Basi di programmazione in PERL

Matteo Re
http://homes.di.unimi.it/~re/
re@di.unimi.it

Dipartimento di Informatica,
Universita degli studi di Milano

Materiale didattico di supporto per il corso di **Biologia Computazionale**,
C.d.L. in Biotecnologie Industriali e Ambientali (BIA),
A.A: 2012-2013, Semestre: I

October 25, 2012

Note preliminari

Perl (Practical Extraction and Reporting Language) è un linguaggio di programmazione inizialmente sviluppato da Larry Wall nel 1987 come linguaggio di scripting per il sistema operativo UNIX. Esso, da lungo tempo, è popolare come linguaggio di programmazione per utenti che scrivono programmi per la prima volta. I motivi di tale popolarità sono molteplici ma, il più importante, è che PERL ,nonstante sia un linguaggio estremamente potente e versatile, semplifica la realizzazione di operazioni che, in altri linguaggi di programamzione, sono molto complesse da realizzare. PERL è disponibile per quasi tutti i sistemi operativi (Linux, Windows, Mac OS X e molti altri).
Un altro punto a favore di Perl è che esso è estremamente popolare e utilizzato nei più svariati ambiti (dall'amministrazione di server internet alla gestione di browser genomici, solo per citarne alcuni) e questo ha portato alla realizzazione di moltissime risorse (documentazione, esempi di programmi e molto altro) pubblicamente disponibili in internet. Di seguito vi segnalo solo alcuni dei siti dedicati a PERL che potrebbero esservi utili durante il corso:

- Perl.it (`http://www.perl.it/`)
- perl.org (`http://www.perl.org/`), il sito ufficiale del linguaggio Perl
- Perl monks (`http://www.perlmonks.com/`), sito dedicato alla programmazione in Perl a diversi livelli di complessità . Contiene diversi tutorial.

L'obiettivo del modulo di programmazione di questo corso è quello di introdurre le caratteristiche di base del linguaggio Perl. Alla fine del corso dovreste possedere tutte le nozioni necessarie per scrivere programmi moderatamente complessi, ed avere inoltre una conoscenza delle risorse pubbliche dedicate a Perl tale da consentirvi (in caso di necessità o di interesse da parte vostra) di passare alla realizzazione di progetti più complessi. La prima parte di del corso sarà dedicata a nozioni di base di programmazione che saranno utilizzate per la scrittura di programmi Perl ma che risultano valide anche in altri linguaggi di programmazione.

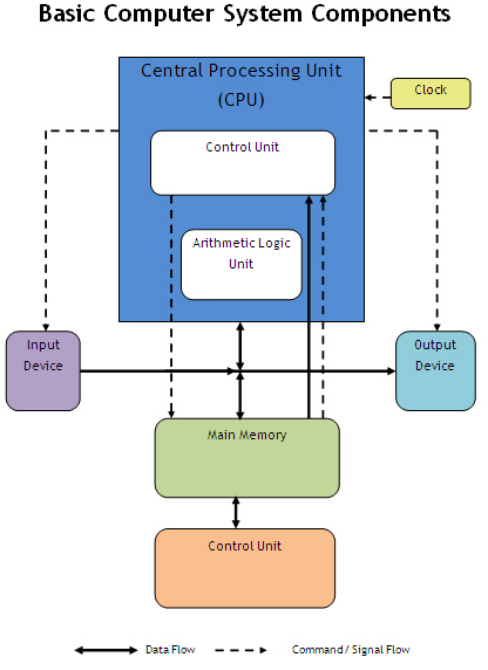
Durante il corso verranno assegnati degli esercizi di programmazione. Alcuni di essi saranno svolti insieme in classe, altri dovranno essere risolti a casa tra una lezione e la successiva. Gli esercizi a casa non sono obbligatori. Possono essere considerati come un'opportunità in più per aiutarvi a imparare a programmare (soprattutto se non avete mai programmato prima). Ad essi non è associato un punteggio che farà media per l'esame ... l'unica influenza che potranno avere sul voto finale è dovuta al fatto che considererò la loro eventuale risoluzione come un indicatore della vostra partecipazione attiva al corso. Tenete comunque presente il fatto che l'esame verrà realizzato mediante un **progetto che comporta la scrittura di un programma in Perl**. So per

esperienza, dalle edizioni precedenti del corso, che gli studenti che risolvono gli esercizi di programmazione non hanno problemi a superare l'esame. Quindi il mio consiglio è il seguente: armatevi di pazienza, provate a risolvere gli esercizi, non preoccupatevi se non riuscite a risolverli al primo tentativo e, soprattutto, **in caso di dubbi inviatemi una email** (il mio indirizzo è nella prima pagina di questa dispensa). Non rimarrete senza una risposta.

Detto questo possiamo iniziare.

Introduzione

Il linguaggio Perl è stato creato da Larry Wall nel 1987. Perl è stato progettato come *linguaggio di alto livello, procedurale ed interpretato*. Recentemente è stata aggiunta la possibilità di scrivere codice Perl seguendo un paradigma di programmazione detto 'programamzione ad oggetti' (Object Oriented Programming, OOP) ma Perl non è il miglior linguaggio disponibile per programamre ad oggetti. Ne esistono altri che sono più indicati per questo tipo di programmazione (ad es. C++ o Java che sono stati concepiti per supportare la OOP fin dalle prime fasi di sviluppo). Per ora non preoccupatevi riguardo ai paradigmi di programmazione. Essi verranno introdotti più avanti.



Di per sè un calcolatore non è molto "intelligente". Sostanzialmente un calcolatore è composto da una serie di interruttori elettronici che possono assumere due stati: spento o acceso (oppure 0 e 1 se preferite). Impostando diverse combinazioni di interruttori potete indurre il calcolatore a compiere delle operazioni quali, ad esempio, mostrare qualcosa sullo schermo o emettere un suono. Al livello più semplice possibile la programmazione è proprio questo: *dire ad un computer cosa deve fare*. Dato che il numero degli interruttori è molto elevato, sarebbe davvero scomodo impostare

il loro stato uno alla volta. Ed è proprio qui che entrano in gioco i *linguaggi di programmazione*. In questo corso eviteremo di discutere di architettura dei calcolatori e ci concentreremo sui linguaggi di programmazione (in particolare Perl). Nonstante questo è necessaria una quota minima di informazioni che vi saranno utili per immaginare cosa succede dietro le quinte mentre un computer esegue delle operazioni.
Nella pagina precesente potete vedere uno schema semplificato dell'architettura di un calcolatore. Senza addentrarci nei dettagli il particolare su cui voglio portare la vostra attenzione sono due componenti: la *Central Processing Unit, o CPU* (altrimenti detta Processore) in blu e la *Main Memory* (o memoria principale, altrimenti detta RAM o *Random Access Memory*, memoria ad accesso casuale). Tralasciamo i rettangoli all'interno della CPU. Ognuno dei componenti del calcolatore svolge una specifica funzione. In particolare:

- **CPU**: il suo compito è quello di effettuare le operazioni di calcolo. Non è in grado di immagazzinare informazioni se non in quantità estremamente limitata.
- **RAM**: il suo ruolo è quello di immagazzinare informazioni in modo che esse possano essere lette e scritte in maniera veloce fornendo, al contempo, una quantità di spazio per le informazioni maggiore rispetto a quella disponibile nella CPU.

Un punto cruciale da notare è che esistono due componenti distinte per immagazzinare dati (RAM) e per elaborarle (CPU). La memoria disponibile sottoforma di RAM viene continuamente scritta, letta ed, eventualmente, cancellata. I dati scritti in RAM verranno elaborati (dalla CPU) producendo risultati che verranno scritti in RAM (e poi, eventualmente, su disco). Quindi è lecito aspettarsi che ogni linguaggio di programmazione sia in grado di effettuare almeno due tipi di operazione: manipolazione della memoria RAM (lettura e scrittura di dati) ed elaborazione di dati (ad es. somma e sottrazione).
Oltre a CPU e RAM esistono anche altri tipi di dispositivi, in particolare dispositivi di INPUT (ad es. tastiera e mouse) e dispositivi di output (ad es. lo schermo).

Il concetto (informale) di algoritmo

Un **algoritmo** si può definire come un *procedimento* che consente di ottenere un risultato eseguendo, **in un determinato ordine** un insieme di passi semplici corrispondenti ad azioni scelte solitamente da un insieme finito. Alcuni esempi sono:

- Procedura per calcolare il minimo comune multiplo tra due numeri naturali.
- Una ricetta.
- Procedura per ordinare un insieme di oggetti in base ad un determinato criterio.

Già dalla definizione "informale" di algoritmo che trovate all'inizio di questa sezione emergono alcuni aspetti fondamentali. Innanzitutto si parla di una sequenza di passi (o istruzioni, dato che ogni passo dice al calcolatore di eseguire un'operazione) le quali, eseguite in un certo ordine, realizzano una determinata operazione. Questo vuol dire che, se si esclude il caso di operazioni davvero semplici, scrivere un programma richiede **prima della stesura del codice**, una parte preliminare di analisi del problema in cui l'obiettivo fondamentale è quello di suddividere il problema in sottoproblemi più semplici. Inoltre un altro aspetto fondamentale è che, una volta determinati i passi da eseguire, dobbiamo anche decidere l'ordine in cui eseguirli poichè la medesima "collezione" di istruzioni eseguita cambiando l'ordine delle istruzioni produce risultati diversi. Un altro aspetto importante che è presente (anche se in modo implicito) nella definizione di algoritmo che stiamo utilizzando è che ogni linguaggio di programmazione contiene una serie di istruzioni (parole chiave) molto semplici. Il compito del programmatore è quello di spezzare un problema in parti più piccole che si possono risolvere utilizzando le istruzioni messe a disposizione dal linguaggio. Nonostante il fatto che la nostra definizione di algoritmo sia abbastanza elastica non tutte le sequenze di istruzioni possono essere definite algoritmi. In particolare gli algoritmi hanno alcune caratteristiche fondamentali:

- La sequenza di istruzioni deve essere finita (*finitezza*).
- La procedura deve portare ad un risultato (*effettività*).
- Le istruzioni devono essere eseguibili materialmente (*fattibilità*).
- Le istruzioni devono essere espresse in modo non ambiguo (*non ambiguità*).

Nella prossima sezione inizieremo a considerare i diversi linguaggi di programmazione. tenete presente che ne esistono moltissimi e che sono stati creati in epoche diverse. Ognuno ha i propri pregi e difetti, alcuni sono adatti per il calcolo intensivo

e sfruttano le caratteristiche dei moderni supercalcolatori, altri sono stati sviluppati per sfruttare al massimo le caratteristiche del vostro smartphone. Ad ogni modo, indipendentemente dal linguaggio di programmazione considerato, le istruzioni saranno eseguite da una CPU che costituisce il livello più basso al quale una machcina esegue dei calcoli. Questo comporta alcuni problemi fondamentali. Uno di essi è che ogni tipo di CPU ha il suo set interno di istruzioni e, quindi, se un programmatore utilizzasse per programmare direttamente le istruzioni della CPU sarebbe costretto a riscrivere lo stesso programma per ogni tipo esistente di CPU. Decisamente una perdita di tempo (per non parlare della necessità di imparare tutti i set di istruzioni di tutti i tipi di CPU). Ovviamente il problema è stato risolto. Vedremo come nella prossima sezione.

Linguaggi di programmazione

I linguaggi di programmazione sono strumenti per comunicare ad una macchina come risolvere un problema. Sono, quindi, strumenti di comunicazione uomo-macchina. Essi permettono di rappresentare un **programma** sottoforma di **algoritmo + strutture dati** comprensibili da una macchina. Essi sono analoghi ai linguaggi naturale, con la differenza che sono utilizzati per comunicare con un calcolatore. Come i linguaggi naturali sono caratterizzati dalle seguenti componenti:

- Insieme di simboli (alfabeto) e di parole (dizionario) che possono essere utilizzati per creare le frasi del linguaggio.
- Insieme delle regole grammaticali (sintassi) per definire le frasi corrette composte dalle parole del linguaggio.
- Significato (semantica) delle frasi del linguaggio
- Per utilizzare correttamente un linguaggio è necessario conoscerne la pragmatica (ad es. quali frasi è opportuno utilizzare a seconda del contesto).

La differenza più evidente tra un linguaggio naturale ed un linguaggio di programmazione è che quest'ultimo **non può essere ambiguo** e quindi *deve* essere dfinito in maniera inequivocabile. In caso di ambiguità la macchina si blocca ed l'esecuzione del programma viene interrotta.

Linguaggi a basso e ad alto livello

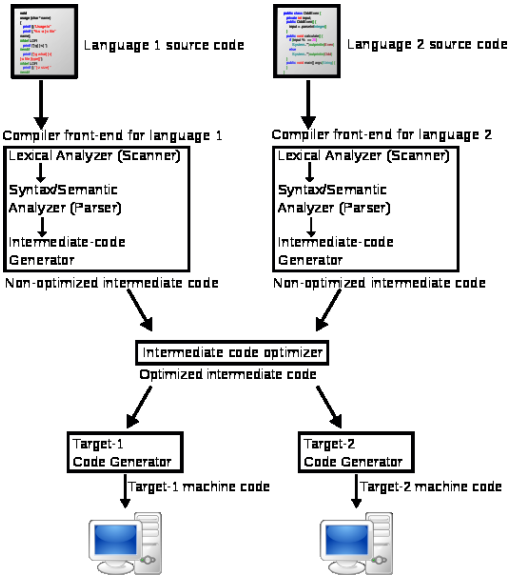
Per **linguaggio a basso livello** si intende il sottogruppo di linguaggi di programmazione orientati alla macchina, al contrario dei linguaggi di programmazione ad alto livello che sono invece orientati all'utente.
Tali linguaggi utilizzano istruzioni estremamente basilari che vengono elaborate direttamente dal processore e permettono un totale accesso alle risorse della macchina. I programmi scritti con questi linguaggi non lasciano niente di sottinteso ma esplicitano ogni istruzione fino all'essenziale, di conseguenza risultano estremamente efficienti in termini di velocità di elaborazione.
Dal momento che le istruzioni sono molto semplici, per raggiungere un buon grado di astrazione (ossia per scrivere programmi che eseguano operazioni più complesse di somme, sottrazioni ecc.) è necessaria una grossa mole di istruzioni, che rendono il programma molto prolisso (più lungo in termine di righe di codice) e di difficile comprensione per un programmatore.

Un **linguaggio di programmazione ad alto livello** è un linguaggio di programmazione diverso dal linguaggio macchina direttamente eseguibile da un computer, ma più vicino o familiare alla logica del nostro linguaggio naturale. L'idea di fondo è che i programmi ad alto livello possono essere ricondotti a programmi in linguaggio macchina in modo automatico, ovvero da un altro programma. Il linguaggio ad alto livello quindi è indipendente dalle caratteristiche fisiche della macchina in cui si opera, per poter essere eseguito, deve prima essere elaborato da un altro programma che lo tradurrà in istruzioni in codice macchina. Quest'idea fu introdotta in informatica negli anni cinquanta, soprattutto grazie al lavoro di John Backus presso la IBM, dove fu sviluppato il primo compilatore per il linguaggio FORTRAN. In seguito, Backus ricevette per questo motivo il premio Turing.
Esistono molti programmi in grado di rendere eseguibile del codice di alto livello ma essi possono essere suddivisi principalmente in due categorie:

- **Compilatori**: Sono programmi (o insiemi di programmi) che traducono codice scritto in un linguaggio di programmazione in un file eseguibile. Ne esistono molti tipi e, dato che il loro obiettivo è produrre codice eseguibile su vari tipi di CPU, ne esistono versioni in grado di tradurre codice scritto nel medesimo linguaggio di programmazione producendo codice specifico per diversi tipi di CPU. Il codice prodotto, essendo ottimizzato per la CPU sulla quale dovrà essere eseguito, è estremamente efficiente.

- **Interpreti**: Sono programmi che non producono un file eseguibile ma accettano in input un file contenente il codice sorgente (in formato testuale) e lo eseguono interfacciandosi con le componenti di basso livello della macchina (ad es. la CPU). In questo caso il file eseguibile non è in nostro programma ma l'interprete stesso al quale invieremo il codice che abbiamo scritto.

Di seguito sono riportati alcuni esempi di linguaggi di programmazione a basso e ad alto livello. Per ogni linguaggio verrà specificato se si tratta di un linguaggio a basso o ad alto livello. A causa del fatto che i programmi scritti in linguaggi basati sull'utilizzo di un interprete non possono essere eseguiti direttamente, molto spesso questi linguaggi vengono definiti linguaggi "interpretati".



Linguaggio macchina (linguaggio a basso livello)

Abbiamo detto che il livello più basso della macchina è la CPU. La CPU è comprende solo serie di zeri e uno. Ogni CPU ha il suo specifico set di istruzioni in linguaggio macchina. Come esempio riporterò il codice macchina di una CPU con architettura MIPS (Microprocessor without Interlocked Pipeline Stages). Questo tipo di CPU si trova, tra l'altro, nelle console Sony PlayStation, Sony PlayStation 2 e Sony PlayStation Portable. Le istruzioni di questa architettura sono tutte composte da 32 bit (o 4 byte). I primi 6 bit (ogni bit corrisponde ad un singolo valore: 0 o 1) contengono il codice operativo. Le istruzioni di tipo J (da jump: salto) ed I (immediate) sono completamente specificate dal campo op mentre le istruzioni di tipo R (registro) comprendono un campo addizionale chiamato func che codifica la specifica funzione da eseguire. Il formato dettagliato delle istruzioni è il seguente:

```
   6      5     5     5     5      6 bit
[  op  |  rs |  rt | address/immediate]  tipo I
[  op  |       target address        ]  tipo J
[  op  |  rs |  rt |  rd |shamt| funct]  tipo R
```

rs, rt, e rd indicano i registri nei quali si trovano gli operandi; shamt sta per "shift amount" mentre address e immediate contengono direttamente degli operandi.

Per esempio l'operazione di somma dei registri 1 e 2 con memorizzazione del risultato nel registro 6 è codificata come:

```
[  op  |  rs |  rt |  rd |shamt| funct]
    0     1     2     6     0     32     forma decimale
 000000 00001 00010 00110 00000 100000   forma binaria
```

L'istruzione finale da inviare alla CPU, quindi, è la seguente:
00000000001000100011000000100000

Abbastanza scomodo da leggere e scrivere! Inoltre le regole per scrivere queste serie di 0 e 1 cambiano da CPU a CPU.

Linguaggio Assembly (linguaggio a basso livello)

Il linguaggio Assembly è la forma *simbolica* del linguaggio macchina. E' quindi un gradino più in alto del linguaggio macchina ma è comunque un linguaggio di basso livello. In questo esempio vediamo (su una architettura diversa da MIPS) un altro esempio di somma di due numeri. In linguaggio macchina :

```
00000010101111001010
00000010111111001000
00000011001110101000
```

e in linguaggio Assembly:

```
LOAD A
ADD B
STORE S
```

Lo scopo del linguaggio Assembly è quello di permettere ai programmatori di sfruttare i set di comandi delle CPU senza utilizzare direttamente il linguaggio macchina (difficile da leggere, scrivere e, soprattutto, ricordare).

Linguaggio Perl (linguaggio ad alto livello)

Proviamo a vedere come si può risolvere il problema della somma di due numeri (e la scrittura del risultato sullo schermo) in un linguaggio di alto livello (non a caso vediamo un esempio in linguaggio Perl dato che Perl sarà il linguaggio che utilizzeremo per imparare a programmare).
Prima di passare all'esempio di codice vero e proprio dobbiamo tenere presente che Perl è un linguaggio ad alto livello interpretato. Sappiamo già che questo vuol dire che dovremo scrivere un file di testo contenente il codice Perl e poi passarlo in input all'interprete *perl* per farlo eseguire. Nel resto della dispensa userò la parola **Perl** per indicare il linguaggio e la parola **perl** per indicare l'interprete. I passi necessari per scrivere ed eseguire un programma Perl sono sempre gli stessi indipendentemente dal programma che vogliamo realizzare:

- Aprire un editor di testo (meglio se specializzato per la programmazione, ad esempio notepad++).
- Scrivere il codice Perl.
- Salvare il file di testo dando un nome semplice da ricordare e che **NON CONTENGA SPAZI O CARATTERI STRANI, se dovete usare più di una parola usate il trattino basso** _ . Importante: fate attenzione a identificare con precisione la posizione in cui avete salvato il file (ad es. cartella codicePerl sul Desktop).
- Aprite un terminale (linea di comando). Quello che in Windows di solito trovate in Programmi $\rightarrow$ Accessori $\rightarrow$ Prompt dei comandi
- Posizionatevi nella cartella in cui avete salvato il file contenente il codice Perl. Per farlo utilizzate il comando **cd** che vuol dire 'change directory'. Esempio: se avete salvato il file contenente il codice Perl nella cartella (directory) codicePerl sul Desktop scrivete cd Desktop (INVIO) cd codicePerl (INVIO). Se volete essere sicuri di essere nel posto giusto scrivete **dir**. Questo comando stamperà il contenuto della directory corrente (e dovreste vedere il nome del vostro file).
- Ora invocate l'interprete passandogli come input il file con le istruzioni Perl: **perl nomedelvostrofile (INVIO)**. E il file sarà interpretato ed eseguito.

Il motivo per cui ci spostiamo della cartella contenente il file con le istruzioni Perl è che, in caso contrario, saremmo obbligati a specificare il percorso del file per intero, invece di scrivere solo il suo nome, e questo potrebbe essere molto scomodo (e lungo) da scrivere. Inoltre, come vedremo in seguito, quando salveremo l'output del programma su file, il file verrà salvato nella cartella corrente (ed eviteremo di spargere i file per il calcolatore con il risultato che poi dovremmo andare a cercarli).
Ora passiamo all'esempio di istruzioni Perl per sommare due numeri e stampare il risultato (quanto riportato di seguito è quello che dovreste scrivere nel file da passare come input all'interprete perl):

```
$a=2;
$b=3;
$c=$a+$b;
print $c;
```

Tutto qui. Come potete vedere il nostro esempio di programma (molto semplice) è decisamente più amichevole se scritto in Perl piuttosto che in codice macchina o in Assembly.

Attenzione. Ogni programma scritto in Perl usa simboli speciali (sono sicuro che avete già notato la presenza del simbolo $ nell'esempio che abbiamo appena visto). I caratteri speciali che utilizzeremo in modo ricorrente sono i seguenti:

```
$ @ # . % { } [ ] ; ~
```

Il più strano è indubbiamente l'ultimo della lista (tilde).

Definizione della soluzione di un problema e trasformazione della soluzione in codice

So per esperienza dai corsi precedenti che l'ostacolo più difficile da superare quando si impara a programmare non è, al contrario di quanto si è portati a credere, la necessità di imparare molte istruzioni (o parole chiave) del linguaggio ma è la *difficoltà di immaginare la soluzione di un problema in un modo che possa essere utilizzato per comunicarla ad una macchina*. Le parole chiave da che utilizzeremo sono relativamente poche (circa 20 o 30) la difficoltà starà nell'utilizzarle (trovare il modo di combinarle) per fare cose relativamente complesse. Inizierò quindi il corso di programmazione soffermandomi non tanto sul linguaggio ma sulle tecniche necessarie per costruire una soluzione ad un problema generico. Quello che vedremo in questa parte del corso si applica direttamente a tutti i linguaggi di programmazione (non è specifico del Perl).

Analisi preliminare del problema

Scrivere programmi equivale, fondamentalmente, a trovare la soluzione ad un problema e renderla comprensibile ad una macchina in modo che quest'ultima possa risolvere il problema in modo automatico. Appare evidente come il primo passo per la stesura di un programma non possa essere quello della scrittura del codice ma debba essere quello della definizione del problema da risolvere.

ATTENZIONE: Scrivere un (qualsiasi) programma sarà molto più semplice se avrete dedicato un pò di tempo a ragionare sul problema da risolvere.

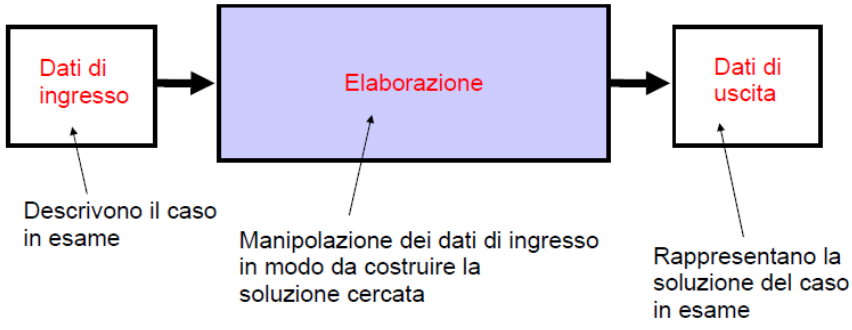
Per scrivere un algoritmo bisogna avere gà in mente la soluzione complessiva del problema da affrontare altrimenti se vi metterete a scrivere codice e, contemporaneamente, a creare la soluzione del problema sarete di sicuro portati a dare più peso alla scrittura del codice piuttosto che a progettare la soluzione ... con il risultato che, magari, alla fine il programma potrà anche essere eseguito dalla macchina ma potrebbe non risolvere il problema di partenza. Esistono infatti diversi tipi di errore che possiamo commettere durante la stesura dei programmi. In particolare possiamo definire due grandi classi di errori:

- Errori di **sintassi**: sono gli errori che impediscono all'interprete (o al compilatore) di rendere il codice eseguibile e di eseguirlo. In presenza di un errore di sintassi l'esecuzione del programma si blocca (se il linguaggio è interpretato) o non riesce ad essere compilato (se usiamo un compilatore). In genere questi errori sono abbastanza innocui (perchè sono evidenti dato che causano un blocco). Basta correggere il testo del programma e tutto va a posto. Generalmente all'inizio si tende ad inserire più errori di sintassi nei programmi poi, con l'esercizio, questi calano. Ma tendono a non sparire mai del tutto. Chi scrive crea programmi da

anni e programma quasi tutti i giorni ... eppure ogni tanto un qualche errore di sintassi ci scappa lo stesso. Quindi non preoccupatevi se un errore vi blocca il programma. E' normale. Soprattutto all'inizio.

- Errori **logici**: Sono i più insidiosi. Il programma (o la sua compilazione) non si bloccano e quindi il programma si comporta in modo inatteso oppure causa un errore solo durante l'esecuzione (all'interno della CPU). Sono i più difficili da risolvere perchè può non essere immediatamente evidente da quale punto del programma derivano.

Vediamo uno schema generale del processo di soluzione di un problema:



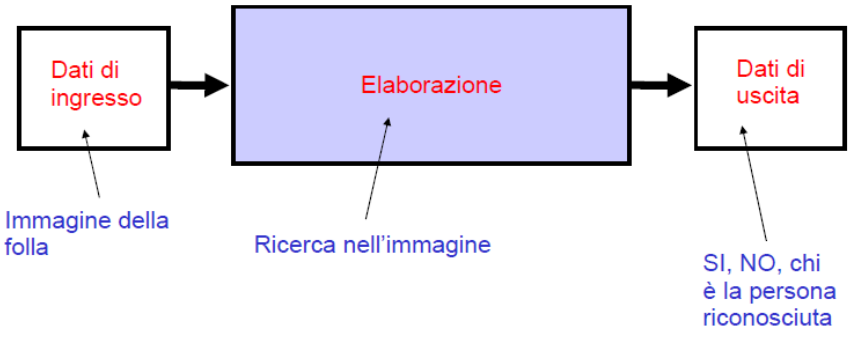
Già da questo schema preliminare appare evidente come alcune parti siano costanti (il che aiuta). Mi riferisco ai dati di input e ai dati di output. La soluzione del problema è quella che permette di trasformare i dati di input nei dati di output. Ne deriva che, prima di iniziare a sviluppare una soluzione dobbiamo avere ben presente i dati a disposizione (questa è la parte facile) e i dati di output (questo è più difficile). La difficoltà nell'inquadrare i dati di output deriva dal fatto che, di solito, il problema è descritto in forma di testo. Ad esempio:

"Dato l'ammontare totale di un prestito e tenuto presente che se tale valore è maggiore o uguale a 100 Euro gli interessi saranno del 10% altrimenti saranno del 6%, calcolare l'importo totale che dovrà essere restituito."

E' ovvio che il dato di input è un valore corrispondente all'importo del prestito ma cosa possiamo dire del dato di output? Sulla base del testo che descrive il problema possiamo dire che il dato di output:

- E' un valore numerico (come il dato di input)
- E' legato al dato di input dal concetto di "interessi" (esistono diversi livelli di interessi)
- Per calcolare il dato di output dobbiamo utilizzare sia il dato di input che i valori dei livelli di interesse (6% o 10%). Questa fase costituisce la **trasformazione** del dato di input in dato di output.

Appare evidente che qualsiasi soluzione che **non utilizza** i dati di input **non può** costituire una soluzione del problema considerato (e descritto dalla frase riportata sopra). Vediamo un altro esempio di problema:
"Data un'immagine di una folla di persone riconoscere una specifica persona."



Una considerazione: essere capaci di risolvere un problema non significa essere capaci di spiegare esattamente come questo avviene. Questi sono dettagli che entreranno in gioco dopo la definizione della soluzione e durante la sua trasformazione in algoritmo e successiva traduzione in programma.

I problemi giocattolo che abbiamo appena visto permettono di capire che non è consigliabile costruire le soluzioni utilizzando direttamente il linguaggio di programmazione che dovremo utilizzare per automatizzarle. Serve qualcosa di più agile per permetterci di giocare a costruire soluzioni in modo veloce ma che, al contempo, sia abbastanza ordinato da poter essere tradotto in codice e che permetta di verificare in fretta se la logica della soluzione regge o se è il caso di provarne un'altra prima di metterci a programmare. La soluzione è l'utilizzo dello **pseudocodice** che vedremo in dettaglio nella prossima sezione di questa dispensa.

Pseudocodice, paradigmi di programmazione e formalizzazione di algoritmi

Prima di vedere da vicino lo pseudocodice fermiamoci a considerare un fatto che, ormai, dovrebbe essere chiaro: la scrittura di codice è solo una parte del processo di realizzazione di programma. Infatti la produzione di software è un processo che comporta una fase di analisi del problema e lo sviluppo di possibili soluzioni. Questo ha almeno due conseguenze fondamentali dal punto di vista di un programmatore:

- Per il medesimo problema **possono esistere più soluzioni**. Infatti sarà del tutto normale per me ricevere soluzioni diverse agli esercizi che dovrete svolgere durante il corso. Quello che è importante non è il codice che utilizzerete per risolvere i problemi ma la logica che sarà alla base di quel codice.
- Dato che esistono diverse soluzioni per ogni problema esisteranno diversi tipi di "tecniche" (o, se preferite, scuole di pensiero o **paradigmi** di programamzione ...) per scrivere programmi. Ognuna ha i suoi pregi ed i suoi difetti. Tra i vari paradigmi i due fondamentali sono quello della **programmazione procedurale** e quello della **programmazione ad oggetti**.

In questo corso ci occuperemo principalmente di programmazione procedurale. In questo paradigma di programmazione l'assunzione fondamentale è che ogni problema può essere risolto tramite una sequenza di operazioni (o procedura). Questo è il paradigma di programmazione classico e si concentra fondamentalmente su **COME RISOLVERE UN PROBLEMA**. L'altro paradigma (programmazione orientata agli oggetti o Object Oriented Programming, OOP) è più recente e si concentra su **QUALI ELEMENTI (OGGETTI) FANNO PARTE DEL PROBLEMA** e sulle loro interazioni. Scrivere programmi orientati agli oggetti è molto più complesso che scrivere programmi

procedurali. Il vantaggio è che l'astrazione che si può raggiungere in questo tipo di programmi è maggiore e leggere il codice di un programma si avvicina molto a leggere un normale testo. Inoltre OOP permette di rendere più facile la manutenzione del codice nel tempo, soprattutto nel caso di programmi complessi. Il nostro scopo, comunque, è imparare a programmare da zero. Quindi utilizzeremo la programmazione procedurale. Fatta questa precisazione possiamo occuparci di capire cosa si intende per pseudocodice e come utilizzarlo.

Pseudocodice

Lo pseudocodice è una delle rappresentazioni più utilizzate per gli algoritmi perchè è facile da scrivere, permette al programmatore di concentrarsi sulla logica di soluzione dei problemi, permette l'utilizzo di parole del linguagigo parlato. Per provare ad utilizzare lo pseudocodice dobbiamo definire un set di convenzioni per la sua scrittura (in modo che pseudocodici scritti da persone diverse possano essere letti da tutti) e dobbiamo definire il set di operazioni di base che possono essere svolte da un computer. Le convenzioni di scrittura dello pseudocodice sono le seguenti:

- Le frasi possono essere scritte in un qualsiasi linguaggio umano (ad es. inglese, ma anche in italiano va bene.)
- ogni istruzione è scritta su una riga
- Combinazioni di parole chiave ed indentazioni sono utilizzate per rappresentare raggruppamenti di istruzioni o cicli (vedremo degli esempi in classe alla lavagna).
- Ogni serie di istruzioni è scritta dall'alto verso il basso he ha **un unico punto di ingresso (input) ed un unico punto di uscita (output)**

Ora passiamo a definire le operazioni di base che possono essere eseguite da un computer.

1. **Leggi (o Get** / **Read)** : un computer può acquisire dati e può farlo in diversi modi. Noi possiamo usare un generico 'Leggi' per indicare questa operazione. In inglese a volte si utilizza Get per indicare 'leggi da tastiera' e Read per indicare la lettura da file.
2. **Scrivi (o Print)** : anche in questo caso i modi di scrivere sono diversi, poichè un computer può scrivere su file o sullo schermo.
3. **+ - *** / : un computer può, ovviamente, fare dei calcoli aritmetici.
4. **Assegnamento** : un computer può assegnare un valore ad una variabile (posizione della RAM). Il simbolo che utilizzeremo per questa operazione è $\leftarrow$. IMPORTANTE: per assegnare un valore ad una specifica variabile dobbiamo specificarne il nome. Se vogliamo assegnare il valore 3 alla variabile a allora scriveremo: a $\leftarrow$ 3 . Se vogliamo indicare che la variabile viene scritta su disco potremo scrivere 'Salva nomevariabile'.
5. **Selezione** : Un computer può confrontare due valori, effettuare un test e scegliere di eseguire una tra due azioni alternative (mutualmente esclusive). Questo potrà essere indicato nel codice utilizzando una struttura a 3 componenti: **SE** ... **ALLORA** ... **ALTRIMENTI**. Al posto dei puntini potremo mettere altre istruzioni e il test andrà specificato da parte alla parola SE .

6. **Ripetizione** : un computer è in grado di ripetere un insieme istruzioni (l'insieme può anche contenere una sola istruzione). Indicheremo la ripetizione come ESEGUIFINCHE' test ... . Il set di istruzioni andrà scritto al posto di ... mentre il test serve a permettere al calcolatore di capire quando deve terminare la ripetizione delle istruzioni, altrimenti non si fermerebbe mai e otterremmo quello che viene definito *ciclo infinito*. Esiste un altro modo di limitare la ripetizione di set di istruzioni. Invece di ripetere finchè una data condizione è vera possiamo eseguire una medesima istruzione ' PEROGNI elemento IN collezione_di_elementi ESEGUI ...'. Ovviamente, finiti gli elementi della collezione termina anche la ripetizione.

Un punto estremamente importante è quello di scegliere dei nomi sensati per le variabili. Lo scopo di una variabile dovrebbe essere chiaro dato il suo nome. Invece di usare a, b ecc. sono consigliabili nomi più facili da interpretare. Se dobbiamo usare più di una parola separiamo le parole usando un _ come, ad esempio, in importo_iniziale_prestito $\leftarrow$ 1200 .

La nostra assunzione da questo punto in poi è che ogni possibile programma può essere scritto in pseudocodice utilizzando solamente tre tipi di strutture di controllo del flusso di esecuzione: **Sequenza, Selezione e Ripetizione**. Tra le tre la più semplice è la Sequenza che rappresenta una serie di passi eseguiti uno dopo l'altro nell'ordine in cui vengono inseriti nello pseudocodice. Essa rappresenta un flusso di esecuzione senza decisioni o ripetizioni.

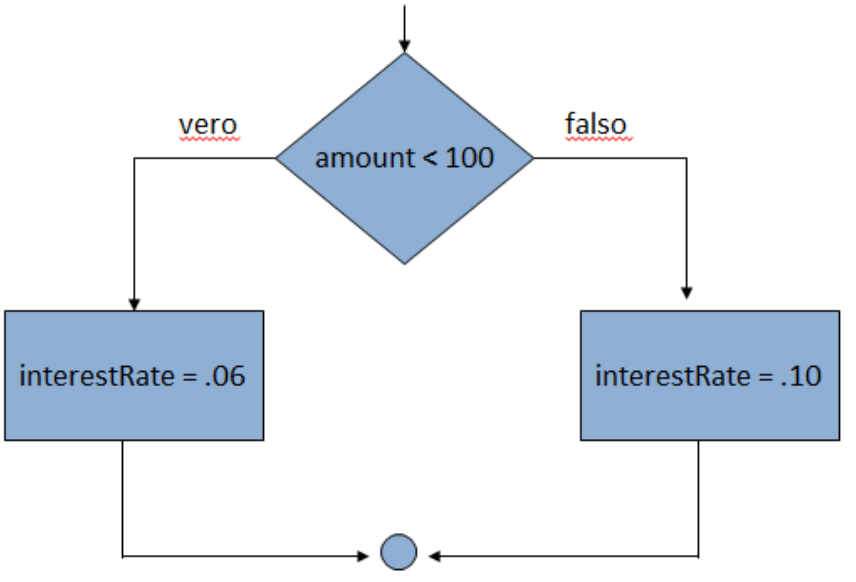
Prima di passare agli esempi pratici vediamo cosa si intende per **indentazione** : L'indentazione è un modo semplice di indicare visivamente i raggruppamenti di blocchi di codice in un file di testo. Si basa essenzialmente sulla distanza del primo carattere della riga di testo dal margine sinistro del file. Un modo molto semplice di indentare è quello di usare delle tabulazioni (tasto TAB della tastiera). L'obiettivo fondamentale è quello di far capire quale struttura di controllo governa l'esecuzione di un insieme di istruzioni. Le regole rispetto ai tre tipi di controllo di flusso sono le seguenti:

- **Sequenza**: tutte le istruzioni appartenenti ad una sequenza devono iniziare nella stessa colonna del file di testo.
- **Selezione** : tutte le istruzioni che cadono all'interno dei blocchi di decisione (ALLORA ... ALTRIMENTI ...) vanno inddentate (premete una volta il tasto TAB) non vanno indentate, invece, le righe che iniziano con SE, ALLORA, ALTRIMENTI, ossia le righe che definiscono la Selezione.
- **RIpetizione** : tutte le istruzioni ripetute vanno indentate (premete una volta il tasto TAB), la riga che inizia con *ESEGUIFINCHE' test* oppure con '*PEROGNI elemento IN collezione_di_elementi ESEGUI*' non va indentata.

E' anche possibile chiudere le sequenze di controllo con parole che ne indicano la fine. Ad esempio per indicare la fine di un SE potremo scrivere un FINESE. In ogni caso queste 'parole di chiusura' devono essere incolonnate con la parola iniziale (quindi, ad esempio, FINESE deve essere nella stessa colonna del suo SE). Ricordatevi di resistere alla tentazione di inserire parti di codice scritte nel linguaggio che utilizzerete per scriver eil programma ... non dimenticatevi che, a questo punto, non state programmando ma state definendo un piano logico per risolvere un problema!

Ora possiamo iniziare a vedere degli esempi di pseudocodice (non vediamo Sequenza dato che si scrive come una serie di istruzioni che iniziano tutte alla prima colonna del file di testo).

PSEUDOCODICE Selezione :



Pseudocodice ➔

```
SE amount < 100
        interestRate = .06
ALTRIMENTI
        Interest Rate = .10
FINE_SE
```

In un **diagramma di flusso**, come quello riportato in questa pagina, una delle cose che creano più confusione è che sia nel caso della Selezione che in quello della Ripetizione viene utilizzato come simbolo grafico un rombo.

DOMANDA 1:
Riuscite ad identificarlo e a dire a quale parte della struttura di Selezione corrisponde? Cioè se indica il test logico, il blocco allora o il blocco altrimenti?

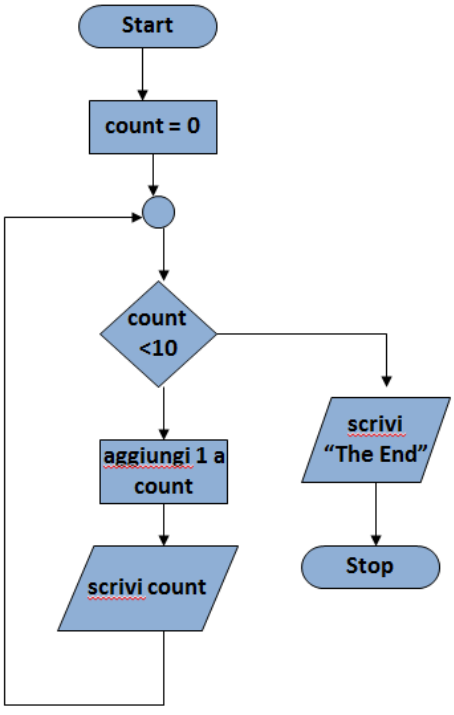
DOMANDA 2:
Una volta identificato cercate di spiegare come mai il rombo è presente sia nella rappresentazione della Selezione che in quella della Ripetizione.

Un suggerimento: ricordatevi che il diagramma di flusso (proprio come lo pseudocodice) rappresenta un piano logico per controllare tutti i possibili flussi di esecuzione del programma. Provate ad immaginare un problema che potrebbe verificarsi

in un blocco di Ripetizione (ne abbiamo già parlato prima ...). La soluzione a questo problema comporta l'utilizzo di un ... Basta così altrimenti scrivo la risposta.

PSEUDOCODICE Ripetizione :

Graficamente l'inizio della ripetizione si può rappresentare con un cerchio come nel seguente diagramma di flusso: come nell'esempio precedente trovate anche lo pseudocodice corrispondente al diagramma di flusso riportato.



PSEUDOCODICE:

```
count = 0
FINCHE' count < 10
        aggiungi 1 a count
        scrivi count
FINEFINCHE'
scrivi "The End"
```

DOMANDA 3:
Riuscite a vedere il ciclo di ripetizione nel diagramma di flusso? Quale parte del diagramma rappresenta la ripetizione?

DOMANDA 4:
Cambierebbe qualcosa nella logica del progamma se la prima istruzione posta **all'interno del blocco di Ripetizione** (aggiungi 1 a count) fosse sostituita con 'sottrai uno a count' ? Motivate la risposta.

Questo conclude la nostra presentazione veloce dello pseudocodice. Naturalmente ci sarebbe ancora molto da dire ma, a questo punto, è più importante che riusciate a prendere confidenza con la scrittura di pseudocodice. Vedremo esempi di pseudocodice per problemi più specifici più avanti. Per ora cercate di risolvere questi esercizi.

ESERCIZIO 1:
Scrivere lo pseudocodice che risolve il seguente problema (che abbiamo già visto prima):

“Dato che l’ammontare totale di un prestito è di 500 Euro e tenuto presente che se tale valore è maggiore o uguale a 100 Euro gli interessi saranno del 10% e altrimenti saranno del 6%, calcolare l’importo totale che dovrà essere restituito.”

ESERCIZIO 2:
Modificate a vostro piacimento il problema dell’esercizio 1 e scrivete lo pseudocodice che lo risolve.

ESERCIZIO 3:
Scrivete lo pseudocodice necessario per la preparazione di una pizza (per il tipo di pizza fate voi ...)

ESERCIZIO 4:
Provate ad ipotizzare un ciclo di Ripetizione in cui il test logico non sia posizionato a monte della Sequenza di istruzioni da ripetere ma a valle. Per questo tipo di ciclo disegnate il diagramma di flusso ed il corrispondente pseudocodice.

Strutture dati

La ricerca di una rappresentazione opportuna per i dati coinvolti in un problema da risolvere è uno step fondamentale nello sviluppo di un programma. Per molte applicazioni la scelta di una struttura dati appropriata rappresenta l'unica scelta di importanza critica durante l'implementazione del programma: una volta che questa scelta è stata effettuata gli algoritmi da adoperare per risolvere il problema sono semplici.
Per memorizzare il medesimo tipo di dato alcune strutture dati richiedono più spazio (memoria) di altre. Per alcune operazioni da realizzare manipolando il medesimo tipo di dati, alcune strutture dati permettono l'utilizzo di algoritmi più efficienti (veloci in termini di tempo) di altre. E' quindi importante conoscere le strutture dati in modo da scegliere quella più adatta da utilizzare nel programma che vogliamo sviluppare.
Avrete notato che parlare di "dato", "tipo di dato" e "strutture di dati" come entità diverse può generare confusione. Cerchiamo quindi di dare una definizione precisa del significato che attribuiremo ad ognuno di essi:

- **Dato** : In un linguaggio di programmazione, un dato è un *valore* che una variabile può assumere.
- **Struttura dati** : è un'entità usata per organizzare un insieme di dati all'interno della memoria del computer, ed eventualmente per memorizzarli in una memoria di massa. La scelta delle strutture dati da utilizzare è strettamente legata a quella degli algoritmi.
- **Tipo di dato** (o semplicemente **tipo**) : è un *nome* che indica l'insieme di valori che una variabile, o il risultato di un'espressione, possono assumere e le operazioni che su tali valori si possono effettuare. A desempio, se il tipo di dato è "integer" (intero) allora su di esso è possibile effettuare operazioni aritmetiche elementari. NB: il concetto di **tipo** indica l'insieme di valori ammissibili per le componenti di una struttura dati **e** delle operazioni che è possibile svolgere su di essi. I tipi di dato possono essere classificati sulla base della loro complessità.
  - **Tipo di dato astratto (o derivato)** : Ad esempio, un modello matematico, dato da una collezione di valori e un insieme di operazioni ammesse su questi valori. I tipi di dato astratti possono essere utilizzati per rappresentare qualsiasi concetto (ad esempio un'automobile) ma sarà sempre composto da un insieme di valori e operazioni amemsse su questi valori. La progrtazione di un tipo di dato astratto e la sua implementazione sono, solitamente, a carico del programamtore in quanto i tipi di dato astratto da utilizzare in un programma dipendono strettamente dal problema da risolvere e dalla soluzione che si decide di realizzare.
  - **Tipo di dato primitivo (o atomico)** : Forniti dal linguaggio di programmazione, essi costituiscono i "mattoni" con cui custruire i tipi di dati astratti. Esempi comuni: integer(+ , - , * , /). boolean(&& , ||) .

La gestione dei tipi di dato (utilizzo e creazione di tipi) è una caratteristica che varia tra i linguaggi di programmazione. In particolare esistono due tipi fondamentali di filosofie di gestione dei tipi che permettono di definire altrettante classi di linguaggi di programmazione:

- Linguaggi **non tipizzati** : In essi il programmatore non è costretto a specificare il tipo di ogni variabile. Sembrerebbero quindi essere i più semplici da utilizzare. Invece no. Il prototipo di linguaggio non tipizzato è il linguaggio macchina. In esso non è necessario specificare il tipo dei dati poichè tutto avviene mediante manipolazioe diretta di bit (0 e 1) e di operazioni anch'esse specificate sottoforma di sequenze di bit.
- Linguaggi **tipizzati** : Abbiamo detto che una variabile è definita dal suo nome (che deve essere unico se non in casi particolari ed estremamente infrequenti). Nei linguaggi tipizzati, per poter creare una variabile, il solo nome non basta. E' necessario inserire anche l'informazione sul tipo della variabile la prima volta che questa compare nel programma. Ad esempio non sarà possibile creare una variabile di tipo intero scrivendo $a = 1$ ma saremo costretti a scrivere **int** $a = 1$. Non tutti i linguaggi tipizzati obbligano il programmatore a specificare il tipo delle variabili. In alcuni casi è l'interprete che cerca di "indovinare" il tipo delle variabili a seconda dei valori che cerchiamo di assegnare loro durante la fase di traduzione in linguaggio comprensibile alla macchina. L'interprete Perl **non obbliga** a dichiarare il tipo delle variabili e cerca di effettuare le conversioni tra tipi di variabili in modo del tutto automatico. Questa è una caratteristica comoda per il programmatore ma può anche essere fonte di errori.

Tipi di dati primari

In questa sezione vengono elencati alcuni tipi di dati primari che sono comunemente resi disponibili dalla maggior parte dei lnguaggi di programmazione.

Booleani (bool) - valori logici :

Il tipo di dato bool ammette due soli valori : **VERO** e **FALSO**. Dal punto di vista della quantità di informazione essi sono i dati meno "dispendiosi" in quanto richiedono un solo bit il cui valore sarà 0 (FALSO) oppure 1 (VERO). Per quanto semplici essi, come tutti gli altri tipi di dati primitivi, sono forniti dal linguaggio insieme a tutti gli strumenti necessari per manipolarli. Dato "strumento" è un termine fin troppo generico si preferisce indicare i simboli che permettono di effettuare *operazioni* sui dati con il nome di **operatori**. Esistono vari tipi di operatori in grado di manipolare o generare dei valori booleani:

- **Operatori logici**:
  - **AND** && : questo operatore *genera un booleano* a partire dal confronto di N valori booleani. Supponiamo di avere tre variabili a, b, c. Essendo valori booleani possono assumere solo i valori VERO o FALSO. Supponiamo inoltre che $A = VERO$, $b = VERO$ e $c = VERO$. In tal caso se scrivo $risultato = a\&\&b\&\&c$ otterrò una nuova variabile $risultato$ avente valore VERO. Perchè il risultato finale sia VERO **tutti** i valori concatenati dagli operatori && **devono** essere VERO. Se avessimo una collezione di 100 valori booleani e 99 fossero VERO e 1 FALSO il risultato finale sarebbe FALSO.

  - **OR** $||$: questo operatore genera un valore booleano a partire da N valori booleani. Questo operatore genera un valore VERO se **almeno uno dei booleani** che vengono esaminati ha valore VERO. Quindi nel caso di una collezione composta da 100 valori booleani di cui 99 fossero VERO e 1 FALSO il risultato finale sarebbe VERO.

- **Operatori relazionali**:

  - **eq , ==** : Se utilizzato per il confronto di due numeri (a==b) genera TRUE se le due variabili *numeriche* contengono lo stesso numero. Se utilizzato per il confronto di due stringhe (a eq b) genera TRUE se le due stringhe contengono esattamente lo stesso testo.
  - **ne , !=** : Se utilizzato per il confronto di due numeri (a!=b) genera TRUE se le due variabili *numeriche* contengono numeri diversi. Se utilizzato per il confronto di due stringhe (a eq b) genera TRUE se le due stringhe contengono testi diversi.
  - $<$ **,** $>$: sono i classici minore e maggiore utilizzati nel confronto tra numeri.
  - $<=$ **,** $>=$: sono i classici **minore o uguale** e **maggiore o uguale** utilizzati nel confronto tra numeri.

Un fatto importante da ricordare è che i valori booleani sono quelli coinvolti nei test logi che permettono il funzionamento dei controlli di flusso (in particolare IF, WHILE e UNTIL).

Interi e numeri in virgola mobile :

Alcuni linguaggi di programmazione utilizzano tipi differenti per questi tipi di dato. Come abbiamo detto in precedenza in Perl questa distinzione non è così netta. Infatti l'interprete Perl realizzerà in automatico tutte le conversioni tra tipi eventualmente necessarie prima di eseguire il codice. Gli operatori disponibili per questi tipi di dato sono i seguenti:

- **Operatori aritmetici**:

  - **+ - * /** : somma, sottrazione, moltiplicazione e divisione, rispettivamente.
  - ****** : elevamento a potenza (a elevato a b, a ** b).
  - **%** : modulo o resto intero (7 % 3 restituisce 1). NB: ha senso solo per gli interi.
  - **++** : autoincremento. Se una variabile a vale 5 scrivere $a++$ aggiorna il suo contenuto e lo fa diventare 6.
  - **−−** : autodecremento. Se una variabile a vale 5 scrivere $a--$ aggiorna il suo contenuto e lo fa diventare 4.

Stringhe :

Questo tipo di dato permette di manipolare del testo. I valori adatti ad essere salvati in variabili di tipo stringa sono sequenze di testo racchiuse da apici singoli ' o da apici doppi " . Ad esempio a='uno' oppure a="uno" . Per quanto Perl cerchi di convertire i tipi di dati in caso di bisogno esso non è in grado di convertire il nome di un numero nel numero stesso. Gli operatori chepermettono di manipolare le stringhe sono i seguenti:

- Alcuni operatori per rmanipolare stringhe

  - **eq** : operatore logico che restituisce un valore booleano dal confronto tra due stringhe. Sestituisce VERO se le stringhe sono uguali.
  - **ne** : operatore logico che restituisce un valore booleano dal confronto tra due stringhe. Sestituisce VERO se le stringhe NON sono uguali.
  - **.** : operatore di concatenazione. Se a contiene ”Pe” e b contiene ”rl” allora c=a.b genera una variabile stringa che contiene ”Perl”.

ATTENZIONE:
Quelli che vi ho mostrato in questa sezione sono gli operatori standard per operare sui tipi di dati primari forniti da molti linguaggi ma essi possono avere simboli diversi in linguaggi diversi (ad esempio in alcuni linguaggi di programmazione AND è rappresentato da & e non da &&). Dato che questo è un corso di programmazione in Perl **ho presentato direttamente gli operatori che si utilizzano per programamre in Perl**.
Altra precisazione: Perl è probabilmente il linguaggio piùpotente per la manipolazione di stringhe e testo. Quelli che vi ho mostrato sono solo gli operatori di manipolazione stringhe fondamentali ma ne esistono molti altri che vedremo più avanti in una sezione dedicata.

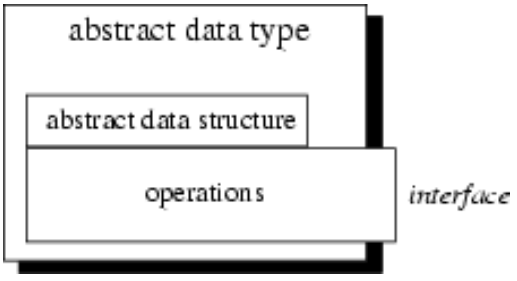
Tipi di dati derivati

Come abbiamo detto in precedenza i tipi di dati derivati possono essere molto complessi e sono in gran parte sviluppati dal programmatore durante la scrittura del programma ma esistono, indubbiamente, tipi di dati che sono più complessi dei tipi primari ma sono utilizzati talmente di frequente che la maggior parte dei linguaggi li fornisce direttamente ai programmatori in modo da non dover essere costretti ogni volta a doverli reimplementare.

ATTENZIONE:
Come nel caso della sezione precedente anche in questa ogni volta che farò degli esempi specifici utilizzerò dei simboli simili a quelli che utilizzeremo quando programmeremo in Perl.

Sequenza :

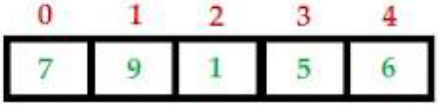
Questo tipo derivato è talmente comune da essere messo a disposizione da tutti i linguaggi di alto livello. A volte ci si riferisce ad esso con nomi diversi tra cui **array** e **vettore**. La sua struttura è davvero semplice, infatti la sequenza, come dice anche il suo nome, non è nient’altro che una collezione di valori dello stesso tipo primario **memorizzati uno dopo l’altro** a cui è possibile accedere singolarmente per mezzo della loro posizione all’interno della sequenza.



Tenuto conto del fatto che un tipo di dato derivato (o astratto) è composto da una collezione di dati ed un insieme di operatori che permettono di manipolare questi dati possiamo fare immediatamente alcune osservazioni importanti:

- I tipi di dato primario contenuti nel tipo di dato astratto **non perdono** tutti gli operatori validi per la manipolazione dei dati primari. Ad esempio se il tipo derivato contiene due numeri interi sarà ancora possibile sommarli ed ottenere ancora un numero intero.
- La manipolazione dei dati primari contenuti nel tipo di dato derivato può avvenire solo **DOPO** che abbiamo selezionato, tra tutti quelli disponibili, l'elemento della struttura dati che ci interessa.
- Poichè si può operare sugli elementi interni solo dopo averli selezionati **ogni tipo di dato derivato DEVE fornire un modo per accedere ai suoi dati interni**. Altrimenti sarebbe inutile. L'accesso agli elementi interni è fornito mediante appositi operatori.

Vediamo come ciò che abbiamo detto si applica al caso particolare della sequenza. Una sequenza (il Perl le chiama Array) è una collezione di elementi ordinati tutti dello stesso tipo primario. Ad esempio:



Possiamo subito notare che ad **ogni elemento** dell'array di numeri interi rappresentato in figura è associato un indice che va da 0 a 4. Un altro fatto importante da notare è che il primo indice è 0 e non 1. Questa è una caratteristica che varia da linguaggio a linguaggio. La maggior parte dei linguaggi di programmazione fa iniziare gli indici degli array da 0 il che ha senso poichè anche la macchina inizia a contare gli interi da 0. Tuttavia esistono alcune eccezioni. Perl utilizza come primo indice degli array il valore 0 mentre (ad esempio) R utilizza come primo indice degli array (che chiama vettori...) il valore 1.

Supponiamo che il nome della variabile di tipo derivato rappresentata in figura sia **array1**. Se vogliamo accedere al terzo elemento (quello con indice 2) e assegnare il suo valore a una variabile di nome a dovremo scrivere (nello pseudocodice):

$a \leftarrow$ array1[2]

Se, invece, vogliamo cambiare il valore dell'elemento in terza posizione in array1 dovremo scrivere:

array1[2] $\leftarrow$ 10

in questo modo il valore 1 viene sostituito con il volre 10. Naturalmente possiamo assegnare ad un elemento della sequenza anche il valore contenuto in un'altra variabile (ad esempio c) in questo modo:

array1[2] $\leftarrow c$

Quindi le parentesi quadre funzionano da *operatori di accesso* agli elementi della sequenza. L'implementazione del tipo di dato derivato sequenza varia da linguaggio a linguaggio ma ha alcuni elementi in comune:

- Il linguaggio (a parte casi rarissimi che non riguardano il Perl) non esegue alcun controllo sull'esistenza o meno degli elementi che compongono la sequenza. Quindi se cercate di assegnare un valore al millesimo elemento di una sequenza di cinque elementi questo genera un errore e il programma si blocca.
- E' prevista la possibilità di aggiungere o rimuovere elementi ad unas equenza **dopo** la sua creazione.
- E' prevista la possibilità di creare una nuova variabile di tipo sequenza prendendo una parte di una sequenza esistente.

Proviamo ad utilizzeare delle variabili di tipo sequenza per scrivere la soluzione di un problema dei seguenti esercizi.

ESERCIZIO 5:
Data una sequenza di 100 numeri interi ed un numero a scrivere un programma in pseudocodice che conti il numero degli elementi $> a$ e degli elementi $<= a$. Qual'è il numero minimo di variabili richieste per risolvere questoproblema? Perchè?

ESERCIZIO 6:
Data una sequenza di 100 numeri interi scrivere un programma in pseudocodice che conti restituisca l'elemento con valore massimo.

ESERCIZIO 7:
Data una sequenza di 300 caratteri appartenenti all'alfabeto a,c,t,g scrivere un programma in pseudocodice che trovi tutti gli eventuali codoni START e stampi la loro posizione. Nel caso in cui nessun codone START venga trovato il programma, prima di terminare, deve scrivere "Codone START (o ATG ... fate voi) non trovato".

Hash :

Anche l'hash, come la sequenza, è una collezione di valori ma, rispetto alla sequenza, ha una modalità di accesso ai singoli elementi molto diversa. Infatti:

- I valori sono memorizzati in ordine del tutto **casuale**.
- Ad ogni valore è associata una chiave **testuale**.
- Per accedere (leggere-scrivere) a un valore dobbiamo fornire la sua chiave.

Perchè questo tipo di struttura possa funzionare correttamente è assolutamente necessario che le chiavi siano **uniche** e, infatti, chiavi duplicate non sono ammesse.

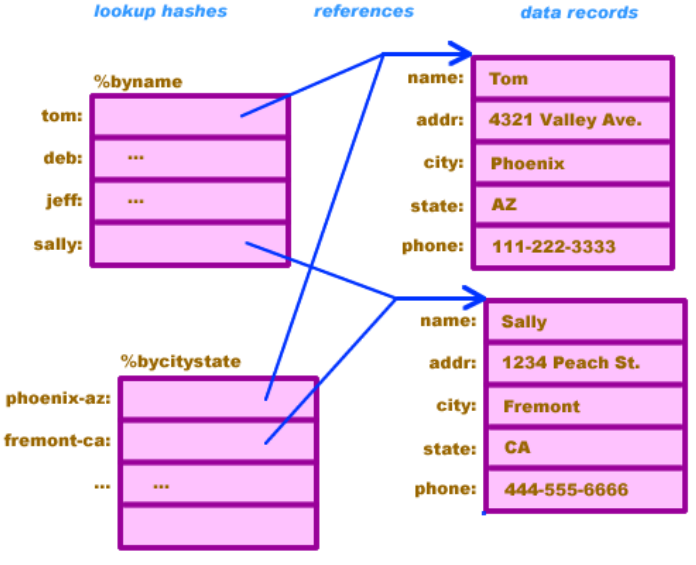
Non fatevi trarre in inganno dal fatto che nella figura le celle sembrano essere ordinate. Questa è una semplificazione (che tornerà utile per salvare spazio nella prossima figura). Se la variabile h riportata in figura è di tipo hash allora per scrivere nel suo secondo elemento dovremo scrivere (nello pseudocodice):

h{'deb'} $\leftarrow$ 5

E per assegnare il valore del suo terzo elemento ad una variabile b dovremo scrivere:

$b \leftarrow$ h{'jeff'}

L'utilità degli hash sta nel fatto che sono molto versatili quando dobbiamo costruire variabili composte. E' ad esempio possibile hash con più di un livello di chiavi. Ad esempio 2, 3 ... o quanti ne servono per rappresentare i dati coinvolti nel problema che vogliamo risolvere.



Se, ad esempio abbiamo una variabile hash con 2 livelli di chiavi e avente nome $byname$ e vogliamo scrivere in una variabile c il nome della città in cui abita Sally

potremo scrivere (nello pseudocodice):

$c \leftarrow$ byname{'Sally'}{'city'}

Allo stesso modo, se vogliamo impostare il valore della città in cui abita Tom potremo scrivere:

byname{'Tom'}{'city'} $\leftarrow$ 'Boston'

E' possibile rappresentare dati estremamente complessi utilizzando degli hash con chiave multilivello. Essi risultano particolarmente utili quando quando dobbiamo immmagazzinare in memoria molte informazioni inerenti ad oggetti aventi un identificativo unico (ad esempio proteine con accession RefSeq o UniProt, o geni, o trascritti ...). Provate a risolvere i seguenti esercizi ipotizzando l'utilizzo di variabili hash.

ESERCIZIO 8:
Avete ricevuto un file contenente informazioni riguardanti 2000 proteine. In questo file ogni riga corrisponde ad una proteina. Il file è 'strutturato' nel senso che la struttura delle informazioni presenti in ogni riga è la seguente:

accessionRefSeq|pesomolecolare|numeroresidui|numeroresiduicarichi

Il primo 'campo' è di tipo testuale, tutti gli altri sono di tipo numerico. Scrivere in pseudocodice un programma che legga il file riga per riga, spezzi ogni riga in corrispondenza del simbolo | , salvi temporaneamente i valori in una variabile di tipo *sequenza* e salvi i dati in un *hash* **ad un livello** in grado di associare ad ogni accession il numero di residui della proteina corrispondente.

ESERCIZIO 9:
Risolvete di nuovo l'esercizio 8 ma, questa volta, slvate le informazioni in un *hash* a **due livelli** che vi permetta di memorizzare non solo il numero di residui ma anche il peso molecolare e il numero di residui carichi.