Scripting Languages & Perl

UNIVERSITÀ DELLA CALABRIA

Francesco Pacenza

Prof. Giovambattista Ianni

Corso di **Sistemi Operativi e Reti** - *A.A. 2017/18*

Dipartimento di Matematica e Informatica

CS@CS

Documentazione consigliata:

[chromatic, 2015]
[Schwartz et al., 2016]
[perl5porters, 2017]

Dicembre, 2017

Introduzione

INTRODUZIONE

Introduzione

Perl:

- Linguaggio di programmazione/scripting generico → Linguaggio Interpretato[1]
- Originariamente usato per la manipolazione di testi
- Oggi impiegato per una ampia gamma di attività
  - System administration
  - Web development
  - Network programming
  - GUI development
  - ...

[1] Il codice sorgente viene tradotto da un interprete in linguaggio macchina al momento dell'esecuzione senza necessità di compilatore

Vantaggi & Svantaggi di Perl

Vantaggi

- Pratico
- Efficiente → Integra delle macro istruzioni già pronte per essere utilizzate
- Completo → Fornisce array, liste, mappe, RegExp, manipolazione di stringhe, pattern matching, ...

Svantaggi

- Minimale e conciso:

```
my @tripled = map { $_ * 3 } @numbers;
```

Espressività in Perl

- Imparare Perl è come imparare un qualsiasi linguaggio parlato
- **TIMTOWTDI** (si pronuncia “Tim Toady”) e significa *There’s more than one way to do it!* (esiste più di un modo per fare la stessa cosa!)
- L’espressività di Perl consente a chi ha poca esperienza di scrivere buoni programmi senza avere una completa conoscenza di tutto il linguaggio

Espressività - Triplicare una lista di interi

Novice Perl hacker

```
my @tripled;
for (my $i = 0; $i < scalar @numbers; $i++) {
  $tripled[$i] = $numbers[$i] * 3;
}
```

Perl adept

```
my @tripled;
for my $num (@numbers) {
  push @tripled, $num * 3;
}
```

Experienced Perl hacker

```
my @tripled = map { $_ * 3 } @numbers;
```

Perl Basics

PERL BASICS

Fondamenti di Perl

Iniziamo ad usare Perl

Verificare la versione e il funzionamento

```
perl -v
```

Output

```
This is perl 5, version 14, subversion 2 (v5.14.2) built
    for i486-linux-gnu-thread-multi-64int
(with 104 registered patches, see perl -V for more
    detail)

Copyright 1987-2011, Larry Wall
...
```

Primo script Perl

Hello World !

- Inseriamo il "path" corretto per l'interprete Perl, sulla prima riga del file:

```
#!/usr/bin/perl
```

- Utilizziamo la funzione "print" per stampare del testo su standard output:[a]

```
print "Hello World !";
```

[a] Notare la differenza tra l'uso di *"testo"* e l'uso di *'testo'*

Lanciamo lo script

Ci sono 2 modi per lanciare uno script Perl:

- Richiamando direttamente l'interprete Perl sullo script:

```
perl nome_file.pl
```

- Rendendo eseguibile il file ed eseguendolo successivamente:

```
chmod u+x nome_file.pl
./nome_file.pl
```

Primo script Perl

Consigli utili

- È possibile utilizzare l'opzione *-w* per ottenere più informazioni in caso di errori

```
perl -w nome_file.pl
```

- È possibile lanciare lo script in debug mode usando l'opzione *-d*

```
perl -d nome_file.pl
```

- È possibile includere strict e warnings per ricevere "suggerimenti" su possibili errori comuni durante la scrittura di script in Perl

```
#!/usr/bin/perl
use strict;
use warnings;
print "Hello World !";
```

Tipi di variabili in Perl

I nomi delle variabili sono sempre preceduti da un *identificatore* (un simbolo) che indica il tipo della variabile

Esempio

```
my $variabile_scalare = "Ciao";
my @array = (23, 42, 69);
my %array_associativo = (1  => "A", 2 => "B");
```

Nomi variabili

Tutti i **nomi di variabili/strutture dati** in Perl iniziano con una lettera o con un underscore "_". Successivamente possono includere qualsiasi combinazione di lettere, numeri e underscores.

Identificatori validi in Perl

```
my $name;
my @_private_names;
my %Names_to_Addresses;
sub function;
```

Identificatori NON validi in Perl

```
my $invalid name;    # gli spazi non sono ammessi nei nomi
my @3;               # non si inizia con un numero
my %~flags;          # i simboli non sono ammessi
```

Variabili e Valori I

Le variabili in Perl sono *contenitori* per dei valori **Values**:

Stringhe

```
my $name = 'Francesco, Giovanni';
my $address = "Corso Mazzini, Cosenza (CS),
    87100";
my $reminder = 'Laboratorio Lab 31b' . 'Lun.
    Mattina 8:30';
```

Numeri

```
my $integer = 42;
my $float = 0.007;
my $sci_float = 1.02e14;
my $binary = 0b101010;
my $octal = 052;
my $hex = 0x20;
```

Variabili e Valori II

Undef

```
my $name = undef; # non necessario
my $rank; # contiene anch'esso undef
```

List

```
my @fibonacci = (1, 1, 2, 3, 5, 8, 13);
```

Scalari (variabili scalari)

Contengono un singolo valore discreto, che può essere:

- string
- integer or floating-point
- filehandle

Sono preceduti dal simbolo **$**

Definizione e inizializzazione

```
my $value;

$value = 123.456;
print "$value\n";

$value = 77;
print "$value\n";

$value = "I am Chuck's big toe.";
print "$value\n";
```

Array I

- Contengono uno o più scalari
- Rappresentano una lista di valori
- La numerazione parte da 0 (sono *zero-indexed*)
- La variabile speciale *$#array* ritorna l'indice dell'ultimo elemento di un array

Sono preceduti dal simbolo @

Array II

Assegnamento (1)

```
my @gatti;
$gatti[3] = 'Jack';
$gatti[2] = 'Tuxedo';
$gatti[0] = 'Daisy';
$gatti[1] = 'Petunia';
$gatti[4] = 'Brad';
```

Assegnamento (2)

```
my @gatti = ('Daisy','Petunia','Tuxedo', ...);
```

Array III

Stampa (semplice) di un array

```
print "@gatti\n";
```

Array vuoto

```
@gatti = ();
```

Array IV

Aggiungere e rimuovere elementi dalla fine di un array

```
my @meals;

push @meals, ('hamburgers', 'spinach');
pop @meals;
```

Aggiungere e rimuovere elementi dall'inizio di un array

```
unshift @meals, ('tofu', 'taquitos');
shift @meals;
```

Hash (array associativi) I

- Rappresenta un insieme di coppie *chiave/valore*
- Struttura dati che associa una chiave (string) ad una variabile scalare
- È possibile usare l'operatore => per associare un valore ad una chiave in maniera più chiara

Sono preceduti dal simbolo **%**

Hash (array associativi) II

Dichiarazione (1)

```
my %cibo_preferito;
$cibo_preferito{Francesco} = 'Pizza';
$cibo_preferito{Giovanni} = 'Ciambella';
```

Dichiarazione (2)

```
my %cibo_preferito = (
  'Francesco', 'Pizza',
  'Giovanni', 'Ciambella',
);
```

Hash (array associativi) III

Dichiarazione (3)

```
my %cibo_preferito = (
  Francesco => 'Pizza',
  Giovanni => 'Ciambella',
);
```

Array associativo vuoto

```
%cibo_preferito = ();
```

Contesto

Ogni operazione in Perl è interpretata in uno specifico **contesto**

- Perl usa il *contesto* al fine di capire come trattare determinati tipi di dati
- Un'operazione verrà eseguita in un contesto o in un altro in base al tipo dell'operazione stessa
  - Si sta assegnando un valore ad uno scalare ? $\rightarrow$ *Scalar Context*
  - Si sta iterando su una lista ? $\rightarrow$ *List Context*

"Amount" context

- **Void**
- **Scalar**
- **List**

"Kind" context

- **Numeric**
- **String**
- **Boolean**

Contesto - Esempi I

Scalari - Contesto String

```
my $zip_code = 97123;
my $city_zip = 'Hillsboro, OR ' . $zip_code;
```

Contesto - Esempi II

Operazioni matematiche su stringhe

```
my $call_sign = 'KBMIU';

# aggiorna la variabile e ritorna un nuovo valore
my $next_sign = ++$call_sign;

# ritorna il vecchio valore e *successivamente*
   aggiorna la variabile
my $curr_sign = $call_sign++;
```

Contesto - Esempi III

Accesso (indicizzato) agli elementi di un array

```
# @gatti contiene una lista di oggetti di tipo Gatto
my $first_cat = $gatti[0];
```

Contesto - Esempi IV

Array in Scalar context - numero di elementi

```
# assegnamento scalare
my $num_gatti = @gatti;

# concatenazione di string
print 'Posseggo ' . @gatti . " gatti!\n";

# addizione
my $num_animals = @gatti + @cani + @pesci;
```

Contesto - Esempi V

Gli indici degli array

```
my $first_index = 0;
my $last_index = @gatti - 1;
# oppure
# my $last_index = $#gatti;

my $last_cat = $gatti[-1];
my $second_to_last_cat = $gatti[-2];
```

Contesto - Esempi VI

Array slice

```
my @youngest_cat = @gatti[-1, -2];
my @oldest_cat = @gatti[0 .. 2];
my @selected_cat = @gatti[ @indexes ];
```

Contesto - Esempi VII

Accesso (key based) ai valori di un array associativo

```
my $address = $addresses{$name};

# automaticamente riconosciuto
my $address = $addresses{Victor};

# necessita le ' ' altrimenti non viene
   automaticamente riconosciuto
my $address = $addresses{'Sue-Linn'};
```

Contesto - Esempi VIII

Controllare se un array associativo contiene una chiave

```
my %addresses = (
  Leonardo => '1123 Fib Place',
  Utako => 'Cantor Hotel, Room 1',
);

print "Have Leonardo's address\n" if exists
   $addresses{Leonardo};
print "Have Warnie's address\n" if exists
   $addresses{Warnie};
```

Contesto - Esempi IX

[Advanced] Trovare gli elementi univoci in una lista

```
my %uniq;
my @uniques = keys %uniq;
print @uniques;
```

Input/output - Uso di scalari e array

Esercizio 1

- Creare uno script che presi 2 interi in input ne mostri la somma, la differenza, il prodotto e il quoziente
- Rimodellare lo stesso programma facendo però uso delle subroutine. Creare quindi una subroutine che prende come parametri 2 variabili per ogni operazione matematica da eseguire
- Creare uno script che presi in input una sequenza di numeri positivi terminati da tappo "-1", li inserisca in un array e successivamente ne calcoli la somma

Input/output - Uso di array associativi

Esercizio 2

- Creare uno script che modelli il funzionamento di una rubrica telefonica
  - Ad ogni persona, identificata tramite nome e cognome, è attribuito un unico numero di telefonica
- Stampare in output per ogni persona il proprio numero di telefono e la lista univoca delle persone in rubrica

Stringhe, variabili e operatori

Esercizio 3

- Date 2 stringhe in input si vuole:
  - Concatenarle;
  - Trovare la lunghezza della nuova sequenza;
  - Stampare la seconda metà della nuova stringa.

  (**Suggerimento:** *puoi usare la funzione* **substr(expr,offset,length)**)

Esercizio 4

- Dato in input una stringa X ed una lettera Y, contare il numero di occorrenze di Y nella stringa X.

(**Suggerimento:** *puoi usare la funzione* **substr(expr,offset,length)** *oppure* **split(/pattern/,expr,limit)**)

Esercizio 5

- Data una stringa in input, creare una subroutine che calcoli il suo complemento inverso.

(**Esempio:** *input = abc, complemento = cba)*

Array e Hash

Esercizio 6 [2]

- Creare un array associativo ed inizializzarlo con i seguenti valori: **A => 1, B => 2**
- Stampare il valore associato alla chiave **A**
- Aggiungere altre 4 nuove coppie di chiave/valore nell'hash
- Cambiare il valore di **A** e settarlo a **0**
- Controllare se esiste un elemento con chiave **C**
- Estrarre una sezione a scelta dei valori dell'hash

  Esempio: *se l'hash fosse composto da A=>1, B=>2 e C=>3, si potrebbe decidere di estrarre da esso i valori delle chiavi A e B e inserirle in un array*
- Estrarre e salvare in un array tutte le chiavi e successivamente tutti i valori dell'hash
- Stampare in output la dimensione dell'hash
- Rimuovere dall'hash gli elementi con chiave **A** e **B**

[2] Esempi pratici a questo link: https://www.tutorialspoint.com/perl/perl_hashes.htm

Operatori

OPERATORI

Operatori - Caratteristiche

Precedenza

La *precedenza* di un operatore decide l'ordine in cui valutare una espressione

Associatività

La proprietà *associativa* di un operatore decide se valutare una espressione da sinistra verso destra o da destra verso sinistra

Arietà

L' *arietà* di un operatore indica il numero di operandi sul quale esso opera

Un operatore *"nullary"* opera su zero operandi
Un operatore *"unary"* opera su un operandi
Un operatore *"binary"* opera su due operandi
Un operatore *"trinary"* opera su tre operandi
Un operatore *"listary"* opera su lista di zero o più operandi

Caratteristiche

Fixity

Gli operatori *infissi* appaiono al centro tra 2 operandi

```
$length * $width;
```

Gli operatori *prefissi* precedono gli operandi

```
-$x # negazione matematica
!$y # negazione booleana
```

Gli operatori *postfissi* seguono i loro operandi

```
$z++ # incremento postfisso
```

Gli operatori *postcircumfix* seguono determinati operandi e ne racchiudono altri

```
$hash{$x} # accesso agli elementi di un hash
$array[$y] # accesso agli elementi di un array
```

Operatori numerici

Gli operatori numerici impongono il contesto *numeric* sui loro operandi

- Operatori aritmetici standard **+, - * /, **, %**
- Varianti degli operatori aritmetici **(+=, -=, *=, /=, **=, %=)**
- Operatori di comparazione
  eguaglianza numerica **(==)**, ineguaglianza numerica **(!=)**,
  maggiore/minore di **(>, <)**, maggiore/minore o uguale **(>=, <=)**,
- Operatore di comparazione per ordinamento **(<=>)**

Operatori per stringhe

Gli operatori per le stringhe impongono il contesto *string* sui loro operandi

- Operatori specifici per le stringhe
  binding per espressioni regolari (**=~** and **!~**),
  concatenazione (**.**)
- Operatori di comparazione
  eguaglianza su stringhe (**eq**), ineguaglianza su stringhe (**ne**),
  maggiore/minore di (**gt, lt**), maggiore/minore o uguale (**ge, le**),
- Operatore di comparazione per ordinamento (**cmp**)

Operatori logici

Gli operatori booleani impongono il contesto *boolean* sui loro operandi

- Operatori booleani standard
  `&&`, `||`, `!`,
  `and`, `or`, `not`, `xor`
- Operatori booleani speciali
  operatori condizionali ternari **`(? :)`**

Altri operatori

Operatori bitwise

Trattano i loro operandi numericamente a livello di bit
left shift **(<<)**, right shift **(>>)**,
bitwise and **(&)**, bitwise or **(|)** and bitwise xor **(^)**,
e le loro varianti con assegnamento **(<<=, >>=, &=, |=, ^=)**

Operatore di ripetizione

```
my @test = ('nights') x 1001;
# conterra' ('nights', 'nights', ...)

my $calendar = 'nights' x 1001;
# conterra' "nightsnights..."
```

Operatore di range infisso

```
my @cards = ( 2 .. 10, 'J', 'Q', 'K', 'A' );
```

Esercizi - Operatori

Esercizio 1

- Creare un array associativo composto dai seguenti valori: A => 1, B => 45, C => 5
- Ordinare in maniera crescente i valori dell'array associativo utilizzando prima l'operatore compare per il contesto *numeric* e successivamente quello per il contesto *string*.
- Quali sono le differenze nell'output ? E perchè ?
- Aggiungere all'array associativo i valori: D => 45, E => 10, F => 1
- Ordinare e stampare prima in ordine crescente i valori contenuti nell'array associativo e a parità di valore ordinare lessicograficamente sulle chiavi

Control Flow

CONTROL FLOW

Control Flow

Definizione

Il *control flow* identifica l'ordine in cui le istruzioni o le funzioni di un programma imperativo sono eseguite o valutate.

I programmi imperativi sono eseguiti a partire dalla prima istruzione fino ad arrivare all'ultima

```
print ("Inizio\n");
print ("Centro\n");
print ("Fine\n");
```

Perl control flow

Alcune direttive in Perl cambiano l'ordine di ciò che potrebbe accadere più in là nel programma

If statement

postfix `if`

```
print $cont if $cont > 10;
```

prefix `if`

```
if ($cont > 10)
{
    print $cont;
}
```

Espressioni condizionali

postfix expression

```
greet_bob() if ($name eq 'Bob' && not
    greeted_bob());
```

prefix expression

```
if ($name eq 'Bob' && not greeted_bob())
{
    print 'Hello, Bob!';
    found_bob();
}
```

Unless statement

postfix `unless`

```
print "You're not Bob!" unless $name eq 'Bob';
```

prefix `unless`

```
unless ($name eq 'Bob')
{
    print "You're not Bob!"
}
```

If-else statement

postfix `unless`

```
if ($name eq 'Bob')
{
    print 'Hi, Bob!';
}
else
{
    print "I don't know you.";
}
```

Unless-else statement

unless-else statement

```
unless ($name eq 'Bob')
{
    print "I don't know you.";
}
else
{
    print 'Hi, Bob!';
}
```

Unless-else statement

if-elsif-else statement

```
if ($name eq 'Robert')
{
    print 'Hi, Robert !';
}
elsif ($name eq 'James') # Attenzione "else if" NON e'
    corretto
{
    print 'Hi, James !';
}
else
{
    print "You're not my friend.";
}
```

Operatore ternario condizionale

Operatore ternario condizionale

```
my $time_suffix = after_noon($time)
                    ? 'afternoon'
                    : 'morning';
# Equivalente a
my $time_suffix;
if (after_noon($time))
{
    $time_suffix = 'afternoon';
}
else
{
    $time_suffix = 'morning';
}
```

Variabili di default

Variabili di default

In Perl le variabili `$_` e `@_` sono variabili globali di default

- `$_` è usata per le variabili scalari
- `@_` è usata per gli array

Construtti per il loop

prefix for-each

```
# Fare il quadrato dei primi 10 numeri
foreach (1 .. 10)
{
    print '$_ * $_ = ' . $_ * $_ . "\n";
}
```

postfix for-each

```
print '$_ * $_ = ' . $_ * $_ . "\n" for 1 .. 10;
```

postfix for-each

```
for my $i (1 .. 10)
{
    print '$i * $i = ' . $i * $i . "\n";
}
```

Iterare su un array I

Iterare su un array - versione 1

```
my @nums = 1 .. 10;
$_ **= 2 for @nums;
print join(", ", @nums);
```

Iterare su un array - versione 2

```
for my $num (@nums)
{
    $num **= 2;
}
print join(", ", @nums);
for (@nums)
{
    $_ **= 2;
}
print join(", ", @nums);
```

Iterare su un array II

Iterare su un array - versione 3

```
for (my $i = 0; $i <= 10; $i += 2)
{
    print '$i * $i = ' . $i * $i . "\n";
}
```

Loop

While loop

```
while (@values)
{
    my $value = shift @values; # Attenzione: senza shift diventa un loop infinito
    print $value;
}
```

until loop

```
until ($finished_running) { ... }
```

Postfix until loop

```
print "Hello, world! " until !$stop;
```

Loop Infinito

Infinite **for** loop

```
for (;;) { ... }
```

Infinite **while** loop

```
while (1) { ... }
```

Postfix infinite **while** loop

```
print "Hello, world! " while 1;
```

Loop su array associativi

Loop sulle chiavi di array associativi

```
for my $addressee (keys %addresses)
{
    print "Found an address for $addressee!\n";
}
```

Loop sui valori di array associativi

```
for my $address (values %addresses)
{
    print "Someone lives at $address\n";
}
```

Loop su liste di chiave/valore di array associativi

```
while (my ($addressee, $address) = each %addresses)
{
    print "$addressee lives at $address\n";
}
```

Controllo sui loop

- **next** - Salta una iterazione del loop (simile al continue in C)
- **last** - Esce immediatamente dal ciclo più interno (simile al break in C)
- **redo** - Restart del loop senza valutare nuovamente la condizione

Files

FILES

Introduzione

Molti programmi interagiscono con il mondo reale per lo più leggendo, scrivendo e naturalmente manipolando i files.

Perl era inizialmente usato come un tool di programmazione per gli amministratori di sistemi e continua ad essere ancora oggi un ottimo linguaggio utilizzato per la manipolazione di testi.

Filehandle

Definizione

Il *filehandle* rappresenta lo stato corrente di uno specifico canale di input o output.

Ci sono 3 canali standard (default) per il filehandle:

- STDIN - Input
- STDOUT - Output
- STDERR - Errore

Esempio

Lettura da STDIN

```
print "Inserisci una frase \n";
$line = <STDIN>; # Legge la successiva riga
    inserita da standard input
print "linea ricevuta:\n$line";
print "inserisci le righe successive!\n";
@lines = <STDIN>; # Legge tutte le righe inserite
    da standard input fino all'inserimento di ^D
print "restanti linee ricevute:\n@lines";
```

Input & Output

Aprire un file in **lettura**

```
open(my $fh, "<", "input.txt")
    or die "Can't open < input.txt: $!";
```

Aprire un file in **scrittura**

```
open(my $fh, ">", "ouput.txt")
    or die "Can't open > output.txt: $!";
```

Chiudere un file sia in **lettura** che in **scrittura**

```
close $fh;
```

Input & Output

Principali modalità di apertura dei files

- **<** Apre un file in modalità solo **lettura**
- **>** Apre un file in modalità solo **scrittura** cancellandone il contenuto se il file era già esistente, altrimenti ne crea uno nuovo
- **>>** Apre un file in modalità solo **scrittura** appendendo il nuovo testo a quello esistente; se il file non era presente ne viene creato uno nuovo
- **+<** Apre il file *sia* in **lettura** *che* in **scrittura**
- **-|** Legge l'output prodotto dalla shell pipeline
- **|-** Scrive sulla shell pipeline

Input & Output

Leggere un file

```
while (my $line = <$fh>) {
    chomp $line; # remuove i newlines
    ...
```

Leggere un file usando la variabile di default $_

```
while (<$fh>) {
    chomp;
    ...
```

Input & Output

Perchè si usa il **while** e NON il **for** ?

`for` impone il contesto *list* sugli operandi. Quando si è in *list* context, la funzione *readline* leggerà l'intero file prima di processarlo.

`while` effettua l'iterazione e legge una linea per volta. Quando è importante risparmiare memoria, è preferibile usare il **`while`**.

Input & Output

Scrivere su un file

```
print $out_fh "Questa linea sara' scritta sul
    file\n";
```

Esercizi - Files

Esercizio 1

- Aprire in modalità solo lettura il file /etc/passwd
- Scorrere il file e stampare tutte le righe pari

Esercizio 2

- Aprire in modalità solo lettura il file /etc/passwd
- Stampare su un file output.txt (aperto in modalità solo scrittura) tutte le righe che contengono come nome utente "root"
  (NB.: Si può utilizzare la funziona split con delimitatore il carattere ":" **oppure** una espressione regolare)

Attenzione:

1. Gestire sempre le eccezioni in fase di apertura e chiusura dei file (stai provando a leggere un file che non esiste ?)
2. È importante che i file rimangano aperti solo il tempo necessario per eseguire le operazioni di lettura/modifica su di essi e non per l'intera esecuzione dello script

Esercizi - Files

Esercizio 2

- Creare uno script che prende come argomenti i nomi di almeno 2 file di testo (es: `perl script.pl file1 file2`) e produca in output un unico file di testo chiamato *merge.txt* che contiene il contenuto di tutti i file passati precedentemente

Espressioni Regolari e Matching

ESPRESSIONI REGOLARI E MATCHING

Introduzione

Perl è talvolta chiamato *Practical Extraction and Reporting Language*

Espressione Regolare

Una espressione regolare (chiamata anche *regex* o *regexp*) è un pattern che descrive le caratteristiche di un particolare pezzo di testo

Il *regular expression engine* di Perl applica questo pattern al fine di trovare un matching tra determinate porzioni di testo.

Sintassi

Una espressione regolare è racchiusa tra 2 slash '/':

/ ESPRESSIONE REGOLARE /

Introduzione

Gli elementi principale di una espressione regolare sono:

- **Elementi (Literals)** $\rightarrow$ Rappresentano una parte di testo
- **Qauntificatori (Quantifiers)** $\rightarrow$ Indicano quante volte ci si aspetta che un certo elemento (o gruppo di elementi) possa essere ripetuto

Gli elementi più semplici sono i carateri alfanumerici.

Esempio:
L'espressione regolare /**a**/ indica che una stringa soddisfa il pattern se contiene almeno una **a**

Pattern Matching

Un pattern matching si definisce mediate l'operatore di ricerca *m*, del tipo:

```
# m/REGEXP/
```

Pattern Matching - Operatore =~

```
$testo = "Oggi e' una bella giornata!";
print "La stringa $testo verifica il match con
    /la/. " if $test =~ m/la/;
# Oppure
print "La stringa $testo verifica il match con
    /la/. " if $test =~ /la/;
```

Pattern Matching

L'operatore $=\sim$ verifica se esiste una corrispondenza tra una stringa ed una certa espressione regolare, avverte Perl che la parte successiva è una espressione regolare. Questo operatore si chiama operatore di binding, per il fatto che effettua una connessione (binding) fra la variabile **$testo** e l'espressione regolare /**la**/.

In sostanza, può essere letto come "applica l'espressione regolare a destra sul testo della variabile a sinistra"

Non si tratta di un assegnamento!!

Modificatori

I *modificatori* di una REGEXP ne determinano il comportamento:

- /**REGEXP[/SUBST]/g** $\rightarrow$ **g** è il modificatore **globale**: sarà ricercata o sostituita ogni evenienza
- /**REGEXP[/SUBST]/i** $\rightarrow$**i** è il modificatore **case-insensitive**: l'espressione regolare sarà case-insensitive

Esempio

```
$testo = "Oggi e' una bella giornata!";
print "$testo verifica il match /LA/
       case-insensitive. \n" if $testo !~ m/LA/i;
```

Come avviene la verifica del match ?

Viene cercata la posizione più a sinistra in cui si riscontra l'intera espressione regolare. La stringa viene esaminata da sinistra a destra finché non viene trovato un riscontro di regexp o finché il confronto fallisce.

Metacaratteri

il "dot" meta character

```
"a" =~ /./   # Match
"." =~ /./   # Match
"" =~ /./    # No match (dot has to match a
   character)
"\n" =~ /./ # No match (dot does not match a
   newline)
```

Metacaratteri

Metacharacters

```
\    Annulla gli effetti del metacarattere
   successivo
^    Identifica l'inizio di una riga; inoltre
   all'inizio di un gruppo nega il gruppo stesso
.    Qualsiasi carattere ad eccezione di quelli che
   identificano una riga nuova
$    Identifica la fine di una riga
|    Indica una condizione OR
()   Indicano un gruppo di caratteri
[]   Indicano intervalli e classi di caratteri
```

Quantificatori

Quantificatori

```
*       Indica 0 o tante occorrenze
+       Indica 1 o tante occorrenze
?       Indica al massimo 1 occorrenza
{n}     Ricerca esattamente n occorrenze
{n,}    Ricerca minimo n occorrenze
{n,m}   Ricerca minimo n e massimo m occorrenze
```

Tabulazioni e newlines

Tabulazioni e newlines

\t	tab	(HT, TAB)
\n	newline	(LF, NL)
\r	**return**	(CR)

Altri metacaratteri

Tabulazioni e newlines

```
\d Ricerca un numero (d sta per digit).
\D Opposto di \d, ricerca qualsiasi cosa che non
   sia un numero.
\w Ricerca un carattere "parola" (w sta per word),
   ovvero lettere, numeri e "_" -> [a-zA-Z0-9_]
\W Ricerca un carattere che non sia \w, ovvero
   tutto quello che non sia lettere, numeri o "_"
\s Ricerca uno spazio, comprese tabulazioni e
   caratteri di fine riga.
\S Opposto di \s. Ricerca qualsiasi cosa che non
   sia uno spazio, una tabulazione o dei caratteri
   di fine riga
\N Ricerca un carattere che non sia newline.
```

Esempi - REGEXP

Esempio

```
o.a       # "o" seguita da qualsiasi carattere
   seguito da "a"
^f        # "f" all'inizio di una riga
^yogi     # yogi all'inizio di una riga
e$        # "e" alla fine di una riga
notte$    # notte a fine riga
aiuto*    # aiuto seguito da zero o piu' caratteri "o"
.*        # Qualsiasi stringa senza un newline
^$        # Una riga vuota
```

Esempi - REGEXP

Esempio

```
[qjk]       # Alternativamente q oppure j oppure k
[^qjk]      # Ne' q, ne' j, ne' k
[a-z]       # Qualsiasi carattere tra a e z inclusi
[^a-z]      # Nessuna lettera minuscola
[a-zA-Z]    # Qualsiasi lettera
[a-z]+      # Qualsiasi sequenza non vuota di
   lettere minuscole
```

Esempi - REGEXP

Esempio

```
arancia|limone      # Alternativamente arancia o
    limone
cas(a|erma)         # casa oppure caserma
(la)+               # la oppure lala oppure lalala
    oppure ...
```

Catturare parti del matching

Spesso è utile potersi ricordare dei pattern per cui c'è stato un match, così da poterli ri-utilizzare

Ciò che avviene è che qualsiasi stringa che soddisfa un match è memorizzata da Perl nelle variabili speciali `$1,...,$9`

In pratica, le variabili `$1, $2,` etc. funzionano all'esterno di un criterio di ricerca e contengono le sequenze effettivamente riscontrate nei vari raggruppamenti in `()` dell'espressione regolare e contate da sinistra a destra

Questo meccanismo consente di "catturare" singole parti dell'espressione regolare per cui si verifica il match

Esempi - REGEXP

Controllo del codice fiscale del tipo MNL TMS 74 M 02 L736 Y

```
$stringa = "MNL TMS 74 M 02 L736 Y";
$stringa =~ /([A-Z]{3})\s([A-Z]{3})\s(\d{2})\s

    ([A-Z])\s(\d{2})\s([A-Z][0-9]{3})\s([A-Z])
                /;
# stampa di tutti i "pezzi" della stringa che hanno
   soddisfatto i match separati da ;
print "CF: $1;$2;$3;$4;$5;$6;$7\n";
```

Quantificatore "ingordo"

Il meccanismo dei quantificatori è "ingordo", ovvero esamina riscontrati. Se questa sequenza non soddisfa l'espressione regolare allora diminuisce il numero di atomi fino al minimo consentito finché non riesce a soddisfare la regex.

Esempio

```
$testo = '"Ciao!", gli disse lei, "Ciao." rispose
    lui.';
$testo =~ /"(.+)"/;
print "$1\n"; # Ciao!", gli disse lei, "Ciao.
```

Quantificatore "minimale"

Posticipando il carattere `?` ad un quantificatore lo si rende quantificatore minimale, ovvero verrà cercata la prima occorrenza possibile del carattere/gruppo, minimizzando il risultato

Esempio

```
$testo = '"Ciao!", gli disse lei, "Ciao." rispose
    lui.';
$testo =~ /"(.+?)"/;
print "$1\n"; # Ciao!
```

Altri esempi

Esempio

```
$string = "aaaabbbaaaaabbbaaaa";
$string =~ /(.*)bbb/;
print "$1\n"; # "aaaabbbaaaaa"

# Se si vuole "aaaabbb" si deve scrivere
$string =~ /(.*?)bbb/;
print "$1\n"; # "aaaa"
```

Sostituzioni

ESPRESSIONI REGOLARI - SOSTITUZIONI

Sostituzioni

Oltre a poter trovare se una stringa soddisfa una particolare espressione regolare, Perl permette di fare sostituzioni basate sulle corrispondenze individuate.

Il modo per fare questo è usare l'operatore **s**

Esempio 1 - Sostituzione

```
$frase = "Universita' degli Studi della Calabria";
print "PRIMA => $frase \n"; # Uni... degli ...
$frase =~ s/degli/Degli/g;
print "DOPO => $frase \n"; # Uni... Degli ...
```

Sostituzioni - Le variabili speciali

Le variabili speciali `$1, $2`, etc. possono essere usate nelle espressioni regolari o nelle sostituzioni mediante i codici speciali `\1, ..., \9`.

Racchiudere le lettere maiuscole tra i due punti ":"

```
$frase = "Universita' degli Studi della Calabria"; print
    "PRIMA => $frase \n";
$frase =~ s/([A-Z])/:\1:/g;
print "DOPO => $frase \n"; # :U:niversita' degli :S:tudi
    della :C:alabria
```

Invertire l'ordine delle parole

```
$frase = "Universita' degli Studi della Calabria"; print
    "PRIMA => $frase \n";
$frase =~ s/(.*) (.*) (.*) (.*) (.*)/\5 \4 \3 \2 \1/g;
print "DOPO => $frase \n"; # Calabria della Studi degli
    Universita'
```

Traslitterazione

La funzione **tr** permette una traduzione (traslitterazione) carattere per carattere.

Il modo per fare questo è usare l'operatore **tr**

Esempio 1 - Traslitterazione

```
$frase = "Universita' degli Studi della Calabria"; print "PRIMA => $frase \n";
$frase =~ tr/USC/OTZ/;
print "DOPO => $frase \n"; # Oniversita' degli Ttudi della Zalabria
```

Esempio 2 - Traslitterazione

```
$frase = "Universita' degli Studi della Calabria"; print "PRIMA => $frase \n";
$frase =~ tr/a-z/A-Z/;
print "DOPO => $frase \n"; # UNIVERSITA' DEGLI STUDI DELLA CALABRIA
```

Perl Funzioni Avanzate

Perl Funzioni Avanzate

Esecuzione di comandi shell

Una stringa racchiusa tra apici inversi (backticks ‘ ‘), oppure indicata attraverso l’operatore **qx**, viene interpolata ed il risultato viene fatto eseguire dal sistema operativo.

L’output del comando è il risultato della valutazione della stringa e il valore restituito dal comando può essere letto

Comandi Shell

```
@output =`ls -l`;
# oppure
@output = qx(ls -l);
print "output del comando @output\n";
```

Sorting di array associativi

Sorting delle chiavi

```
%studenti = (
    "ale", 10567,
    "cla", 1789,
    "adri", 6443
);
# Stampa ordinata in base alla chiave (nome)
foreach $nome (sort keys %studenti) {
    print "$nome - $studenti{$nome}.\n";
}
```

Sorting di array associativi

sort consente di specificare una routine di ordinamento:
sort `{$a cmp $b}` ordinamento lessicografico
sort `{$b cmp $a}` come prima ma in ordine inverso
sort `{$a <=> $b}` ordinamento numerico ascendente
sort `{$b <=> $a}` ordinamento numerico discendente
sort `{($hash{$b} <=> $hash{$a}) || ($a cmp $b) }`
ordinamento prima per valore e poi per chiave in array associativi

Stampa ordinata in base al valore (matricola), ascendente

```
foreach $nome (sort { $studenti{$a} <=>
    $studenti{$b} } keys %studenti)
{
    print "$nome - $studenti{$nome}.\n";
}
```

Split su stringhe

La funzione split "spezza una stringa in pezzi mettendoli in un array"

Per specificare cosa deve essere considerato come separatore si passa a questa funzione un'espressione regolare

Sorting delle chiavi

```
$info = "root:x:0:0:root:/root:/bin/bash";
@dati = split( /:/, $info );
print "i dati sono @dati\n";
```

Split su stringhe

L'array `@ARGV` contiente gli argomenti passati allo script tramite linea di comando

`$#ARGV` è pari al numero di argomenti meno 1 perchè in `$ARGV[0]` è contenuto il primo argomento

Le operazioni su array in Perl (incluse **`shift`** e **`pop`**) operano su `@ARGV` implicitamente al di fuori delle funzioni

Split su stringhe

Stampa tutti gli argomenti

```
foreach (@ARGV)
{
    print "$_\n";
}
```

Prende il primo argomento

```
my $number = shift or die "Usage: $0 NUMBER\n";
```

Caso Speciale

`ARGV` ha un caso speciale. Se si legge da un *null file handle* `<>`, Perl tratterà ogni elemento in `@ARGV` come se fosse il nome di un file da aprire in lettura.

Se `@ARGV` è vuoto, Perl leggerà da standard input `<STDIN>`

Nota che `$ARGV` contiene il nome del file corrente mentre si legge da `<>`

Processo il contenuto di ogni file passato come argomento

```
while (<>)
{
    ...
}
```

References


chromatic (2015).
Modern Perl.
The Pragmatic Bookshelf, fourth edition.
http://www.modernperlbooks.com/books/modern_perl_2016.


perl5porters (2017).
perldoc.perl.org Official documentation for the Perl programming language.
https://perldoc.perl.org/.


Schwartz, R. L., brian d foy, and Phoenix, T. (2016).
Learning Perl.
O'Reilly Media, 7th edition.
https://www.learning-perl.com.