puntatori di puntatori;

Esempio

```
$ops="ooops";
$refrefref = \\\$ops;
print $$$$refrefref."\n";
```

Uso dei puntatori

I puntatori possono essere definiti come entry point di funzioni

Esempio

```
$coderef = sub { print ”Salve!\n" };
&$coderef;        #esegue la funzione
```

Esempio

```
$coderef = sub { print shift };
&$coderef("ok!");          #stampa ok!
print "\n";
$coderef->("ok!");         #stampa ok!
print "\n";
```

Definizione di strutture

Perl consente di definire strutture (record) composte da tipi di dati diversi

Esempio

```
struct myrecord => {
        val1  => '$',
        val2 => '@',};

$rec = myrecord->new(
            val1=>'scalar',
            val2=>[1,2,3],
);
print "valore 1=".$rec->val1."\n";            #stampa> scalar
print "valore 2="."@{$rec->val2}"."\n";       #stampa> 1 2 3
print "valore 3="."@{$rec->val2}[2]"."\n"; #stampa> 3
```

Subroutine

- Si possono racchiudere blocchi di codice all'interno di *subroutine* in modo da poterli richiamare più volte nel programma.
- Le subroutine vengono così definite

  sub *nome _subroutine*{

  istruzioni….

  return *valore*;}

- E' possibile passare dei parametri alla subroutine al momento della chiamata. Tali parametri sono memorizzati dal Perl nell'array **@_**
- Per richiamare la subroutine si deve specificare il nome preceduto dal simbolo **&** seguito dai parametri contenuti tra parentesi e separati da virgola

  &*nome_subroutine*(*par1*,*par2*)

- Le subroutine possono essere definite nel file anche dopo la loro chiamata

Soubroutine

- Con la funzione **local** possono essere definite delle variabili locali che hanno validità solo all'interno della subroutine
- Le variabili non definite come **local** risultano variabili globali anche se definite dentro il blocco di una subroutine
- Per estrarre le i parametri dall'array si possono usare gli indici dell'array oppure l'operatore **shift** che li estrae di seguito a partire dal primo
- In mancanza di **return** il valore restituito dalla subroutine è l'ultimo calcolato

Subroutine

```
$p=&mysub (3,2);
print $p."\n";
sub mysub{
        $val1=@_[0];
        $val2=@_[1];
        print "\n\n\$val1 = $val1\n";
        print "\$val2 = $val2\n";
        $valore_finale=$val1+$val2;
        return $valore_finale;
}
print &mysub (3,2);
print $valore_finale;
```

Soubroutine

```
sub mysub{
        local($val1,$val2,$valore_finale);
        $val1=shift;
        $val2=shift;
        print "\n\n\$val1 = $val1\n";
        print "\$val2 = $val2\n";
        $valore_finale=$val1+$val2;
        return $valore_finale;
}
print &mysub (3,2);
print $valore_finale;       #non stampa niente
```

Passaggio per riferimento

Oltre al passaggio delle variabili per valore si può effettuare il passaggio delle variabili per riferimento

Esempio

```
sub duplica{
        local($tmp)=@_[0];
        print $tmp;
        $$tmp=($$tmp)*2;
}

$var=3;
&duplica(\$var);
print "var = $var\n";                #stampa >6
```

Librerie esterne

- Per rendere modular il codice, si può memorizzare un insieme di subroutine in un file separato e poi includerlo nello script che ne fa uso tramite il comando **require**, passandogli il nome dello script che si vuole includere

  require “C:/Perl/mylib.pl”

  require tmp::mylib cerca un file **.pm**

- Il Perl, se non riceve un path completo, va a cercare i file passati a un comando require nelle directory di default contenute nell’array **@INC**

- Per inserire una directory nell’array @INC si può usare la funzione **push** come solito: push(@INC,”/home/myhome/mylib”);

- Il file libreria che deve essere incluso deve avere come ultima linea

  1;

Librerie esterne - esempio

mylib.pl

```
sub mysub{
        istruzioni…….
        return $valore_finale2;
}
1;
```

myprog.pl

```
require "./mylib.pl";
print mysub(5,6);
```

```
#!/usr/bin/perl
....
print "Content-type:text/html\n";print "\n";
&ReadEnvironment;
print "<HTML>";....print "<FORM>";
@cgipairs =  split ("&", $temp=<STDIN>);
$i=0;
foreach $pair ( @cgipairs ){
        ($name, $value) = split ("=", $pair);
        $value =~ s/\+/ /g;
        $cgivar[$i] = "$value";
       $i=$i+1;}
print "<BODY>";
print "<p>Ho inserito </BR>";
&insert(@cgivar);
......print "</HTML>";
```

Link utili

- www.perl.org
- www.perl.com
- www.activestate.com
- www.cpan.org