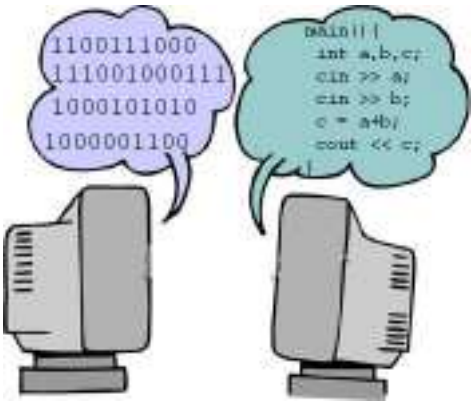
LinPROG

introduzione ai linguaggi di programmazione

(2011.11)



Indice

Premessa e pre-requisiti

Questi appunti hanno lo scopo di introdurre al mondo dei linguaggi di programmazione sia riconducendoli, come anche le classificazioni nella letteratura del settore, alla data di nascita che mettendone in evidenza le diversità per mezzo del confronto di frammenti di codice.

Gli appunti non hanno alcuna pretesa esaustiva, ma solo di mostrare le caratteristiche, le differenze e il modo di esecuzione di programmi scritti sia in linguaggi tradizionali che in linguaggi di scripting, attraverso un cammino fra alcuni linguaggi significativi che è anche un cammino attraverso lo studio delle tecniche di programmazione per come si sono evolute. Naturalmente, per come è naturale in questi casi, la scelta dei linguaggi trattati può essere *faziosa* nel senso che è dettata da scelte di natura didattica che, come tali, sono personali.

Si presuppone una certa conoscenza della struttura delle istruzioni macchina e di come viene eseguito, da una CPU, un programma in linguaggio macchina (vedere *PC inside*). Un minimo di conoscenza di un linguaggio di programmazione (per esempio C++), anche se non indispensabile, è di aiuto nella comprensione, soprattutto, delle analisi comparative dei linguaggi presentati. Anche la conoscenza generale delle reti Client/Server, e di HTML, può concorrere ad una maggiore comprensione della parte nella quale vengono trattati i linguaggi di scripting.

Comunicare con il computer

Come noto il computer è un sistema a stati binari, intendo con questo che tutti i dati che viaggiano al suo interno sono formati da stringhe binarie, sequenze di 0 e 1. Anche le istruzioni che compongono un programma che deve essere eseguito, sono in tale formato.

Per quanto riguarda le istruzioni che possono essere utilizzate per scrivere un programma, il loro uso presuppone la conoscenza del set di istruzione della CPU e, in ogni caso, una volta scritto il programma per quella CPU, questo non è eseguibile da una CPU con diverso set di istruzione.

Dopo la fase iniziale durante la quale programmare un computer significava modificare collegamenti elettrici, si arrivò ad avere dei dispositivi che permettevano l'introduzione dei codici esadecimali rappresentanti le istruzioni del programma da eseguire. Si trattava di un grosso passo avanti infatti il codice esadecimale, per il programmatore, è più facilmente comprensibile di interminabili, e monotone, sequenze di 0 e 1, ma restano due problemi insormontabili per chi vuole comunicare con il computer:

- Le sequenze binarie sono l'ambiente naturale per un computer, ma sono completamente incomprensibili per l'uomo. Anche se si sostituiscono i numeri binari con il loro equivalente esadecimale, la comprensibilità di un programma non guadagna di molto. Certamente è più comune invertire due bit e quindi commettere un errore difficilmente rintracciabile (è una impresa ardua rintracciare all'interno di lunghissime sequenze di 0 e 1 i bit invertiti) che scrivere in modo errato un codice esadecimale, se non altro perché è composto da simboli diversi. La comprensibilità in ogni caso non migliora di molto: come si può, infatti, rintracciare facilmente dove è, per esempio, la parte di codice in cui si effettua una determinata operazione?
- La sequenza delle istruzioni da fare eseguire è quella che sta alla base del funzionamento di un elaboratore. Il programmatore deve tradurre l'algoritmo in termini di byte, somme fra byte ecc...

Con queste premesse scrivere un programma è una cosa molto difficoltosa. Si aggiunga a questo che se, all'inizio, coloro che scrivevano i programmi erano le stesse persone che avevano progettato

e costruito il computer, col passare del tempo e la diffusione dei computer il programmatore diventa un soggetto diverso rispetto al progettista hardware.

Occorreva inventarsi un modo di scrivere i programmi con un linguaggio più comprensibile e, possibilmente, in maniera vicina a come un programmatore concepisce un algoritmo per la risoluzione di un problema. Naturalmente il computer continua a comprendere soltanto segnali binari, ma, ecco l'idea, si può progettare un programma che, a partire da un file di testo contenete le istruzioni di un programma scritte in modo comprensibile a una lettura da parte di una persona, traduca detto programma nella forma comprensibile ad una macchina. Naturalmente bisogna stabilire le regole da adottare per la scrittura del programma in modo da progettare un traduttore, che è pur sempre un programma, in grado di tradurre a partire da regole di scrittura ben fissate: nascono i linguaggi di programmazione.

Che cos'è un linguaggio di programmazione

Un linguaggio di programmazione è un linguaggio dotato di un insieme di regole per scrivere programmi per computer, ovvero un insieme di istruzioni che a partire da un insieme di dati di input, applicando una funzione di trasformazione descritta, appunto, dalle istruzioni, produca un insieme di dati di output.

In un linguaggio di programmazione si possono distinguere:

- L'insieme delle **parole chiavi**: un insieme di parole che hanno un significato particolare per chi deve tradurle in istruzioni eseguibili o deve eseguirle e a ciascuna delle quali corrisponde una azione ben definita. Le parole chiavi non possono essere usate se non per indicare le azioni ad esse associate. Per esempio, in C++, non si può definire una variabile con nome `main`, essendo questa una parola chiave a cui corrisponde il nome della funzione che viene eseguita all'avvio del programma. In definitiva il linguaggio deve essere non ambiguo.
- I **caratteri speciali**: un insieme di caratteri con significati particolari. Per esempio, in C++, sono caratteri speciali: il carattere punto e virgola (;) che chiude una istruzione o il carattere virgola (,) che delimita i componenti di un elenco o il carattere spazio che distingue un componente (una parola) dall'altro. Anche in questo caso, tali caratteri, non possono essere utilizzati se non con quel significato particolare.
- Un insieme di **regole sintattiche**: un modo di mettere assieme le parole così da formare frasi per indicare le azioni che devono essere compiute.

Una volta note le caratteristiche di un linguaggio di programmazione, si può tradurre un algoritmo in frasi (affermazioni, ***statement***) di quel linguaggio.

Dal sorgente all'eseguibile

Il programma scritto utilizzando le regole di un determinato linguaggio di programmazione, viene chiamato ***programma sorgente***.

Per poter produrre un file in una memoria di massa contenete il sorgente di un programma, in genere, si utilizza un editor: un programma cioè che salva su memoria di massa un file di *testo puro* senza formattazioni (colori, grassetto, sottolineato, allineamenti, ecc...). Le formattazioni, infatti, non hanno alcuna influenza sull'esecuzione di un programma. Anche se, per la produzione del sorgente, può andare bene qualsiasi software in grado di produrre file di testo puro, esistono editor

specifici per programmatori che forniscono tutte le agevolazioni necessarie per la fase di generazione di un sorgente.

Qualsiasi sia lo strumento utilizzato per produrre il sorgente, lo scopo finale della scrittura di un programma è quello della sua esecuzione. Sono possibili due approcci diversi all'esecuzione di un programma:

- La **compilazione**. In questo caso il programma sorgente viene, da un *compilatore*, tradotto in istruzioni macchina eseguibili da una CPU.

  Il compilatore effettua diverse elaborazioni sul sorgente al fine di produrre il codice eseguibile. Si comincia da una fase di ***analisi*** in cui il compilatore acquisisce informazioni sulle istruzioni in modo da essere in grado di tradurle. In questa fase il compilatore ricava dal sorgente gli *atomi* (le singole componenti delle varie istruzioni) al fine di una loro classificazione, per esempio, come parole chiavi o come variabili scelte dal programmatore, verifica la correttezza sintattica degli statement. Successivamente si comincia a a produrre l'oggetto in cui le istruzioni del sorgente sono sostituite da parti direttamente in linguaggio macchina o da riferimenti a parti contenute nelle librerie del compilatore (componenti software già tradotti in linguaggio macchina). La fase successiva si occupa delle interconnessioni fra i moduli. Il programma generato nella fase precedente va *collegato* a moduli delle librerie o ad eventuali altri moduli software. Dei collegamenti se ne occupa un modulo software chiamato ***linker***.

  Il collegamento può essere effettuato in due modi: *statico* e *dinamico*. Nel primo caso (collegamento statico) tutti i programmi e le librerie necessarie sono incluse nell'eseguibile che risulta di grandi dimensioni, ma contiene tutto ciò di cui ha bisogno. Nel secondo caso (collegamento dinamico) l'eseguibile ha dimensioni minori rispetto al primo caso e le librerie sono caricate in memoria quando c'è necessità di utilizzarle. Le librerie esterne sono chiamate DLL (Dynamic Link Libraries) in ambiente Windows e SO (Shared Object) in ambiente Linux. Naturalmente l'installazione di un programma comporta anche l'installazione delle librerie necessarie al suo funzionamento, ma, in questo modo, si ha la possibilità di aggiornare le librerie senza che questo implichi la ricompilazione del programma e, inoltre, possono coesistere anche diverse versioni delle librerie.

  In ogni caso, alla fine del processo, si avrà un programma in formato binario eseguibile la cui esecuzione può essere lanciata dal Sistema Operativo.

- L'**interpretazione**. In questo caso non esiste una traduzione del codice sorgente. Il programma viene eseguito direttamente dall'interprete. L'interprete è, sostanzialmente, un programma che esegue altri programmi. Laddove un compilatore è un traduttore, un interprete è invece un esecutore. Nella macchina sulla quale eseguire il programma, si lancia l'interprete che poi leggerà ed eseguirà le istruzioni contenute nel sorgente.

In linea generale per ogni linguaggio di programmazione potrebbe esistere sia un compilatore che un interprete, anche se alcuni linguaggi nascono per essere compilati ed altri per essere interpretati. Esistono anche situazioni ibride.

Compilazione e interpretazione

In passato lo schema predominante per la traduzione e l'esecuzione di un programma era lo schema compilativo. Tutti i linguaggi di programmazione adottavano tale schema:

- il programma viene compilato su una determinata **piattaforma** (risorse hardware, principalmente la CPU, e sistema operativo).
- L'eseguibile risulta velocissimo perché sfrutta le caratteristiche peculiari dell'hardware.

La forza principale dello schema compilativo è anche il suo limite: la trasportabilità. Tutte le volte che si vuole adattare il programma ad un nuovo hardware o Sistema Operativo è necessario, nella migliore delle ipotesi, ricompilare il programma per la nuova piattaforma.

Per risolvere il problema delle dipendenze del programma da una piattaforma, si è adottato uno schema che, basandosi su librerie compilate per le varie piattaforme, potesse eseguire il codice senza necessità di ricompilazione. L'esecuzione del programma è più lenta e si richiedono più risorse di memoria: quelle necessarie per l'interprete e per il programma, ma non c'è necessità di ricompilare il programma che diventa portabile. La maggiore lentezza di un programma interpretato rispetto ad uno compilato è dovuta anche al fatto che, nel caso di un interprete, la CPU deve eseguire l'interprete e il programma. Si pensi, inoltre, a come verrebbe eseguita una struttura ciclica nello schema compilativo e in quello interpretativo.

- Schema compilativo: il programma viene tradotto in linguaggio macchina e una istruzione presente nel corpo del ciclo può essere eseguita tutte le volte richieste e il tempo di esecuzione dipende soltanto dai cicli macchina necessari per l'esecuzione di quella istruzione.
- Schema interpretativo: una istruzione presente all'interno del ciclo viene controllata e interpretata ogni volta che se ne richiede la sua esecuzione.

In conseguenza delle peculiarità dei due schemi presentati si utilizza lo schema compilativo in ambienti in cui la velocità o le dimensioni sono fattori critici e si utilizza lo schema interpretativo in fase di sviluppo di un programma quando, per la sua messa a punto, si risparmia il tempo di nuove compilazioni oppure quando la velocità non è un fattore critico e, invece, è importante la portabilità: si pensi, per esempio, alle applicazioni web.

L'utilizzo di ciascuno dei due schemi esposti comporta vantaggi e svantaggi, anche in relazione allo stato di avanzamento dello sviluppo di una applicazione:

- Schema compilativo: adatto per preparare il programma all'esecuzione nell'ambiente cui è destinato. Se cambia la piattaforma, il programma deve essere ricompilato.
- Schema interpretativo: adatto alla fase di sviluppo di un programma dove non è importante la velocità ma il test sull'algoritmo per la correttezza del programma. Il programma è indipendente dalla piattaforma. L'esecuzione del programma è molto più lenta dell'equivalente compilato.

Nel tempo sono stati tentati anche degli ibridi allo scopo di poter sfruttare i vantaggi dei due schemi esposti. Un tipico esempio in cui viene adottato un approccio di questo genere lo si trova per il linguaggio Java. In questo caso viene utilizzato un compilatore che produce codice in linguaggio intermedio (codice per una CPU generica) che viene chiamato ***bytecode***. In questo modo il programma è portabile: un programma in bytecode è eseguito da un altro programma (l'interprete) che viene anche chiamato ***macchina virtuale***. Nel caso di Java si avrà la JVM (Java Virtual Machine).

Per cercare di ridurre la differenza di prestazione fra un programma compilato e uno interpretato sono state introdotte tecniche di ***compilazione just-in time***. In questi casi, in sede di esecuzione, il bytecode invece di essere interpretato, viene passato ad un compilatore che lo traduce in linguaggio

macchina subito prima dell'esecuzione.

Classificazioni dei linguaggi

Nel corso degli anni sono stati progettati moltissimi linguaggi di programmazione. Alcuni conservano la loro validità nel corso degli anni grazie anche alla quantità di software sviluppato con essi, e ancora utilizzato. Altri sono stati creati per scopi particolari o per sperimentare determinate tecnologie. Di altri infine si è persa ogni traccia.

In generale la comparazione fra i vari linguaggi tiene conto di due fattori:

- L'**evoluzione**. Non si tratta soltanto di un *fatto anagrafico* dei linguaggi, ma del rapporto fra il linguaggio e il modo con cui un algoritmo è concepito dal programmatore e adattato per poter utilizzare le strutture disponibili nel linguaggio stesso. Un linguaggio è più o meno evoluto se è possibile, più o meno facilmente, tradurre in quel linguaggio l'algoritmo. Per fare un esempio esplicativo si immagini un linguaggio che preveda l'istruzione per eseguire soltanto l'operazione di somma aritmetica ed uno che preveda anche la moltiplicazione. Se si utilizza il primo occorrerà tradurre la moltiplicazione come serie di somme. Ciò che per il programmatore è una singola operazione (la moltiplicazione) deve essere tradotta in una serie di operazioni in ragione degli strumenti disponibili nel linguaggio. Seguendo l'esempio si potrà affermare che il secondo linguaggio è più ***evoluto*** del primo, o, anche, ***a più alto livello***. In base all'evoluzione, i linguaggi di programmazione si classificano in **generazioni**.
- Il **paradigma di programmazione** supportato. Un paradigma di programmazione è un insieme di strumenti forniti da un linguaggio di programmazione e definisce un insieme di regole per il modo con cui scrivere un programma. Un paradigma di programmazione nasce come evoluzione di un altro o, certe volte, rendendo obbligatorie le regole di buona programmazione affermatisi nella fase precedente.

Nascita dei linguaggi: l'Assembly

Il primo linguaggio di programmazione, riconosciuto come tale, è il linguaggio **Assembly**. Naturalmente il computer non è in grado di eseguire un programma se non in linguaggio macchina. Il sorgente Assembly è necessario che venga convertito in linguaggio oggetto, utilizzando l'**Assembler** che, appunto, ha questo compito.

Per poterne esaminare le caratteristiche, si riporta un esempio di programma scritto in Assembly.

```
section .bss
        somma:  resw 1  ;variabile per risultato

section .text
    global _start       ;necessario al linker

_start:                 ;entry point del programma

        mov  eax,2      ;mette 2 in un registro della CPU
        mov  ebx,3      ;mette 3 in un registro della CPU
        add  eax,ebx    ;somma i due numeri
        mov  somma,eax  ;conserva in memoria il risultato
        mov  eax,1      ;chiamata al sistema (sys_exit)
        int  80h        ;chiama il kernel
```

Da un esame, anche generale, del listato riportato si può notare:

- L'uso, qui per la prima volta, delle *variabili* (nell'esempio `somma`). Invece di usare indirizzi per rintracciare la word in cui si conserva il risultato della somma, si assegna alla locazione un nome simbolico che la individua in modo univoco. È questa caratteristica che fa individuare l'Assembly come primo linguaggio di programmazione.
- L'uso di *mnemonici* al posto, nelle istruzioni, dei codici operativi. La combinazione di lettere utilizzata agevola il programmatore nella comprensione dell'operazione eseguita. Per esempio è più facile ricordare che `add` è il codice operativo di una somma invece di un codice, anche, esadecimale.
- L'uso di nomi simbolici negli operandi, anche per indicare i registri interni della CPU.

Il modo di scrivere il programma, rispetto anche all'introduzione di codici esadecimali, è cambiato: è più semplice comprendere quello che fa il programma. L'uso intensivo dei commenti agevola ulteriormente la comprensione.

L'aspetto negativo, dal punto di vista della comprensibilità, è costituito dal fatto che, anche se scritte in modo diverso ad ogni istruzione in linguaggio Assembly corrisponde una istruzione in linguaggio macchina. Il programmatore è costretto a tradurre l'algoritmo, per come lo concepisce lui, in sequenze di istruzioni che seguono la logica della macchina: l'Assembly è, infatti, una rappresentazione simbolica del linguaggio macchina e, per questo motivo, viene considerato un **linguaggio a basso livello**, intendendo in tale modo affermarne la vicinanza con il linguaggio macchina. Spesso si dice che l'Assembly è un linguaggio di seconda generazione.

Un'altra caratteristica peculiare dell'Assembly è il fatto che, più che di un linguaggio bisognerebbe parlare, più propriamente, di un insieme di dialetti. Essendo, infatti legato indissolubilmente al linguaggio macchina, varia da CPU a CPU. Per esempio il linguaggio macchina della CPU1 potrebbe supportare operazioni a tre operandi che, invece, non sono supportate dalla CPU2. Inoltre il legame che un programma ha con il Sistema Operativo, comporta nel caso della scrittura di un programma in Assembly, la conoscenza dei dettagli delle comunicazioni fra il programma stesso e i servizi del Sistema Operativo. Il programma di esempio termina chiamando il Sistema Operativo che riprende il controllo della macchina. La chiamata ha codice 80h in ambiente Linux, ma, per esempio, 21h in ambiente DOS e, inoltre, le convenzioni di chiamata (la penultima istruzione dell'esempio) cambiano.

Nonostante le evidenziate difficoltà nel suo utilizzo, l'Assembly, essendo l'unico linguaggio con una corrispondenza 1:1 con il linguaggio macchina, è tuttora utilizzato principalmente in ambienti in cui la velocità di esecuzione è un fattore critico. Se occorre, infatti, sfruttare tutte le potenzialità di una macchina è necessario utilizzare un linguaggio che *parli* direttamente alle sue componenti.

I linguaggi di Terza Generazione

Gli sforzi volti a cercare di inventare linguaggi di programmazione che mettessero a disposizione strumenti sempre più potenti e semplici per la codifica di algoritmi, ha portato alla fine della metà del secolo scorso alla nascita di due linguaggi, di concezione diversa rispetto all'Assembly, e dedicati alla scrittura di algoritmi riguardanti la risoluzione di problemi legati alle applicazioni dei computer in quegli anni: il **Fortran** (1954) per lo sviluppo di applicazioni scientifiche e il **COBOL** (1961) per lo sviluppo di applicazioni gestionali, tuttora, per la loro solidità, utilizzati. I nomi sono

acronimi, rispettivamente, di Formula Translation e Common Business Oriented Language.

Si tratta dei capostipiti della famiglia dei cosiddetti linguaggi **3GL** (Third Generation Language, linguaggi di Terza Generazione).

```
IDENTIFICATION DIVISION.
PROGRAM-ID. SOMMA.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER. COBOL85.
OBJECT-COMPUTER. COBOL85.

DATA DIVISION.
WORKING-STORAGE SECTION.
77   NUM1  PIC 99.
77   NUM2  PIC 99.
77   SOMMA PIC 99.

PROCEDURE DIVISION.
MAIN SECTION.
     DISPLAY "Primo numero da sommare "
     ACCEPT NUM1.
     DISPLAY "Secondo numero da sommare "
     ACCEPT NUM2.
     ADD NUM1 NUM2 GIVING SOMMA.
     DISPLAY "Somma dei numeri " SOMMA.
FI-MAIN. STOP RUN.
```

L'esame delle caratteristiche generali del listato riportato, anche se la natura del COBOL è quella, come osservato, di applicazioni gestionali che utilizzano file su memorie di massa, e quindi di programmi molto diversi da quello di esempio, mette in evidenza le peculiarità dei linguaggi di terza generazione.

- Nella dichiarazione delle variabili (nella `DATA DIVISION` nel COBOL) le variabili non fanno più riferimento a byte o word come è richiesto dall'Assembly, ma, in forma più generale, si fa riferimento a *tipi*: numerici o alfanumerici. Ci si astrae dalle esigenze della macchina e ci si avvicina alla logica di chi scrive il programma che, per esempio, se deve scrivere l'istruzione per effettuare una somma, la scrive prescindendo dallo spazio riservato alle variabili o dal tipo specificato di variabili (intere o virgola mobile).
- Le istruzioni, come quella per la somma presente nel programma di esempio, sono codificate in modo da evidenziare il tipo di operazione da fare e come essa si concretizza nella mente del programmatore più che come viene conservata nella memoria di un computer. La corrispondenza con il linguaggio macchina non è 1:1 come nell'Assembly, ma 1:n. Ad una istruzione di un linguaggio 3GL corrisponde un blocco di istruzioni macchina.

La terza generazione dei linguaggi di programmazione è chiamata anche la generazione dei **linguaggi procedurali**. Con questo termine si intende un paradigma di programmazione che prevede di sviluppare il programma come composto da un insieme di istruzioni raggruppate e delimitate da opportuni caratteri o parole chiavi. In ogni blocco di codice sono descritte le azioni che deve svolgere il computer per conseguire i risultati attesi.

I linguaggi di programmazione appartenenti a questa generazione nascono come strumenti messi a disposizione per scrivere programmi di una certa categoria: il Fortran per le applicazioni

scientifiche, che hanno la caratteristica di richiedere molti calcoli su un insieme ristretto di dati, il COBOL per quelle gestionali che invece hanno caratteristiche opposte: pochi calcoli ma su una massa enorme di dati. Per la loro natura specialistica questi linguaggi vengono chiamati anche **algoritmici**.

Mano a mano che si ampliavano i campi di applicazione dei computer, nascevano nuovi linguaggi di programmazione che mettevano a disposizione strumenti specialistici per la risoluzione di algoritmi di una certa classe. Un linguaggio 3GL è composto da un insieme finito di parole chiavi scelte in modo da rendere semplice la scrittura di alcuni algoritmi. Per esempio in Fortran esistono istruzioni per il calcolo della radice quadrata o per il calcolo del seno di un angolo, istruzioni che non esistono, invece, in COBOL: chi sviluppa programmi gestionali non ha infatti alcun bisogno del calcolo di radici quadrate o del seno di un angolo, cosa invece comune per chi sviluppa applicazioni di carattere scientifico. Per chi ha necessità di utilizzare applicazioni di altro tipo occorrono altri strumenti; ecco quindi la nascita di speciali linguaggi per particolari applicazioni.

La programmazione strutturata

Fra i 3GL un linguaggio che ebbe una notevole diffusione (famoso in senso positivo e negativo) e che continua, anche questo, ad essere utilizzato anche se in forme diverse rispetto alle caratteristiche originali, è il **BASIC** (Beginner's All purpose Symbolic Instruction Code). Sviluppato dal 1963, nasce per essere un linguaggio semplice da imparare e per poter essere utilizzato, come evidenziato anche dal nome, dai principianti per la scrittura di programmi generici. Il BASIC comprende istruzioni, derivate dal Fortran, per calcoli scientifici e istruzioni, anche se non specializzate come nel COBOL, per il trattamento di dati residenti su memorie di massa. Gli strumenti resi disponibili dal linguaggio possono essere utilizzati anche da chi non ha buone basi di programmazione. Anche se nasce compilato le sue implementazioni più comuni sono interpretate, anche in ragione del fatto che un linguaggio interpretato si adatta maggiormente, rispetto ad uno compilato, ad essere usato da un principiante.

La diffusione del BASIC con il suo *permissivismo* e, principalmente, la mancanza di regole di buona programmazione portò ben presto a quello che fu chiamato ***spaghetti code***, termine dispregiativo per indicare quei programmi in cui è difficile seguire il flusso delle istruzioni che le compongono. Le istruzioni sono intrecciate come gli spaghetti in un piatto.

```
20 i = 0
30 i = i + 1
40 if i <> 10 then goto 70
50 print "Programma terminato."
60 end
70 print i,  " al quadrato = ",  i * i
80 goto 30
```

Il programma dell'esempio, scritto in BASIC, calcola e visualizza i quadrati dei numeri da 1 a 9, solo che, il continuo rimando ad altre righe contenenti codice (le istruzioni `goto`), rende incomprensibile, e difficile da seguire logicamente, la sequenza delle istruzioni. La manutenzione di programmi che usavano in maniera intensiva l'istruzione di modifica sequenza `goto` diventavano quasi impossibili da mantenere.

Lungo il cammino verso la definizione di regole di buona programmazione, si incontrano due tappe fondamentali:

- l'enunciazione, nel 1966, del *teorema di Böhm-Jacopini* che afferma che qualunque algoritmo si può scrivere utilizzando soltanto tre strutture: la sequenza (elenco delle istruzioni che evidenzia l'ordine di esecuzione), la selezione (la scelta fra due diverse sequenze di istruzioni che vengono eseguite in dipendenza da una condizione che può essere vera o falsa), il ciclo (una sequenza di istruzioni che viene ripetuta finché resta valida una condizione).
- la pubblicazione, nel 1968, dell'articolo *Go To Statement Considered Harmful* del prof. Edsger Dijkstra, dove si attaccava l'uso della famigerata istruzione considerandola emblema di cattiva programmazione, auspicandone la scomparsa dai futuri linguaggi.

Fra gli anni '60 e '70 vengono sviluppati i concetti della programmazione strutturata, paradigma di programmazione nato per applicare le nuove regole. Il prof. Niklaus Wirth pubblica l'articolo *Program Development by Stepwise Refinement* sulla metodologia top down per costruire soluzioni a problemi complessi, e il libro *Algoritmi + Strutture di Dati = Programmi*, testo fondamentale di programmazione.

Nel paradigma della programmazione strutturata si possono utilizzare solo le tre strutture previste dal teorema di Böhm-Jacopini. Ogni struttura deve avere un solo punto di ingresso e un solo punto di uscita (non è più ammesso saltare in qualsiasi punto del programma) e, inoltre, ogni struttura può avere al suo interno solo strutture di controllo del tipo delle tre ammesse.

Il nuovo paradigma poteva essere applicato anche utilizzando gli strumenti resi disponibili dai linguaggi di programmazione esistenti, che consentivano tale paradigma ma ne rendevano l'uso difficoltoso. Occorrevano quindi nuovi linguaggi che supportassero il paradigma rendendo facile la sua applicazione. Wirth progetta, a tale scopo, nel 1970, il **Pascal**: linguaggio sviluppato per la didattica della programmazione.

Alla programmazione strutturata si affianca, come estensione, la **programmazione modulare** basata sullo sviluppo di un programma in singoli moduli indipendenti fra di loro, ognuno con interazione minima con il mondo esterno e con gli altri moduli, e sull'utilizzo di interfacce semplificate per la comunicazione fra i moduli. Wirth estende gli strumenti, resi disponibili dal Pascal, progettando il **Modula-2** che supporta pienamente, come si evince anche dal nome scelto, il paradigma modulare.

Il paradigma della programmazione modulare consente lo sviluppo di linguaggi di programmazione che, anche se rientrano nella famiglia dei linguaggi procedurali, hanno caratteristiche diverse rispetto a quelli esistenti fino a quel momento:

- I linguaggi di terza generazione, esistenti in tempi precedenti la definizione delle regole della programmazione strutturata, presentavano un insieme finito di parole chiavi dedicate alla risoluzione di problemi di una determinata categoria. Se occorrevano elaborazioni diverse da quelle comuni nelle applicazioni, bisognava *arrangiarsi* con quello che c'era o rivolgersi ad un altro linguaggio. Per esempio il trattamento dei file su memorie di massa nel Fortran è abbastanza primitivo: le informazioni possono essere elaborate come insiemi di byte e, d'altra parte, chi sceglie il Fortran lo sceglie per le potenti istruzioni matematiche disponibili. Se serve elaborare molti dati dalle memorie di massa: o ci si arrangia con quello che si ha, aumentando notevolmente il codice da scrivere o si usa un altro linguaggio più specializzato (per esempio il COBOL). Da qui la proliferazione dei linguaggi con notevoli disagi per il programmatore che, nonostante avesse acquisito esperienza nell'utilizzo di un certo linguaggio, era obbligato a imparare un nuovo linguaggio qualora gli occorrevano funzionalità diverse, sprecando il know-

how acquisito.

- I linguaggi di programmazione sviluppati dopo la teorizzazione della programmazione modulare partono da un'altra prospettiva: poche parole chiavi, giusto quelle necessarie per codificare le strutture di controllo e la possibilità di aggiungere moduli con funzioni che espandono il linguaggio specializzandolo per determinate applicazioni.

```
#include <stdio.h>
#include <string.h>                                                    /*1*/

main()
{
  char parola[50];
  int l;

  printf("Calcola lunghezza di una parola \n");
  printf("Introdurre parola ");
  gets(parola);
  l=strlen(parola);                                                    /*2*/

  printf("Introdotta parola di %d caratteri \n",l);
}
```

Il programma in **linguaggio C**, riportato come esempio, calcola e visualizza la lunghezza di una parola introdotta da input. Per il calcolo della quantità di caratteri viene utilizzata una funzione (2) contenuta nel modulo `string` di cui si include l'interfaccia, per l'utilizzo delle funzioni contenute nella libreria, nella 1. In questo modo si aggiungono al C le funzioni per il trattamento di stringhe. Aggiungendo moduli diversi si specializza il linguaggio. Si potrebbero così avere moduli per la gestione della grafica, dei database e così via.

Il linguaggio C viene creato nel 1979 da Kernighan e Ritchie. Pensato per scrivere sistemi operativi e software di sistema, si tratta di un linguaggio ad alto livello e, tuttavia, comprende caratteristiche del linguaggio Assembly che lo fanno apprezzare per la sua efficienza e lo fanno definire come *linguaggio di medio livello*.

Un esempio di uso delle funzioni

Per una più agevolare esposizione dei principi del paradigma della programmazione strutturata si propone lo sviluppo di un programma in linguaggio C che calcola il totale di una fattura conoscendo, per ogni riga che la compone, la quantità di oggetti venduti e il prezzo unitario:

```
#include <stdio.h>

float CalcTot();

main()                                                                 /*1*/
{
  float totale;

  printf("Calcolo totale di una fattura conoscendone le righe\n");

  /* aggiornamento totale con valore calcolato dalle righe */

  totale = CalcTot();                                                  /*2*/
```

```
  /* Stampa totale */

  printf("Totale fattura %f\n",totale);
}

float CalcTot()                                                    /*3*/
{
  float t,pu;
  int i,qv;

  t = 0.0;

  printf("Righe fattura\n");
  for(i=1;;i++){
    printf("Riga n. %d\n",i);
    printf("Quant.venduta e prezzo unitario (0 0 per finire) ");
    scanf("%d %f",&qv,&pu);

    if((qv<=0)||(pu<=0)) break;

    t += qv*pu;
  };
  return t;
};
```

Un programma C è composto da più funzioni: nell'esempio proposto la funzione `main` e la funzione `CalcTot` definita in 3. La funzione `main` viene avviata quando viene lanciato il programma, la funzione `CalcTot` è invece richiamata dalla funzione `main` nella 2. Quando si sviluppa la funzione `main` si considera l'algoritmo risolutivo del problema presupponendo l'esistenza di macro-istruzioni che forniscono i risultati intermedi necessari: nell'esempio la sola `CalcTot` richiamata in 2. Successivamente si può sviluppare `CalcTot` occupandosi solo di questo aspetto del problema da risolvere. L'elaborazione è suddivisa nelle diverse funzionalità che ne fanno parte applicando un procedimento che consente di risolvere un problema per volta.

Gli obiettivi perseguiti dalla suddivisione del programma in funzioni sono sostanzialmente:

- **Riutilizzazione del codice**. Le elaborazioni scomposte nelle loro parti elementari (le funzioni) presentano blocchi di codice che possono essere utilizzati più volte con risparmio di energie da parte del programmatore e di costi per lo sviluppo. Per favorire la riutilizzazione la funzione deve essere quanto più possibile indipendente dal contesto.
- **Facilità di manutenzione**. Una funzione che esegue una singola elaborazione è più semplice da verificare di un intero programma in cui ogni singola parte può influire nei risultati ottenuti in un altra parte.

La programmazione ad oggetti

Con il diffondersi del paradigma della programmazione strutturata ci si rese conto che gli obiettivi assunti come riferimento, di generare moduli indipendenti dal contesto, erano stati raggiunti solo parzialmente.

La programmazione modulare pone l'attenzione sulla suddivisione delle *funzionalità* che devono essere implementate in un programma ma, come ricordava anche Wirth nel suo celebre libro, *Algoritmi + Strutture Dati = Programmi*, gli algoritmi riguardano elaborazioni su insiemi di dati

che, nella programmazione strutturata, *restano fuori*. Una funzione non può essere completamente esportata in un altro contesto perché fa riferimento a dati che, nel nuovo contesto, sono diversi.

> *La **programmazione orientata agli oggetti** (**OOP**, Object Oriented Programming) è un paradigma di programmazione, che prevede di raggruppare in un'unica entità (la classe) sia le strutture dati che le procedure che operano su di esse, creando per l'appunto un oggetto software dotato di proprietà (dati) e metodi (procedure) che operano sui dati dell'oggetto stesso.*
>
> *La modularizzazione di un programma viene realizzata progettando e realizzando il codice sotto forma di classi che interagiscono tra di loro. Un programma ideale, realizzato applicando i criteri dell'OOP, sarebbe completamente costituito da oggetti software (istanze di classi) che interagiscono gli uni con gli altri.* (Da Wikipedia.)

Il primo linguaggio che supportava completamente il nuovo paradigma, fu il **Simula** (1967), seguito dallo **Smalltalk**. Negli anni '80 furono create le estensioni ad oggetti per il linguaggio C e si diede vita al **linguaggio C++**. L'obiettivo che si pose Bjarne Stroustrup, il ricercatore danese a cui si deve l'implementazione e la definizione iniziale di C++, fu quello di non disperdere l'enorme specializzazione raggiunta dai programmatori nel linguaggio C (estremamente diffuso). Chi utilizza C++ può utilizzare la *vecchie* strutture sintattiche del linguaggio C e le *nuove* estensioni ad oggetti.

Per esporre l'utilizzo del paradigma OOP si riprende ora il problema presentato in precedenza del calcolo del totale di una fattura conoscendone le righe composte ognuna da quantità venduta e prezzo unitario.

Nel problema esposto si possono evidenziare due entità (***classi***) che interagiscono: la fattura e la riga. Per ognuna bisogna individuare i dati coinvolti e il tipo di operazioni richieste per quei dati:

```
class fattura{
public:
  fattura(): totale(0.0) {};                                    /*1*/
  void aggiorna(float t){totale +=t;};                          /*1*/
  friend ostream& operator<<(ostream&, const fattura&);         /*2*/
protected:
  float totale;                                                 /*3*/
};
ostream& operator<<(ostream& output, const fattura& f)          /*2*/
{
  output << "Totale fattura: " << f.totale << endl;
  return output;
};
```

la classe fattura, per quanto richiesto dal problema da risolvere, è provvista della proprietà `totale` (3) e dei metodi (1) per il trattamento del dato. Nel caso specifico è previsto un metodo per l'inizializzazione e un altro per l'aggiornamento. È possibile anche ridefinire l'operatore per l'output (<<) in modo da adattarsi alle istanze della classe (2).

```
class riga{
public:
  void nuova(int, float);                                       /*1*/
  float totriga();                                              /*1*/
  friend ostream& operator<<(ostream&, const riga&);            /*2*/
protected:
  int qv;                                                       /*3*/
```

```
  float pu;                                                    /*3*/
};

ostream& operator<<(ostream& output, const riga& r)            /*2*/
{
  output << "Quantita\' venduta: " << r.qv
         << " Prezzo unitario: " << r.pu << endl;
  return output;
};

// metodo per inserimento di una nuova riga

void riga::nuova(int q, float p)                               /*1*/
{
  qv = q;
  pu = p;
};

// metodo per il calcolo del totale della riga

float riga::totriga()                                          /*1*/
{
  float tr;
  tr = (float) qv*pu;
  return tr;
};
```

Per la classe riga sono definiti (3) gli attributi quantità venduta (`qv`) e prezzo unitario (`pu`) e i metodi (1) per la generazione di una nuova riga e per il calcolo del totale della riga. Anche per le istanze della classe è definito (2) l'operatore di output.

Tutto quello che riguarda la fattura o la riga della fattura è inserito nella rispettiva classe. La classe può essere utilizzata in un qualsiasi contesto che necessiti di una fattura o delle caratteristiche di una riga. La OOP è dotata di meccanismi che permettono di adattare, aggiungendo e specificando ma senza avere necessità di modifiche del codice esistente, la classe a nuove esigenze. Per esempio se si dovesse avere l'esigenza, in una diversa elaborazione, di calcolare lo sconto su ogni riga, il codice per lo sconto può essere aggiunto con un sistema del tipo: *oltre alle caratteristiche già esistenti aggiungi anche queste altre o modifica, in questo modo, quelle altre*.

Nel programma, che utilizza le classi, interagiscono istanze (***oggetti***) delle varie classi. Ogni oggetto in quanto appartenente alla classe ne gode di tutte le proprietà. I metodi rappresentano delle *competenze* di tutti gli oggetti della classe:

```
#include <iostream>
#include “c_fattura”                                           /*1*/
#include “c_riga”                                             /*1*/
using namespace std;

int main()
{
  int qvend,i;
  float pzunit;
  riga r;                                                      /*2*/
  fattura f;                                                   /*2*/

  // elaborazione righe fattura
```

```
  for(i=1;;i++){
    cout << "Riga n. " << i << endl;
    cout << "Quantita\'_venduta prezzo_unitario (0 0 per finire) ";
    cin >> qvend >> pzunit;

    if((qvend<=0) || (pzunit<=0))
      break;

    r.nuova(qvend,pzunit);                                    /*3*/
    cout << r;                                                /*4*/
    f.aggiorna(r.totriga());                                  /*3*/
  };

  // stampa fattura

  cout << f;                                                  /*4*/

  return 0;
}
```

Dopo aver incluso nel programma anche il codice delle due classi (1), si possono dichiarare (2) oggetti delle classi definite. I due oggetti `r` ed `f` hanno le caratteristiche definite per le rispettive classi: `r` è una riga come la si intende nella classe riga così come `f` è una fattura per come si intende nella classe. Con gli oggetti si interagisce *inviando messaggi* (3). L'oggetto risponde per come definito dal metodo: la generazione di una nuova riga e l'aggiornamento del totale della fattura con il totale della righe.

Il linguaggio SQL

In considerazione della vasta diffusione è obbligo fare riferimento, in questa sede, al linguaggio SQL, anche se, a dire il vero, non si tratta di un vero e proprio linguaggio di programmazione: non si possono scrivere programmi utilizzando solo SQL.

Il linguaggio SQL è un linguaggio di definizione e interrogazione di grandi insiemi di dati collegati fra loro (i **database**).

Fra gli anni '60 e '70, Edgar F. Codd, ricercatore presso i laboratori IBM, crea il modello relazionale per i database. In poche parole si tratta di un modello che vede i dati rappresentati in formato di tabelle e che permette di interagire con un database in modo più flessibile di quanto non permettevano i modelli elaborati precedentemente. Nel 1974 ad opera di Donald Chamberlin nasce SQL come strumento per comunicare con database relazionali. Inizialmente concepito come linguaggio da utilizzarsi in maniera *stand-alone*, da solo e da parte di un utente, per poter ottenere informazioni da un database, in seguito acquista capacità di essere inglobato (*embedded*) in un 3GL. In questo modo il linguaggio 3GL si espande ad acquisire strumenti per interagire con un database.

```
SELECT cognome, nome
  FROM clienti
  WHERE fatturato > 1000
  ORDER BY cognome, nome
```

il codice riportato chiede l'elenco (con cognome e nome) dei clienti con fatturato maggiore di 1000, ordinati alfabeticamente.

Il codice evidenzia una caratteristica importante di SQL che lo differenzia, per esempio, da un 3GL:

laddove in un qualsiasi linguaggio 3GL bisogna specificare ***come*** fare per ottenere il risultato atteso, in SQL si specifica ***cosa*** bisogna ottenere come risultato. Per questa caratteristica SQL viene definito come ***linguaggio non procedurale***. In alcuni casi viene definito anche come linguaggio 4GL.

Linguaggi di scripting

I programmi compilati, come già notato, sono veloci ma non portabili. In ambienti con piattaforme che variano spesso o non conosciute, è necessario utilizzare interpreti. Le applicazioni di rete hanno queste caratteristiche e, inoltre, un programma compilato non è, per motivi di sicurezza, ben accetto: il programma potrebbe avere dei bug o funzionalità nascoste (backdoor) che possono compromettere il funzionamento dei servizi forniti o interferire con gli utenti. Inoltre in un ambiente di rete non è possibile sfruttare la velocità di un programma compilato, la sua caratteristica fondamentale. In ambiente Client/Server il problema principale è la comunicazione fra due processi residenti in computer anche distanti fra di loro.

Nelle applicazioni per computer sono, si può dire, da sempre esistiti linguaggi interpretati utilizzati per effettuare in automatico determinate operazioni. Si tratta dei cosiddetti ***linguaggi di scripting***: un file di testo (lo script) contiene le istruzioni che saranno eseguite da un interprete. Dato il notevole sviluppo delle reti in generale, e di internet in particolare, i linguaggi di scripting si sono evoluti diventando molto potenti e paragonabili agli altri linguaggi di programmazione.

Nelle tecnologie web uno script può esistere in un file autonomo (*stand-alone*) o inglobato all'interno di una pagina web, nel codice HTML (*embedded*).

In ambiente di rete si distingue fra:

- Scripting lato client (**client-side**). Lo script è eseguito dall'interprete che, stavolta, è il browser che richiede la pagina HTML, la visualizza sul monitor dell'utente ed esegue, se presenti, le istruzioni contenute nello script. Il server interviene soltanto nell'invio della pagina richiesta.
- Scripting lato server (**server-side**). Lo script è passato dal web server all'interprete che lo esegue e che prepara la pagina web da inviare, da parte del web server, al client che ne ha fatto richiesta.

La distinzione fra elaborazione client-side e server-side negli ambienti di rete è dovuta alla esigenza di non gravare il server, fin quando possibile, con elaborazioni che possono essere svolte dal client evitando un sovraccarico della rete.

Scripting client-side: JavaScript

JavaScript originariamente sviluppato all'interno della Netscape Communications, è stato standardizzato fra il 1997 e il 1999. Il codice può esistere in un file esterno e, in questo caso, nella pagina HTML viene inserito, nel punto opportuno, il nome del file: il browser legge dal file le istruzioni e le esegue. Il codice può essere incluso, all'interno di appositi indicatori, in un punto qualsiasi della pagina HTML.

```
...
<script type="text/javascript">

  // Calcolo risultato operazione aritmetica

  function calcola(){
```

```
    var n1 = eval(document.f1.primo.value);
    var n2 = eval(document.f1.secondo.value);

    document.f1.risultato.value = n1 + n2;
  }
</script>
...
<!-- Acquisizione degli input dell'utente -->

<FORM name="f1">
  Primo numero
  <INPUT type="text" name="primo" size="5">
  Secondo numero
  <INPUT type="text" name="secondo" size="5">
  Risultato
  <INPUT type="text" name="risultato" size="5" readonly=”readonly”>
  <INPUT type="button" value="Esegui Somma" onClick="calcola()">
</FORM>
...
```

Lo script è incluso fra i marcatori `<script ...>` e `</script>` e può comparire in qualsiasi parte della pagina anche se, motivi non trattati in questi appunti, portano alla conclusione che è meglio inserire il codice nella parte di intestazione della pagina.

Il listato di esempio mette in evidenza alcune caratteristiche peculiari di JavaScript:

- La possibilità di intercettare le azioni dell'utente. Nel codice la funzione che calcola la somma è richiamata dal clic del mouse sul pulsante con etichetta `Esegui Somma`.
- La possibilità di accedere agli oggetti della pagina. La funzione `calcola()` legge gli input dell'utente inseriti nei campi `primo` e `secondo` della form HTML e scrive nel campo `risultato`.

Scripting server-side: PHP

Linguaggio di scripting, con licenza open source, originariamente concepito per sviluppare pagine web dinamiche, attualmente utilizzato per scrivere applicazioni web lato server, PHP può, oltre che essere inglobato in pagine HTML, essere utilizzato per scrivere applicazioni stand-alone. È nato nel 1994 ad opera del danese Rasmus Lerdof e ha avuto larga diffusione per la possibilità di connettersi e interfacciarsi, facilmente, con un gran numero di database.

Quando un client fa richiesta al web server di una pagina che contiene uno script PHP, il server interagisce con l'interprete PHP che esegue le istruzioni contenute nello script inviatogli.

```
<?php
  $connessione = mysql_connect("localhost","root","");
  mysql_select_db("azienda");

  // preparazione query

  $q  = "SELECT cognome,nome ";
  $q .= "FROM clienti ";
  $q .= "WHERE fatturato > " .$_POST[fat];
  $q .= “ ORDER BY cognome,nome”;

  // invio query e preparazione pagina con risultati

  $recset = mysql_query($q);
```

```
  while($tr = mysql_fetch_array($recset)){
    echo $tr['cognome'];
    echo $tr['nome'],"<br>";
  };
  mysql_close($connessione);
?>
```

Nel codice di esempio, PHP si interfaccia con il database MySQL (le funzioni utilizzate cominciano con `mysql_`).

La query in SQL viene composta tenendo conto anche degli input dell'utente inviati dal client. Tali input, uno nel caso dell'esempio, sono accessibili dal codice PHP per mezzo dell'array associativo `$_POST[]`. Il codice invia la query a MySQL (`mysql_query`), riceve i risultati, li trasforma in maniera opportuna (`mysql_fetch_array`) e li stampa su una pagina HTML, per esempio utilizzando l'istruzione `echo`. La pagina così costruita viene passata al web server che provvede ad inviarla al client che ne ha fatto richiesta.

La caratteristica di PHP di generare, al volo e secondo i dati ricevuti, le pagine web, ne fanno uno strumento molto diffuso per lo sviluppo di quelle che vengono comunemente chiamate ***pagine web dinamiche***.