Perl

Perl

- da “man Perl”:

  Perl is a language optimized for **scanning** arbitrary **text files**, **extracting** information from those text files, and **printing** reports based on that information.

  It's also a good language for many **system management** tasks.

  The language is intended to be **practical** (easy to use, efficient, complete) **rather than beautiful** (tiny, elegant, minimal).

Perl – caratteristiche

- interpretato (con fase di precompilazione)
- procedurale, dotato di alcuni costrutti funzionali (e.g. map, grep)
  - supporta il paradigma object oriented
- garbage collection automatica
- lascamente tipato a runtime
  - alcuni controlli (opzionali) nella fase di precompilazione
- sintassi concisa, veloce da scrivere
- nessun limite alla dimensione dei dati
- supporto
  - ottima documentazione (man page, libri O'Reilly, tutorial)
  - ampia libreria standard e contributi di terzi parti (CPAN)

Perl – caratteristiche

- punti di forza
  - espressioni regolari
  - TIMTOWTDI (There Is More Than One Way To Do It)
  - disponibilità di librerie
  - portabilità
- punti di debolezza
  - richiede dedizione
  - scalabilità
  - leggibilità del codice
- riferimenti:
  - man perl (e tutte le manpage derivate)
  - appunti: capp 377—382
  - *Programming Perl*, Larry Wall et al., O'Reilly

Hello, World!

- i programmi Perl sono script eseguibili dall'interprete (/usr/bin/perl)

```
#!/usr/bin/perl
print “Hello, world!\n”;
```

- meglio (abilita warning e controlli statici):

```
#!/usr/bin/perl –w
use strict;
print “Hello, world!\n”;
```

“Bignami” della sintassi

- sintassi Perl ~ sintassi C + sintassi sh + sintassi sed
- un programma Perl è composto da una sequenza non vuota di *statement*
  - ogni statement è terminato da “;”
  - commenti: iniziano con “#”, si estendono fino a fine riga
  - i blank sono indistinguibili (al di fuori delle stringhe)
- gli identificatori rispettano le convenzioni lessicali di C
- le stringhe possono essere delimitate da ' ... ' o “ ... ”
  - all'interno delle stringhe delimitate da “ ... ” viene eseguita espansione delle variabili
- l'uso di parentesi nell'invocazione di funzioni è opzionale
- riferimenti: man perlsyn

$Hello, World!!

```
#!/usr/bin/perl –w
use strict;
print(
  '$Hello');
$a = “world!”;
print “, ”;      print “$a!\n”;
```

Tipi di Perl

- le variabili Perl sono lascamente tipate
- ogni variabile ha esattamente uno dei seguenti tipi
  - scalare
  - array
  - hash
- il tipo di una variabile è identificato in base al carattere che la precede
  - $ scalare (e.g. $foo)
  - @ array (e.g. @bar)
  - % hash (e.g. %baz)
- riferimenti: man perldata

Scalari

- una variabile di tipo scalare può contenere un *valore singolo ed indivisibile*
- sono scalari: interi, numeri in virgola mobile, stringhe, caratteri
- Perl effettua in automatico la conversione tra scalari diversi quando necessario

```
my $animal = "camel";
my $answer = 42;
print $animal;
print "The animal is $animal\n";
print "The square of $answer is ",
      $answer * $answer, "\n";
```

Array

- un variabile di tipo array può contenere una *sequenza ordinata di valori* (non necessariamente dello stesso tipo)

```
my @animals = ("camel", "llama", "owl");
my @numbers = (23, 42, 69);
my @mixed   = ("camel", 42, 1.23);
```

- accesso agli array (zero-based)

```
print $animals[0]; # notate il "$"!
print $animals[1];
```

- dimensioni ed ultimo elemento

```
print $#mixed;  # indice dell'ultimo elemento
print @animals * 2;# num. degli elementi * 2
print ($#animals + 1) * 2   # perchè faticare?
```

Array slice

- è possibile accedere direttamente a sottoarray

```
my @animals = ("camel", "llama", "owl");
@animals[0,1];
  # ritorna l'array ("camel", "llama")
@animals[0..2];
  # ritorna l'array ("camel", "llama", "owl")
@animals[1..$#animals];
  # tutti tranne il primo
```

Hash

- una variabile di tipo hash può contenere una *lista di associazioni chiave/valore*

```
my %fruit_color =
  ("apple", "red", "banana", "yellow");
my %fruit_color = (
  apple  => "red",
  banana => "yellow");
```

- accesso

```
$fruit_color{"apple"};  # ritorna "red"
```

- array delle chiavi e dei valori

```
my @ks = keys %fruit_color; # (“apple”, “banana”)
my @vs = values %fruit_color; # (“red”, “yellow”)
```

Alcune variabili predefinite

- scalari
  - $$ pid
  - $< uid
  - $? return code dell'ultimo comando di sistema eseguito
  - $_ variabile di input e pattern matching predefinita
- array
  - @ARGV parametri posizionali
  - @_ argomenti di una subroutine
- hash
  - %ENV variabili di ambiente
- riferimenti: man perlvar

Scoping e blocchi

- Perl supporta il raggruppamento di istruzioni in blocchi racchiusi tra { ... }
- definizione di variabil
  - globali (e.g. $foo = “bar”)
  - locali (e.g. my $foo = “bar”)
  - temporanee (scoping dinamico)
- le variabili locali sono soggetta alla semantica classica dello scoping statico
  - sono visibili solamente all'interno del blocco nel quale sono definite
- riferimenti: man perlsub

Scoping e blocchi

```
my $a = "foo";
{
  my $b = "bar";
  print $a;       # stampa "foo"
  print $b;       # stampa "bar"
}
print $a;         # stampa "foo"
print $b;         # non stampa nulla
```

Controllo condizionale

- if ... then ... else

```
if ( condition ) { # n.b. non c'e' "then"
  ...
} elsif ( other condition ) {
  ...
} else {
  ...
}
```

- unless (equivalente a if (! condition))

```
unless ( condition ) {
  ...
}
```

- le condizioni sono espressioni Perl
  - i valori 0, “0”, “”, (), undef rappresentano il falso

Controllo condizionale

- la sintassi del controllo condizionale richiede un blocco anche nel caso di statement singolo

```
if ($completed) {
  print "done."
}
```

- per abbreviare la sintassi nel caso di statement singolo esistono versioni postfisse dei controlli condizionali
  - si ottengono statement simili alla lingua inglese

```
print "done." if $completed;
print "out of money" unless $money;
```

Cicli limitati

- for
  - sintassi molto simile al C, stessa semantica

```
for (my $i=0; $i <= 10; $i++) {
  print “foo\n”;
}
```

- foreach
  - costrutto che permette di iterare sui valori di un array

```
foreach (@array) {
  print "This element is $_\n";
}
foreach my $k (keys %hash) {
  print "key $k, value $hash{$k}";
}
```

Alcuni operatori

- aritmetici
  - binari: + - * /
  - unari: -
- confronti
  - numerici: == != < > <= >=
  - letterali: eq ne lt gt le ge
- booleani: && || ! and or not
- altri:
  - concatenazione di stringhe .
  - moltiplicazione di stringhe x
  - range ..

Alcuni operatori

- assegnamento =
- incremento ++ -- += *= -= .= ecc.
- riferimenti: man perlop

Manipolazione di valori

- la libreria standard di Perl dispone di un ampio numero di funzioni predefinite, riportiamo alcune delle funzione atte a manipolare valori del linguaggio
- manipolazione di scalari:
  - chomp, lc/uc, reverse, chr/ord, length
- manipolazione di array:
  - pop, push, shift, unshift, grep, join, map, reverse, sort
- manipolazione di hash:
  - delete, exists, keys, values
- riferimenti
  - man perlfunc
  - perldoc -f *<nome_funzione>*

I/O (1/2)

- la gestione dell'I/O in Perl è stata progettata per essere simile ai concetti della shell
- questa analogia è rispettata dalla sintassi di open

```
open(INFO, "< info") || die "can't open info";
open RESULTS, "> output" or die "can't save";
open(LOG, ">> my.log") || die ...;
open(PRINTER, "| lpr") || die ...;
open(LOG, "tail –f my.log |”) || die ...;
```

- il primo argomento di open è il nome del *filehandle* associato al file una volta aperto
- chiusura di un filehandle

```
close RESULTS;
```

- riferimenti: man perlopentut

I/O (2/2)

- lettura da un filehandle

```
my $line = <INFILE>;          # una riga
my @lines = <INFILE>;         # tutte le righe
```

- scrittura su di un filehandle

```
print OUTFILE "una riga qualsiasi\n";
print OUTFILE @lines;
```

- filehandle predefiniti: STDIN, STDOUT, STDERR
- e.g., cat:

```
while (my $l = <STDIN>) {
  print STDOUT $l;
}
```

- e.g., cat:

```
while (<>) { print; }     # TIMTOWTDI
```

Espansioni

- al pari della shell, Perl supporta l'espansione delle variabili ed altre forme di espansione
  - espansione dei comandi

```
my $ls_output = `ls`;
```

  - espansione delle wildcard

```
my @sources = <*.c>;
```

Test su file

- la libreria standard mette a disposizione funzioni equivalenti ai predicati verificabili su file con il comando interno test delle shell
- i nomi delle funzioni corrispondono alle opzioni di test
  - e.g.

```
print “aaaargh” unless -f “/etc/passwd”;
die “can't read shadow” unless -r “/etc/shadow”;
mkdir $my_dir if ! -d $my_dir;
```

Altri assaggi di libreria standard

- funzioni numeriche
  - abs, cos/sin/tan, exp, hex/oct, log, rand, sqrt, ...
- I/O
  - open, close, dbmopen, dbmclose, getc, select, readdir, ...
- filesystem
  - chdir, chmod, chown, mkdir, link, unlink, rename, ...
- gestione dei processi
  - system, fork, kill, pipe, wait, ...
- utenza
  - getpwent, getlogin, getpwuid, ...
- networking
  - accept, bind, connect, socket, ...

Espressioni regolari

- il linguaggio delle espressioni regolari permette di definire linguaggi (i.e. pattern) su stringhe
- Perl permette di utilizzare le espressioni regolari sia come predicati su stringhe (i.e. una stringa verifica un pattern) sia come meccanismo per estrarre sottostringhe
  - esistono molteplici tool nei sistemi GNU/Linux che permettono di utilizzare espressioni regolari con sintassi diverse. La sintassi Perl è una delle più diffuse
- riferimenti:
  - man perlrequick (regular expression quick start)
  - man perlretut (tutorial)
  - man perlre (reference manual)

Regexp – sintassi (1/4)

- sintatticamente una espressione regolare (regexp) è una stringa
  - e.g. “ciao” è una espressione regolare
- i caratteri che compongono una regexp si distinguono in caratteri semplici e metacaratteri
  - i caratteri semplici sono pattern la cui semantica sono le stringhe che li contengono
    - e.g. la regexp “ciao” definisce il linguaggio di tutte le stringhe che la contengono: “ciao”, “foociaobar”, “foociao”, “ciaobar”, ...
  - i metacaratteri definiscono pattern più complessi
    - nella sintassi Perl, sono metacaratteri: `^ \ . $ | ( ) [ ] * + ?`
    - il quoting dei metacaratteri si ottiene precedendoli con \

Regexp – sintassi (2/4)

- . corrisponde ad un carattere qualsiasi
- ^ e $ corrispondono all'inizio ed alla fine della stringa
  - e.g. ^f.o$ insieme delle stringhe lunghe 3 caratteri, che iniziano per f e terminano per o
- | corrisponde ad un alternativa tra due espressioni regolari
- ( ) definiscono un raggruppamento
  - e.g. ^(foo)|(bar)$ insieme delle stringhe “foo” e “bar”
- [ ] definiscono un insieme di caratteri
  - e.g. ^[abcdef0123456789]$ numeri esadecimali ad 1 digit
  - è possibile utilizzare negazione [^] e range di caratteri [-]
    - e.g. ^[^aeiou][0-9]$ stringhe di lunghezza 2, che non iniziano per vocale e che contengono una cifra in seconda posizione

Regexp – sintassi (3/4)

- insiemi predefiniti di caratteri
  - \s blank
  - \d digit
  - \w word character [a-zA-Z0-9_]
  - \S non blank
  - \D non digit
  - W non word character

Regexp – sintassi (4/4)

- quantificatori
  - è possibile richiedere che una certa regexp sia ripetuta un certo numero di volte utilizzando quantificatori postfissi
    - ? zero o una volta (opzione)
    - * zero o più volte
    - + una o più volte
    - {n} n volte
    - {n,} n o più volte
    - {n,m} da n ad m volte
  - e.g.
    - \d+ una sequenza di 1 o più digit
    - aaa(bbb)? una sequenza di 3 a seguite da una sequenza opzionale di 3 b

Regexp – matching

- l'operatore Perl =~ permette di legare una stringa ad un operatore su regexp
- l'operatore più semplice su regexp è l'*operatore di match*
  m/*regexp*/*flag* (la m iniziale è opzionale)
  - se la stringa verifica il pattern definito dalla regexp ritorna un valore vero, altrimenti falso

```
if ($username =~ m/^zack$/) { ... }
```

- l'operatore di match lega inoltre alle variabili $1, $2, ... le parti di stringa corrispondenti ai raggruppamenti

```
if ($email =~ /(.+)+@(.+)/) {
  print "Username: $1\n";
  print "Domain: $2\n";
}
```

Regexp – sostituzione

- un altro operatore su regexp è l'*operatore di sostituzione* s/*regexp*/*stringa*/*flag*
  - permette di sostituire in-place le parti di una stringa che verificano una regexp con un'altra stringa che eventualmente contiene riferimenti alla stringa legata all'operatore
  - e.g.

```
my $s = “foorab”;
$s =~ s/rab/bar/;      # $s ora contiene foobar

my $s = “foo:bar”;
$s =~ s/(.*):(.*)/$2:$1/
   # $s ora contiene bar:foo
```

Regexp – flag

- gli operatori su regexp possono essere invocati specificando una sequenza di flag, vediamo 2 dei più importanti:
  - *flag i* indica di effettuare matching case insensitive

```
print "non stampa questo ..." if ("Foo" =~ /foo/);
print "... ma stampa questo" if ("Foo" =~ /foo/i);
```

  - l'operatore di sostituzione sostituisce per default solamente la prima occorrenza della espressione regolare specificata

```
my $s = "aaaa";
$s =~ s/a/b/;          # $s contiene baaa
```

  - il *flag g* indica di effettuare una *sostituzione globale*, di tutte le occorrenze

```
$s =~ s/a/b/g;          # $s contiene bbbb
```

Regexp – split

- la funzione di libreria standard split permette di dividere una stringa in un array di stringhe ottenuto spezzando la stringa di input alle occorrenze di una regexp

```
my $s = "a/very/very/very/long/relative/path";
my @pieces = split /\//, $s;
my $snd = $pieces[1]; # contiene very
```

- regexp e stringa sono opzionali e hanno come default la stringa $_ e la regexp \s+

```
while (<>) {
  my @pieces = split;
  print $pieces[2], "\n";
}
```

Regexp – delimitatori

- gli operatori su regexp sono solitamente delimitati da /
- è quindi necessario quotare il carattere / nel caso in cui compaia in una regexp
  - e.g. verificare che un path si riferisca ad un file in /etc/

```
my $path = "/etc/passwd";
if ($path =~ /^\/etc\//) { ... }
```

- per alleggerire la sintassi, è possibile cambiare il carattere utilizzato come delimitatore dell'operatore di matching

```
if ($path =~ m|^/etc/|) { ... }
```

  - diventa obbligatorio specificando esplicitamente la lettera dell'operatore
  - molto utile nel caso di regexp su path

Regexp – greedyness

- e.g. modificare la prima componente di un path

```
my $s = "/foo/bar/baz";
$s =~ s|/(.*)/(.*)|/pippo/$2|;
   # uhm ... $s contiene /pippo/baz/
   # e non /pippo/bar/baz
```

- i quantificatori sono greedy: si estendono "consumando" più caratteri possibile
  - nell'esempio, la prima occorrenza di .* ha consumato foo/bar
  - soluzioni:
    - definire regexp più precise

```
$s =~ s|/([^/]*)/(.*)|/pippo/$2|;
```

    - utilizzare quantificatori non greedy

```
$s =~ s|/(.*?)/(.*)|/pippo/$2|;
```

Subroutine

- Perl supporta la definizione e l'utilizzo di *subroutine:*
  - parti di codice che possono essere invocate passando un array di argomenti e che possono ritornare un valore
- gli argomenti passati alle subroutine sono accessibili all'interno dell'array @_
- per ritornare un valore si utilizza il costrutto return

```
sub sum {
  my ($a, $b) = @_;
  return $a + $b;
}
print sum(10,14);
```

Subroutine – shift

- è possibile accedere agli argomenti della funzione anche utilizzando shift

```
sub square {
  my $a = shift;
  return $a * $a;
}
print square(11);
```

Subroutine – prototipi

- sebbene non strettamente necessario, è consigliabile dichiarare i *prototipi* delle funzioni in modo che l'interprete possa segnalare errori di invocazione staticamente durante la precompilazione

```
sub sum($$) {
  my ($a, $b) = @_;
  return $a + $b
}
print “non stampa questo”;
print sum(10);   # errore di precompilazione
```

- riferimenti: man perlsub

Uso di moduli esterni

- la diffusione di Perl come linguaggio di programmazione general purpose è avvenuta anche grazie alla sterminata quantità di librerie (moduli) sviluppati da terzi per svolgere i compiti più disparati
  - l'insieme di questi moduli è raccolto nel Comprehensive Perl Archive Network (http://cpan.org)
  - i moduli più diffusi sono pacchettizzati per le maggiori distribuzioni
  - la procedura di installazione è standardizzata
  - esiste una sezione apposita di man (3pm) per la documentazione dei moduli Perl
- riferimenti: man perlmod

Uso di moduli esterni

- e.g. stampa dell'oggetto (Subject) del 17-esimo messaggio di posta elettronica contenuto in una mailbox il cui nome è passato come parametro, utilizzando il modulo Mail::Box

```
use Mail::Box::Manager;
my $mbox = shift || die "which mailbox?";
my $mgr = Mail::Box::Manager->new;
my $folder = $mgr->open(folder => $mbox);
print $folder->message(17)->subject;
```

- riferimenti:
  - man Mail::Box, man Mail::Message
  - http://cpan.org

Fun with Perl

- Perl poetry contest

  [da: http://www.foo.be/docs/tpj/issues/vol5_1/tpj0501-0012.html]

  Though leaves are many, the root is one;
  Through all the lying days of my youth
  I swayed my leaves and flowers in the sun;
  Now I may wither into the truth

```
while ($leaves > 1) { $root = 1;}
foreach($lyingdays{'myyouth'}) {
  sway($leaves, $flowers);
}
while ($i > $truth) { $i--; }
sub sway {
  my ($leaves, $flowers) = @_;
  die unless $^O =~ /sun/i;
}
```

Fun with Perl

- 1001 modi di scrivere “Just Another Perl Hacker”
- riferimenti:

  http://www.perlmonks.org?node_id=423988

```
#!/usr/bin/perl -w
use strict;open my ($f )  ,$0;local   $/;my $a =
 <$f>  ; $a=~s/\s+//g  ;$a=~/_{2}(.+$)/;$a=
$1;while($a=~m/(.{13})$/){unshift(@;,$1)
;$a=~s/.{13}$//;}foreach (@;
){for (my $i=0;$i<128
;$i++){print chr
($i) if (crypt
(chr($i),$_)
eq $_);}}
__END__
oyINpLQGuRwwCe.P5IVph0xPw
T8R3/s500K0YUw3dVujjlsowlo
EfUEms7OOkp3M5fH78hGOgxsJ
U406Ru7Bl6B/HI9LcpZLPXsw
jk6eGkhSThneVa2YTL/TWOm
H0HSsMn146/lpvNCANYDjV
cvRTejnJ4wtbAHzsAuira
mYhutHKecx/zwh61iLpt
h3SRAXP.lotAU1cwerQ
H.pgVvqzLOuTmagpGL
x/vApQsuo2Xxmv/.i
ZdyXFEbczWhScflJ
Ee2CPpMzZZLfH3p
wZSs5Devx6zjZc
Rmn5olZiat262
RPseOlOid/QO
ISSPLXdIANM
JTcJp2A.bt
6MSA7QlfH
IMdy97lW
16pRLKa
H5pdLD
qUtK6
R2aO
cQV
ut
o
```

Esercizio

- scrivere una utility check_shell.pl che verifichi che tutti gli utenti aventi UID > 1000 abbiano una shell valida
  - le informazioni sugli utenti devono essere lette da /etc/passwd
  - una shell è valida se:
    1. è listata in /etc/passwd
    2. corrisponde ad un file eseguibile sul filesystem
  - i commenti presenti in /etc/shells devono essere ignorati
  - le informazioni a riguardo degli utenti che non hanno una shell valida devono essere salvate su un file di output nel formato

    nome_utente spazio shell
  - il nome del file del file di output viene letto da linea di comando
  - verificare inoltre che la shell di root risieda in /bin
    - in caso contrario il programma deve terminare con un errore