Guida alla Programmazione Generale

Davide Aversa

2009

Indice

Parte I

Principi Base

Capitolo 1

La Programmazione

1.1 Introduzione

Questo "libro" nasce un po dal caso.

In principio *Giuda alla Programmazione Generale* non doveva essere altro che una seria di 10 articoli da pubblicare sul mio blog tecnologico *Slashcode* come rimedio alla noia estiva. Poi sono arrivati i commenti positivi ed ho capito che poteva nascere qualcosa di utile. Cosi ho cominciato a rivedere e raccogliere tutto ciò che avevo già pubblicato.

L'approccio utilizzato è quello più semplice e alternativo possibile. Partiremo dai concetti astratti di programmazione per arrivare a quelli più legati all'implementazione.

Nel mio pellegrinare nella rete mi sono trovato spesso davanti a persone desiderose di imparare a programmare ma che ignoravano i fondamenti di questa "dottrina". In questi casi la domanda che segue naturale è sempre "quale linguaggio per cominciare?"

Vorrei ricordare a queste persone che programmare non significa conoscere un linguaggio di programmazione (o per lo meno, non solo) bensì conoscere la base concettuale che sta alla base del concetto stesso di "programmazione" senza trascurare il computer nel suo funzionamento a livello "macchina", come "pensa", insomma... come funziona.

Per questo la prima parte di questo volumetto girerà attorno alla programmazione in senso lato, astratta da ogni implementazione, in modo tale da fornire quello strato di informazioni e concetti che sarà la base per lo studio di ogni linguaggio incontrerete nel vostro cammino.

1.2 Che cos'è?

Partiamo dal basso. Da molto in basso. Talmente in basso che andremo oltre il concetto di programmazione per calcolatori concentrandoci su finalità, mezzi e basi della programmazione "pura".

Per programmazione si intende comunemente *un insieme di operazioni ordinate che vanno effettuate per raggiungere un obbiettivo.*

Come vedete questa definizione si adatta a molti ambiti e non solo quello informatico. Possiamo infatti programmare una vacanza. Anche in questo caso infatti dobbiamo scegliere una serie di tappe da fare giorno dopo giorno (*operazioni ordinate*) in cui dovremo andare (*esecuzione*) per poter passare una vacanza che ci soddisfi (*obbiettivo*).

Questo, mi raccomando, è il concetto fondamentale che sta dietro a tutto quello che faremo d'ora in avanti.

L'insieme di queste tre parole chiave (operazioni ordinate, esecuzione, obbiettivo) prende il nome di *algoritmo.* Un algoritmo è costituito infatti da una *lista finita* di operazioni da eseguire finalizzate al raggiungimento di un obbiettivo in un *tempo finito.*

Notate che ho evidenziato la parola "finito". Questo perch, anche se sembra, non è scontato. Esistono infatti parecchi "pseudo-algoritmi" o "liste di operazioni" che necessitano di essere ripetute infinite volte per raggiungere il loro scopo (per esempio trovare il valore del pi greco, o gli zeri di un polinomio per approssimazioni successive). Questi non sono algoritmi data la loro inapplicabilità (ma possono diventarlo se ci accontentiamo di una certa approssimazione).

Da queste considerazioni appare evidente il legame fra la programmazione pura e gli algoritmi. La programmazione diventa quindi "l'arte" di miscelare uno o più algoritmi per raggiungere uno o più scopi.

1.3 Algoritmi

L'astrazione da ogni linguaggio ci permette di dimenticarci per il momento le difficoltà implementative di ogni algoritmo e concentrarci solo sulla sua essenza.

Partiamo quindi dalla struttura di algoritmi veramente semplici.

Il più semplice tipo di algoritmo consiste in una semplice lista di istruzioni. Se per esempio vogliamo scrivere l'algoritmo per fare la pasta potremo scrivere:

1. Metti l'acqua a bollire.
2. Aspetta che l'acqua bolla.
3. Mettere la pasta.
4. Aspetta che sia cotta.
5. Scola la pasta.

Questo è un algoritmo *aciclico* (ovvero privo di salti all'indietro, in poche parole di istruzioni ripetute più volte) e a *flusso unico* (ovvero tutte le istruzioni sono eseguite almeno una volta). E' il modello di algoritmo più semplice.

Approfondiamo ora il concetto di flusso. Per **flusso** si intende uno dei possibili "percorsi" in un algoritmo che conducono dall'inizio alla fine.

Chiariamo con un esempio.

Supponiamo il seguente algoritmo (dove *condizione* è una qualunque condizione):

1. A
2. Se condizione B
3. altrimenti C
4. D

L'algoritmo presenta due possibili flussi:

1. A B D – Se la condizione è verificata.
2. A C D – Se la condizione **non** è verificata.

Per la gestione dei flussi abbiamo a disposizione solo due istruzioni: **salti condizionati** e **salti incondizionati**.

I "salti condizionati" sono esattamente quelli nell'esempio, indicano dove saltare **solo se** la condizione è verificata.

I "salti incondizionati", invece, sono istruzioni che "obbligano" in ogni caso a saltare all'istruzione specificata. In realtà i salti incondizionati possono essere visti come salti condizionati la cui condizione è sempre vera, riducendo in pratica il numero di istruzioni di controllo ad una sola. Per questioni di semplicità però continueremo a considerarle due istruzioni separate.

Approfondiremo la gestione dei flussi più avanti nel testo.

Torniamo al nostro esempio. Al lettore attento non sarà sfuggito però che ogni istruzione potrebbe considerarsi a sua volta come un algoritmo. Per esempio la (1) potrebbe diventare:

1. Prendi la pentola.
2. Riempila d'acqua.
3. Accendi il fornello.
4. Metti la pentola sul fornello.

E via dicendo. A sua volta anche "prendi la pentola" potrebbe essere scomposto.

Ma fino a che punto dobbiamo espandere le operazioni di un algoritmo?

La risposta in verità dipende dal linguaggio che si userà per realizzare l'algoritmo. Se vogliamo eseguirlo in assembly dovremmo espanderlo ai massimi livelli mentre se vogliamo farlo in Java avremo bisogno di un minore dettaglio.

É buona norma però, quando si scrivono algoritmi, partire sempre da un livello di dettaglio basso e scomporre via via i sotto-problemi risultanti in altri sotto-algoritmi. Un po come abbiamo fatto per l'esempio della pasta. Questa segnatela perch all'atto pratico è una delle cose più potenti che un programmatore possa fare: la *modularizzazione* (di cui parleremo in seguito).

L'istruzione (2) "Aspetta che l'acqua bolla." scomposta rivela proprietà che non avevamo visto nell'algoritmo generale. Scomposto infatti assume questa forma:

1. Guarda l'acqua nella pentola.
2. Se bolle termina, altrimenti torna a (1).

Come si nota questo è un algoritmo *ciclico* (dei più semplici pergiunta) in quanto presenta un salto condizionato ci fa tornare indietro ad un istruzione precedente. Questo genere di salti viene chiamato **ciclo**.

Quindi il nostro algoritmo aciclico di partenza in realtà nasconde un sotto-algorimto ciclico.

Ora ci chiediamo qual'è il metodo più efficace per rappresentare un algoritmo? Il metodo forse più conosciuto è quello dei **diagrammi di flusso** (Vedi ad esempio Figura 1.1).

Il diagramma di flusso è quindi una rappresentazione immediata per via grafica di quello che fa l'algoritmo. Come si vede dall'immagine, un diagramma di flusso mostra esplicitamente quali sono e come si comportano i vari flussi di un algoritmo. Lo svantaggio che lo rende inapplicabile in quasi tutti i casi pratici è che è inadatto per algoritmi vasti e complessi in quanto stare dietro a frecce e blocchi sparse per una superficie troppo elevata disorienta più che aiutare.

A questo punto ci viene in aiuto lo **pseudo-codice**.

Lo pseudo-codice in realtà non è molto dissimile dalla lista di istruzioni che ho usato per rappresentare l'algoritmo della pasta. In poche parole è un algoritmo espresso in linguaggio naturale ma impostato in modo tale da somigliare a un listato di codice.

Non c'è una tecnica universale per lo pseudo-codice, ognuno è libero di trovare il suo stile a patto che il risultato sia comprensibile e chiaro ai più. Per esempio (2) "Aspetta che l'acqua bolla." diventerebbe una cosa del tipo:

```
algoritmo AspettaAcquaBolle() :
```

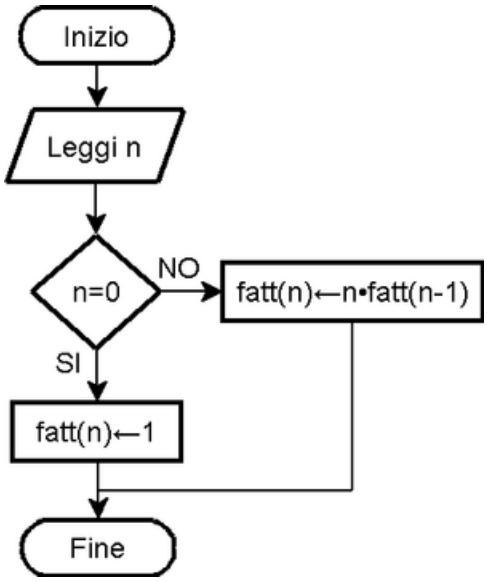


Figura 1.1: Il diagramma di flusso dell'algoritmo di calcolo del fattoriale. Il calcolo del fattoriale è un algoritmo ciclico a flusso multiplo e ricorsivo.

```
    Finche' <l'acqua NON bolle> :
        <Guarda L'acqua>
```

Mentre l'algoritmo del fattoriale diventerebbe:

```
algoritmo Fattoriale(Numero Intero N) :
    Se N=0 :
        restituisci 1
    altrimenti :
        restituisci N * Fattoriale(N-1)
```

Come vedete il risultato è facilmente interpretabile e la struttura permette una conversione piuttosto semplice in un linguaggio di programmazione vero e proprio. Si tratta cioè di tradurre riga per riga lo pseudo-codice in codice.

1.4 Dividi et Impera

Unavolta che siamo a conoscenza del problema dobbiamo tenere presente che esistono diversi algoritmi che possono risolverlo. Se questi, a livello astratto, sono equivalenti, non lo sono all'atto pratico. Il calcolatore infatti

è un entità fisica e come tale impiega quantità di tempo finite per eseguire ogni passo di un algoritmo.

Per questo algoritmi che in linea teorica funzionano, implementati a livello macchina, diventano inapplicabili in quanto finiscono per richiedere quantità esagerate di tempo.

Parleremo del problema dell'efficenza in modo più approfondito in seguito, ora però affronteremo come progettare un algoritmo con il metodo del **Dividi et Impera**.

La progettazione di un algoritmo è la fase più delicata del processo di programmazione. Spesso scelte che sembrano ininfluenti nella nostra mente si tramutano in barriere computazionali invalicabili.

Durante il progetto possiamo distinguere 3 fasi:

1.4.1 Scomposizione in sotto-problemi

In questa fase abbiamo nella nostra testa solamente l'obbiettivo che ci prefiggiamo (risultato) e i punti di partenza (dati in ingresso). Per esempio abbiamo l'obbiettivo di fare la spesa nel minor tempo possibile avendo come dati in ingresso la posizione dei supermercati della città.

La prima cosa da fare a questo punto è trovare a grandi linee i passi che ci portano dalla partenza all'obbiettivo. Nel nostro esempio sappiamo che la soluzione consiste nel:

1. Trovare la strada più breve per arrivare ad un supermercato.
2. Andare a fare la spesa.
3. Trovare la strada più breve per tornare a casa.

1.4.2 Risoluzione dei sotto-problemi

Trovati i sotto problemi del nostro problema principale dobbiamo risolvere i sotto-problemi (i quali di solito sono più semplici del problema di partenza) tenendo conto che abbiamo a disposizione solo tre elementi:

- Istruzioni (le istruzioni da eseguire).
- Controlli su condizioni (corrisponde all'italiano se...).
- Salti (per saltare ad un istruzione specifica).

Qui non c'è un metodo universale. Bisogna lavorare di intuito suddividendo, se necesarrio, il sotto-problema in sotto-sotto-problemi, se necessario fino al punto in cui i sotto-sotto-sotto-...-sotto-problemi non diventino le istruzioni base di cui disponiamo.

1.4.3 Unione dei risultati

Questa fase non è banale come sembra. Per ricostruire il problema principale sembra ovvio che vadano messi in fila i risultati dei sotto-problemi. Invece no. Qui è il punto (non l'unico ma uno dei principali) in cui entra in gioco l'efficienza.

Per tornare all'esempio della spesa ci rendiamo conto. Che i sotto-problemi (1) e (2) in realtà fanno la stessa cosa: la strada più breve all'andata, con buona possibilità, sarà anche la più breve al ritorno.

Quindi perché calcolarla due volte? Basta salvare da qualche parte il risultato del problema (1) e riutilizzarlo in (3) senza pagare due volte il costo (in termini di calcolo) dell'elaborazione del percorso.

In questo esempio sembra ovvio ma vi assicuro che non è cosi. Spesso quando si programma si tende a andare a risparmio sulle variabili (forse per non sforzarsi a trovare un nome) e quindi a volte per usare un valore si applica più volte la funzione che lo genera. Ma spesso questo appesantisce enormemente il lavoro e l'efficienza.

Sul costo e la complessità di un algoritmo torneremo in seguito perché è un argomento interessante e utile.

1.4.4 Conclusione

Abbiamo quindi definito queste fasi:

SCOMPOSIZIONE - RISOLUZIONE - UNIONE.

Questo principio prende il nome di *Dividi et Impera* ed oltre ad avere il vantaggio di aiutare nella "creazione" rende il software modulare il che comporta:

1. Maggiore capacità di manutenzione e individuazione degli errori.
2. Possibilità di riutilizzare la stessa soluzione di un sotto-problema di un algoritmo in un altro algoritmo (Per esempio il problema di trovare la strada più breve per fare la spesa lo si può riutilizzare nel trovare la strada più breve in altre situazioni).

Capitolo 2

I Linguaggi di Programmazione

2.1 Linguaggi Compilati e Linguaggi Interpretati

Questa è la prima parte dedicata ai linguaggi di programmazione a cui molto probabilmente eravate interessati se vi siete spinti fin qui.

Ora che avete nozioni sufficenti su cos'è la programmazione e cos'è un algoritmo non dovrebbe essere difficile spiegare in cosa consiste un linguaggio di programmazione.

Un **linguaggio di programmazione** non è altro che un linguaggio artificiale che associa specifici costrutti sintattici una o più operazioni eseguibili dal processore.

Senza addentrarmi però in formalismi diamo un occhiata alle caratteristiche "esterne" dei vari linguaggi. In particolare in questa sezione mi concentrerò sulla prima grande divisione dei linguaggi di programmazione: i **linguaggi compilati** e i **linguaggi interpretati**.

Dobbiamo per prima cosa spiegare come fa il vostro pc a sapere che 2+2 fa 4 e soprattutto come fa a capire perchè deve dirvelo.

Gli ingredienti sono due: un **file binario** (composto cioè da una serie di 0 e di 1) caricato da qualche parte (in genere la memoria) e un processore in grado di leggerlo.

Il processore è essenzialmente stupido: lui prende dal file i primi 32 (o 64) zeri-uni (bit) e li legge, capisce l'istruzione che deve fare, la esegue, e passa ai 32 bit successivi, oppure, se l'istruzione era un salto, ai 32bit piazzati in un punto specificato (per esempio) nei 32bit appena letti.

Le istruzioni codificate nei 32bit che il processore di volta in volta va a prendere sono le istruzioni **assembly**, sono strettamente collegate allo specifico processore e per questo variano da macchina a macchina.

Ma chi scrive il file binario? Ovviamente il programmatore. Ma come lo scrive? Ovviamente non mettendo in fila 0 e 1.

Potrebbe scriverlo in assembly. Ma il limite non è (come si pensa) la difficoltà bensì il fatto che esso varia da processore a processore e quindi un programma scritto interamente in assembly diventerebbe del tutto inutilizzabile su un processore diverso e andrebbe riscritto da capo.

E come si sa i programmatori sono **pigri**.

2.1.1 Linguaggi Compilati

Per ovviare al problema nascono i *linguaggi di programmazione compilati*. I linguaggi di programmazione mettono a disposizione set di istruzioni (quasi sempre molto più leggibili e umane delle istruzioni assembly) che raggruppano una o più istruzioni macchina. Esempi di linguaggi compilati sono il *C/C++, Fortran, Delphi e altri...*

I linguaggi di programmazione prima di diventare eseguibili vanno per l'appunto **compilati**.

Un compilatore non è altro che un programma che traduce il testo scritto nel dato linguaggio in un listato assembly proprio della macchina su cui è in esecuzione. In questo modo lo stesso codice in un linguaggio di "alto" livello può venire compilato su macchine diverse e continuare a funzionare benissimo.

Il compilatore *gcc* per esempio permette (tramite il parametro -S) di generare un file .s contenente il listato assembly invece di generare l'eseguibile vero e proprio. Potete così dare un occhiata al linguaggio "grezzo" che parla il vostro PC. State attenti che anche un programma semplice come *Hello World!* genera un listato assebly di centinaia di righe.

In pratica **l'unico programma che va riscritto quando si cambia tipo di processore è il compilatore**.

Il listato assembly però non è ancora pronto per essere eseguito. Infatti esso va assemblato con un programma chiamato **assemblatore**. L'assemblatore non fa altro che codificare ogni istruzione assembly nella rispettiva serie di 0 e 1.

A questo macchine più moderne aggiungono il cosiddetto linker. Il Linker si incarica di attaccare in cima al file binario tutte quelle istruzioni che ne permettono una corretta esecuzione all'interno di un sistema operativo.

Questi tre elementi (*compilatore-assemblatore-linker*) sono le tre parti fondamentali per rendere eseguibile del codice scritto nel linguaggio di programmazione *XYZ*.

2.1.2 Linguaggi Interpretati

Col passare del tempo però è sorta la necessità di creare programmi che non sopportassero solo il trasporto da un processore ad un altro ma anche tra un sistema operativo ed un altro. Questa spinte evolutiva è opera soprattutto di *internet*: nel giro di pochi anni, milioni di computer con i SO più disparati si

affacciarono in un mare comune. Era quindi necessario trovare un linguaggio comune comprensibile da tutti.

Nascono per questo i **linguaggi di programmazione interpretati**. Tali linguaggi non passano attraverso i tre elementi descritti in precedenza ma girano su una macchina virtuale in grado di leggere direttamenteil codice e di tradurlo al processore che può cosi eseguire. Esempi di linguaggi interpretati sono *Java, Python, PHP, C Sharp* e molti altri.

Il processo di traduzione effettuato dalla macchina virtuale quando esegue un programma, essendo un passaggio in più, rende ovviamente meno efficiente l'esecuzione del programma.

In realtà quasi tutti i linguaggi interpretati moderni sono *semi-compilati* ovvero vengono compilati in uno pseudo-assembly (il *bitcode* di Java e Lisp o il *pyCode* di Python) che viene eseguito sempre dalla macchina virtuale. Questo è uno dei molteplici espedienti utilizzati dai linguaggi interpretati per aumentare la velocità di esecuzione.

È usanza ormai mescolare il più possibile queste due forme di linguaggi. Nelle applicazioni complesse i linguaggi interpretati come il Python svolgono la funzione di "collante" per tenere insieme parti scritte in linguaggi performanti come C. Ad esempio si può scrivere in C la parte di programma che ha bisogno di prestazioni elevate e ricoprirla con un interfaccia grafica scritta in Python.

2.2 Paradigma a oggetti,procedurale e funzionale

Ora affronteremo un altra categorizzazione fondamentale dei linguaggi di programmazione ovvero quella fra **linguaggi procedurali** e **linguaggi orientati agli oggetti** (e in misura minore quelli **funzionali**).

2.2.1 Linguaggi Procedurali

Innanzitutto è bene sapere che in principio fu il verbo. Insomma, come ci dice anche la bibbia, in principio non esistevano oggetti. La programmazione non solo era puramente procedurale ma era anche lineare ovvero non veniva modificata da input esterni in run-time. Basta pensare alle schede perforate del primo Fortran per rendersi conto che la sequenza di istruzioni era veramente difficile da modificare una volta iniziata l'esecuzione.

Quindi tutti i linguaggi storici sono procedurali. Un linguaggio procedurale infatti segue l'idea classica di programmazione di cui abbiamo parlato in precedenza: in pratica esegue le istruzione in sequenza partendo dalla prima che incontra e si muove all'interno del programma soltanto attraverso **chiamate di funzione** (ovvero chiama altri algoritmi scritti da qualche parte che hanno un nome specifico) e tutti i dati a cui può accedere non sono visti in altro modo che come numeri e indirizzi di memoria.

E' il modo più semplice di concepire la programmazione.

Linguaggi procedurali noti sono il *C*, *Fortran*, *BASIC* e *altri*.

2.2.2 Linguaggi Object Oriented

Successivamente per molte applicazioni iniziò a diventare necessaria non soltanto una rappresentazione degli algoritmi ma anche una rappresentazione della realtà comprensiva di relazioni fra entità reali, proprietà comuni, sottoinsiemi di entità, attributi...

Pioniere di questo fu **Smalltank** un linguaggio pensato appunto per la creazione di simulazioni della realtà.

Questo approccio viene chiamato Paradigma ad Oggetti (o *Object Oriented*).

I linguaggi di questo tipo offrono al programmatore la possibilità di creare oggetti che contengono:

- **Attributi**: ovvero dati, caratteristiche, informazioni riguardanti l'oggetto. Rappresentano in pratica **quello che un oggetto sa di se**.
- **Metodi**: ovvero "abilità" dell'oggetto. Rappresentano in pratica **quello che un oggetto sa fare**.

Per esempio possiamo pensare ad un blog come ad un oggetto: i suoi attributi saranno il nome e la lista dei post che contiene mentre i suoi metodi saranno le sue abilità come ad esempio cancellare o aggiungere un post.

Caratteristica comune dei linguaggi object oriented consiste nell'**ereditarietà**. E' un concetto complicatissimo che cercherò di semplificare cosi (e approfondiremo ovviamente in seguito): abbiamo definito il nostro oggetto blog. Ma a pensarci bene il nostro blog non è solo un blog *MA ANCHE* un sito web. Considerando questo possiamo dire che l'oggetto blog *eredita* dall'oggetto sito web.

Ecco. Ogni qualvolta vi trovate difronte ad un oggetto che è *ANCHE* qualche altra cosa probabilmente quel qualche altra cosa è la sua sovraclasse.

Alcuni esempi: donna e uomo ereditano dall'oggetto essere umano; libro di cucina, libro di narrativa e saggio sono tutti oggetti che ereditano dall'oggetto libro; pomodoro eredita da vegetale che a sua volta eredita da essere vivente.

Come vedete è molto facile sfruttando il paradigma ad oggetti creare una copia "virtuale" di un oggetto materiale.

Ovviamente sugli oggetti andrebbero dette altre centomila cose e appena arriverà il momento analizzeremo nel dettaglio i vari aspetti sia tecnici che logici.

Linguaggi Object Oriented noti sono *C++*,*Java*,*PHP*,*Python* e moltissimi altri.

Bisogna comunque tener presente il fatto che un linguaggio procedurale non è detto implichi che sia meno potente rispetto ad un linguaggio orientato

agli oggetti in quanto tutto ciò che si può fare con la programmazione a oggetti può essere fatto con la procedurale. Basti pensare che nelle prime versioni di C++ il codice veniva tradotto prima da un compilatore in codice C e poi successivamente compilato nel vero senso della parola.

La differenza sta nella notevole semplificazione sia realizzativa che di concetto che un linguaggio Object Oriented offre per questo tipo di applicazioni.

2.2.3 Linguaggi Funzionali

I linguaggi funzionali invece (come il *PROLOG* e il potentissimo *Lisp*) hanno un impostazione del tutto diversa. Fra i linguaggi funzionali esistono numerosi approcci.

Il PROLOG ad esempio è un linguaggio inperneato sulle *proposizioni logiche*. Per esempio posso definire la regola *APPARTIENE(x,y)* come tutte le cose y che appartengono a x. L'interprete PROLOG, quando gli viene posta una domanda si incaricherà di dedurre la risposta tramite questo set di regole.

Il LISP invece è fondamentalmente basato sulle *funzioni ricorsive*. Il programmatore scrive delle funzioni che a sua volta chiamano delle funzioni. La differenza fondamentale fra il LISP ed un qualunque linguaggio procedurale è l'assenza di funzioni iterative come *for* e *while* e l'impossibilità di memorizzare variabili. Questo comporta che tutti gli algoritmi vanno ripensati come algoritmi ricorsivi anche se versioni moderne del LISP integrano anche un approccio procedurale affiancato al classico funzionale.

Questo genere di linguaggi non hanno attualmente molte applicazioni nell'informatica tradizionale ma vengono utilizzati spesso nelle intelligenze artificiali grazie alla loro proprietà di "reinventare se stessi": i linguaggi funzionali possono autogenerare con facilità del codice che andranno poi ad eseguire adattando così il proprio funzionamento in base a input esterni.

2.3 Linguaggi di Scripting e Ordinari

Per finire la nostra panoramica sui linguaggi di programmazione, e prima di addentrarci nei particolari più tecnici, affronteremo i cosiddetti **linguaggi di scripting**, non una novità nel panorama informatico ma che negli ultimi 15 anni stanno subendo uno sviluppo e una diffusione di tutto rispetto.

Per affrontare questo argomento cominceremo dalle caratteristiche che differenziano questi linguaggi dai linguaggi ordinari con cui fino a pochi anni fa erano fatti tutti i programmi esistenti.

Innanzitutto i linguaggi di scripting sono sempre linguaggi interpretati o al più semi-compilati.

Il programma che si occupa dell'esecuzione di questi programmi prende il nome di **shell**.

Storicamente le shell (e quindi i relativi linguaggi di scripting) erano utilizzate esclusivamente per interagire con il sistema operativo. La prima shell di un certo rilievo fu la *Thompson Shell*, utilizzata come shell di default dei sistemi Unix fino alla versione 7 quando fu sostituita dalla *Bourne Shell*.

Attualmente le shell di sistema sono numerose (tanto per citarne alcune, *Bourne Again Shell* conosciuta meglio come *B.A.SH.* la quale è la shell di default dei moderni sistemi Unix, la *Korn Shell*, la *C Shell* che ha una sintassi simile al C e infine *COMMAND.COM*, e il suo successore cmd.exe, la shell di default dei sistemi DOS e Windows) e alcune di loro mettono a disposizione linguaggi di scripting piuttosto potenti.

Quando utilizzate il terminale di linux o il prompt dei comandi di windows in realtà state programmando. Siete già programmatori. State già effettuando algoritmi.

Per esempio la serie di comandi che date per andare nella cartella "SPORCIZIE E PORCATE" per cancellare un file di nome "xxx.avi" è formalmente un programma. L'unica differenza fra quello ed uno script è che state istruendo interattivamente la shell invece di dargli un file (il programma o script appunto) in cui la shell va a leggersi i comandi da eseguire.

Non cambia assolutamente nulla. A parte che fare uno script per cancellare un file sarebbe uno spreco di tempo e inoltre... non lo si potrebbe utilizzare due volte (quindi a che pro fare uno script usa e getta?).

Proprio per il loro compito strettamente legato al sistema gli script si differenziano dai programmi tradizionali per:

- Complessità relativamente bassa.
- Esecuzione di mansioni accessorie specifiche.
- Una spiccata linearità (inizio-esecuzione-fine).
- Mancanza di interfaccia utente.
- Richiamo di programmi esterni per svolgere funzioni complesse.

Successivamente, all'incirca dal 1993 in poi, si sono sviluppati anche linguaggi di scripting non necessariamente legati al sistema operativo. Molti addirittura mettono a disposizioni più o meno completamente modalità di esecuzione Object Oriented, una funzionalità che va decisamente oltre l'uso storico dei linguaggi di script. Fra questi annoveriamo per esempio *Python*, *Ruby*, *Tcl* e *Perl*.

Lo sviluppo di internet inoltre, favori la diffusione dei linguaggi di scripting per il web quali ad esempio PHP, JavaScript, tutti i linguaggi legati ad ASP e altri...

Ma cosa distingue allora un linguaggio di scripting suddetto da un linguaggio tradizionale?

Caratteristia comune nei linguaggi di scripting consiste nel fatto che il programmatore può disinteressarsi delle risorse di sistema. Per esempio può trascurare la gestione della allocazione e deallocazione della memoria, la dichiarazione del tipo delle variabili e conversione tra tipi.

In generale un linguaggio di scripting permette a chi lo utilizza di trascurare i dettagli implementativi e i tecnicismi per concentrarsi esclusivamente sul problema da risolvere.

Meraviglioso direte voi. Ma ogni cosa ha il suo prezzo e questo prezzo nel caso dei linguaggi di scripting sta nelle prestazioni che risultano non adatte a programmi che utilizzano massicciamente il processore per compiti di calcolo.

Capitolo 3

La Memoria

3.1 Organizzazione della memoria

Iniziamo ora la serie di argomenti legati al funzionamento dei linguaggi di programmazione. Cercherò come al solito di limitare al minimo i tecnicismi soprattutto ora che la faccenda comincia a farsi complicata.

Cominceremo dal più importante, vasto e primordiale aspetto: la memoria.

E' il più importante perché è il programma stesso a risiedere in memoria e qualunque operazione vi salti in mente di fare necessita di accedere ad essa. È il più vasto perché come vedremo esistono vari aspetti da tenere in considerazione. Infine è primordiale perché è comune a TUTTI i linguaggi di programmazione siano essi procedurali, a oggetti, di scripting, etc...

3.1.1 Organizzazione

La memoria è un archivio con tanti cassetti. Ogni cassetto può contenere al massimo un byte (ovvero 8bit) di informazioni ed è individuato da un numero preciso chiamato indirizzo.

Le informazioni nei cassetti prendono il nome di *dati*. In generale un dato non è in un solo cassetto ma è diviso in 1,2,4 o 8 cassetti adiacenti cosicché il processore deve sempre sapere la dimensione base del dato che vuole leggere e ordinerà alla memoria cose del tipo Dammi i 2 cassetti a partire dall'indirizzo XYZ oppure Metti questo dato nei 4 cassetti a partire dall'indirizzo ABC.

Un dato di 1 byte è chiamato (appunto) Byte, un dato che occupa 2 byte è chiamato Word mentre un dato di 4 byte è chiamato DWord (Double Word). Qui incontriamo la prima ambiguità. Per Word infatti si indica l'unità base di un processore. La dimensione in bit di una Word varia quindi da processore a processore: in un processore a 32bit la Word è di 32bit, in un processore a 64bit la Word è di 64bit. Storicamente però una Word assume il valore di 16bit come eredità dei processori a 16bit.

Io interpreterò da ora in poi una Word in base al suo valore di definizione riferendomi cioè all'unità base del processore.

3.1.2 Le variabili

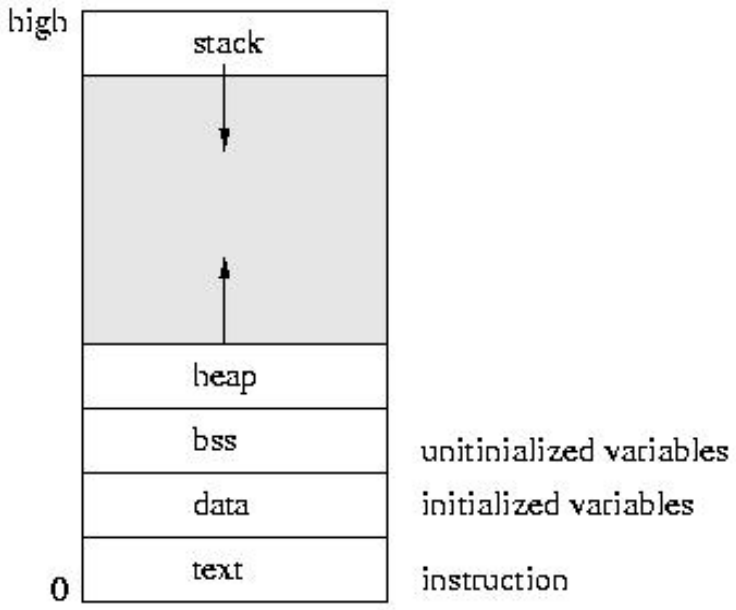
I linguaggi di programmazione utilizzano per l'accesso alla memoria le variabili. Esse non sono altro che nomi mnemonici a quei cassetti della memoria. La variabile "anni" è sicuramente più leggibile di qualcosa del tipo 0xA3B577FF.

Le variabili hanno quasi sempre un tipo. Il tipo, oltre alle sue funzioni sintattiche, serve ad indicare al calcolatore la dimensione del dato: gli interi occupano 4byte, i caratteri occupano 1byte, i float (numeri in virgola mobile) occupano 4byte e cosi via.

Le variabili sono alla base della programmazione: qualunque linguaggio (ad esclusione dei linguaggi funzionali) utilizza le variabili per accedere alla memoria.

3.1.3 Heap, Stack e Text

Ora entriamo un altro pochino nel dettaglio. Quando un programma è in esecuzione sorge la necessità di organizzare un po questi cassetti (tenere un archivio in ordine è di aiuto anche per un computer). In particolare vorremmo almeno dividere i cassetti in modo tale che il testo del programma sia separato dai cassetti che useremo per memorizzare le variabili.



Per questo motivo la memoria durante l'esecuzione viene divisa in tre parti:

- **Text**: La zona in cui c'è il testo del nostro programma ovvero la lista delle istruzioni che il processore eseguirà.
- **Stack**: La zona in cui vengono memorizzate le informazioni delle funzioni e le variabili statiche e locali.
- **Heap**: La zona in cui vengono memorizzate tutte le variabili dinamiche (e in un certo qual modo le variabili globali).

Riguardo al Text non c'è nulla di più da dire.

Lo stack (in italiano pila) può essere immaginato come una pila di block notes in cui noi possiamo leggere solo quello affiorante. Ogni qual volta in un programma si chiama una funzione, si prende un nuovo block notes e sulla prima pagina si annota l'ultima istruzione eseguita (per poterci tornare appena finita la funzione) e poi si lasciano bianche un certo numero di pagine corrispondenti alle variabili locali (ovvero proprie della funzione) e ai parametri della funzione.

Appare quindi evidente che quando è in esecuzione una funzione tutte le variabili locali e i parametri delle funzioni precedenti (sotto il block notes affiorante) sono illeggibili.

Per questo una variabile locale è visibile solo nella funzione in cui compare.

Inoltre nello stack le variabili hanno dimensione fissa e per questo si dicono statiche.

Lo heap (in italiano mucchio) invece può essere visto come una lavagna in cui il programma può appuntare variabili con dimensione variabile, oggetti complessi e soprattutto variabili accessibili a tutte le funzioni. Li ci sono tutte le variabili dinamiche.

Prima di utilizzare un pezzo della memoria dell'heap bisogna allocarla. Il processo di allocazione consiste in:

- "Prenotare" lo spazio necessario.
- Restituire alla funzione chiamante l'indirizzo della zona prenotata.

Una volta che la memoria non serve più la si dealloca in modo da liberare lo spazio.

3.2 Puntatori

Ora che conosciamo la struttura "a cassetti" della memoria e come viene organizzata a tempo di esecuzione possiamo introdurre il concetto dei **puntatori** (o riferimenti).

Questo tipo di variabili sono ben noti a chi programma in C ma esistono in modo meno diretto in decine di linguaggi, come il Java, tanto per dirne qualcuno.

Il concetto espresso dai puntatori è mio avviso semplice anche se poi finisce per complicare la vita a molti. Quindi spero di trasmettervi la "semplicità" con cui riesco a navigare io nel mare dei riferimenti.

Abbiamo visto che una variabile non è altro che un nome mnemonico per individuare il cassetto in cui sono infilati i dati ad essa associati.

Un puntatore, invece, si basa su un altro concetto seppur molto simile. Esso è infatti un nome mnemonico per individuare l'indirizzo del cassetto in cui sono contenuti i dati. Sono quindi variabili che contengono l'indirizzo di una variabile.

Come vedete la differenza è sottile ma allo stesso tempo sostanziale. Un puntatore quindi non contiene il dato ma l'indirizzo in cui è contenuto il dato. Il dato in questione può essere un qualunque dato e persino un altro puntatore.

Perché utilizzare l'indirizzo e complicarsi la vita quando invece potremmo usare direttamente una variabile?

Semplice. Le motivazioni sono fondamentalmente due: a volte è necessario, a volte è dannatamente comodo.

Abbiamo visto infatti nella scorsa lezione delle limitazioni in cui incorrono le variabili, in particolare le variabili di una funzione. Esse sono infatti statiche e locali.

Queste due forti limitazioni possono essere superate grazie ai puntatori. Cominciamo dalla località.

Supponiamo che la funzione XYZ chiami la funzione ABC e vogliamo che ABC abbia come risultato quello di modificare tre variabili locali di XYZ. Proviamo quindi a passare come parametri di ABC le tre variabili. Le modifichiamo con ABC e poi torniamo in XYZ... sorpresa! Le variabili non sono cambiate!

Perchè? E' semplice. Come abbiamo detto in precedenza quando viene invocata una funzione si prende un nuovo "block notes" in pila e si copiano delle informazioni della funzione chiamante ma soprattutto si copiano i valori dei parametri.

E' questo il punto! In ABC non abbiamo le variabili originarie ma copie e qualunque modifica la faremo su queste copie (che verranno inesorabilmente distrutte non appena la funzione ritornerà).

Allora come fare? Ovvio, passando i puntatori alle tre variabili da modificare come parametro di ABC. In questo modo ABC avrà una copia dell'indirizzo in cui sono le 3 variabili. Ma l'indirizzo punta esattamente all'indirizzo delle variabili di XYZ! Quindi qualunque modifica si riperquoterà anche sulle variabili di XYZ perchè in realtà sono la stessa variabile.

Questa gestione dei puntatori però è propria di quei linguaggi di medio-

alto livello come C ed è difficile trovare altri linguaggi che permettano un cosi capillare uso dei puntatori.

I puntatori sono invece necessari quando trattiamo oggetti dinamici.

Gli indirizzi di memoria hanno il vantaggio di essere sempre statici (un indirizzo ha sempre 32bit). Quindi se una variabile non riesce a trattare gli oggetti dinamici ci riesce benissimo il puntatore poichè in realtà lui lavora con l'indirizzo in cui comincia l'oggetto dinamico.

Quindi qualunque oggetto dinamico viene rappresentato sempre con un puntatore. Anche gli oggetti in Java sono in realtà puntatori. Oppure le liste in python. Tutti gli oggetti dinamici vengono trattati come puntatori.

Consideriamo una funzione QWERTY che ha come scopo quello di prendere due liste di numeri ed unirla in un unica grande lista. Ovviamente non possiamo scrivere questa funzione con parametri statici, infatti non sappiamo la dimensione delle liste che ci verranno passate e, inoltre, **non ci interessa saperlo!** Noi vogliamo che la funzione QWERTY unisca liste di qualunque dimensione e non liste di dimensione predeterminata.

Per risolvere questo problema sappiamo già cosa fare: non passiamo come attributi le liste in se ma passiamo i puntatori alle due liste. In questo modo la funzione potrà raggiungere le locazioni di memoria in cui si trovano le liste ed operare sulle stesse a piacimento.

Sebbene possiate scordarvi del primo esempio a meno che non vogliate scrivere in C, non potete assolutamente scordarvi quest'ultimo concetto in quanto ve lo ritroverete fra i piedi in ogni linguaggio.

Concludo con un esempio e una considerazione che spero chiarisca l'idea e vi faccia notare quanto è naturale il concetto dei puntatori.

Nella vita di tutti i giorni usiamo i riferimenti per tutto: quando parliamo di un libro diciamo il suo titolo il quale non è altro che un riferimento al libro in se. Se io cambio il titolo non si modifica il libro... semplicemente mi riferisco ad un altro (che può anche non esistere). Quando voglio indicare casa mia dico il suo indirizzo. Se io dico un altro indirizzo non si sposta la casa... semplicemente penso ad un altra casa.

3.3 Tipi di Dato

Abbiamo accennato anche cosa sono i riferimenti (o puntatori): variabili che invece di contenere il dato contengono l'indirizzo in cui trovare il dato.

Abbiamo però visto anche che le variabili conoscono anche il tipo di dato a cui si riferiscono o che contengono. Essi si riferiscono infatti all'insieme a cui appartiene il dato. Esempi di tipi sono gli interi, i caratteri, i booleani (vero o falso) e i float (numeri con virgola mobile).

Questo è l'ultimo argomento forte riguardante la memoria e lo affronteremo subito.

Bisogna far subito presente che in origine i tipi non esistono. I tipi sono

strutture create dai linguaggi e come tali scompaiono a linguaggio macchina. Come abbiamo visto due lezioni fa, sono utilizzati principalmente per istruire il compilatore sulla dimensione del dato. Il tipo è quindi un astrazione utile all'interpretazione dei dati. Non è però necessaria, infatti i linguaggi possono essere tipizzati o non tipizzati.

I linguaggi assembly sono un ottimo esempio di **linguaggi non tipizzati**. In questo tipo di linguaggi infatti l'interpretazione del tipo di dato è lasciata al programmatore. L'assembly infatti macina solo gruppi di zero e di uno indipendentemente da cosa rappresentino.

Il resto dei linguaggi esistente fa parte dei **linguaggi tipizzati**. In questa categoria di linguaggi ad ogni variabile viene associato una annotazione di tipo. Esistono due aspetti con cui queste annotazioni vengono gestite ed ogni aspetto genera due sotto-categorie.

Il primo aspetto riguarda il controllo dei tipi (o *type checking*) ovvero il procedimento che permette di verificare se i vincoli imposti dai tipi sono soddisfatti. Questo controllo viene effettuato in fase di compilazione (*static check*) oppure in fase di esecuzione (*dynamic check*).

Da questo aspetto i linguaggi si dividono in *strong typing* (o fortemente tipizzati) e *soft typing* (o debolmente tipizzati).

I linguaggi strong typing sono caratterizzati da rigidi controlli sui tipi come:

static check (per i linguaggi compilati) - controllo dei tipi in fase di compilazione;

cast - impossibilità di eseguire conversioni di tipo implicite;

type safety - ovvero sicurezza riguardo ai tipi con controlli anche in fase di esecuzione.

I linguaggi soft typing invece contravvengono a queste tre regole e, al caso limite, si riducono a linguaggi non tipizzati.

Il secondo aspetto riguarda invece la vita della variabile. In poche parole riguarda la possibilità di una variabile di cambiare o meno tipo durante la sua esistenza.

Si dividono quindi in linguaggi a tipizzazione statica e linguaggi a tipizzazione dinamica.

Nei primi una variabile non può cambiare il suo tipo per nessun motivo. Esempi di questi linguaggi sono C/C++ e Java.

Nei secondi invece una variabile può, nel corso dell'esecuzione del programma, variare il tipo a cui appartiene. Linguaggi a tipizzazione dinamica sono fra i tanti Python, Fortran e Visual Basic.

Con questo abbiamo concluso la nostra panoramica sulla memoria. Abbiamo visto come è organizzata, come si "gioca" con essa e come il linguaggio ci permette di interpretare i dati.

Capitolo 4

La CPU

TODO

Parte II

Elementi Comuni

Capitolo 5

Gestione dei flussi

5.1 Salti Condizionati

I *salti condizionati* sono l'elemento fondamentale di ogni linguaggio di programmazione. Troviamo salti condizionati in ogni linguaggio, dall'assembly al python, dal C al più moderno C#.

Queste istruzioni hanno la funzione di modificare il flusso del programma in base al verificarsi o meno di una condizione. Modificare il flusso, come immaginate, è fondamentale per scrivere programmi interattivi (ovvero lontani dagli script di sistema) in cui il programma deve adattarsi alle richieste che l'utente impartisce tramite tastiera o mouse.

I salti condizionati, nei moderni linguaggi di programmazione, si dividono fondamentalmente in due categorie:

Salti Condizionati in Avanti - Sono i salti condizionati che formano le istruzioni IF-THEN-ELSE e SWITCH CASE. Sono associate al concetto di *divisione* del flusso. Usando un salto condizionato in avanti, infatti, non si separano due zone di codice e solamente una delle due verrà eseguita dopo il salto.

Salti Condizionati in Indietro - Sono i salti condizionati che formano le istruzioni FOR e WHILE. Sono associate al concetto di *ciclo*. Tali salti infatti provocano la ripetizione di codice già eseguito.

5.1.1 IF-THEN-ELSE

L'istruzione *if-then-else* è sicuramente l'istruzione di salto più conosciuta. E' l'unica ad essere implementata in tutti i linguaggi di alto livello.

$$IfThenElse(condizione, then, else)$$

Tale istruzione è un istruzione ternaria, prende cioè, tre parametri.

condizione - È la condizione di salto. Può assumere solo due valori.

then - È l'istruzione eseguita se la condizione risulta soddisfatta.

else - È l'istruzione eseguita se la condizione NON è soddisfatta.

Nei linguaggi di programmazione più comuni ha solitamente una sinstassi del tipo:

```
if (condizione)
    istruzione1
else
    istruzione2
```

Viene quindi implicitata la locuzione "then".

5.1.2 SWITCH-CASE

Lo *switch-case* è un altra istruzione molto comune che ha lo scopo di semplificare il codice nel caso di numerosi *if* a catena. Esso infatti associa ad ogni valore della variabile un istruzione corrispondente.

Sopponiamo infatti di voler eseguire un istruzione a seconda del valore di una variabile: se la variabile vale 0 facciamo una cosa, se vale 1 ne facciamo un'altra, se vale 2 un'altra ancora e così via. Questo si traduce a livello di codice in una cosa del tipo:

```
if (variabile=0)
    istruzione1
else if (variabile=1)
         istruzione2
     else if (variabile=2)
         istruzione3
[...]
```

Ci accorgiamo quindi che il codice rischia di diventare poco leggibile. Lo switch-case invece riduce questo codice a

```
switch (variabile)
    case 0 : istruzione1
    case 1 : istruzione2
    case 2 : istruzione3
[...]
```

Un codice vistosamente più pulito.

5.1.3 WHILE

L'istruzione *while* è un altro esempio di salto condizionato. E' l'istruzione fondamentale associata al ciclo. Tale istruzione infatti ripete una serie di istruzioni finché la condizione rimane vera.

Ha una forma del tipo:

```
while (condizione)
    istruzione
```

Il rischio di questo genere di istruzione è quello di non rendere mai false la condizione. Se la condizione non diventa mai falsa il programma si inbatte in un ciclo infinito e, in pratica, si blocca.

5.1.4 FOR

5.2 Salti Incondizionati

I salti incondizionati puri non esistono in nessun linguaggio. Ad esclusione del deprecato *goto* infatti, non esiste un istruzione che permetta di saltare ovunque nel testo automaticamente.