LINGUAGGI

# DAL FORTRAN IV AL FORTRAN 77

SECONDA EDIZIONE

ROBERTO FARABONE
ROBERTO DORETTI

GRUPPO EDITORIALE
JACKSON

# DAL FORTRAN IV AL FORTRAN 77

Roberto Doretti - Roberto Farabone

GRUPPO
EDITORIALE
JACKSON
Via Rosellini, 12
20124 Milano

SECONDA EDIZIONE: 1987

GRAFICA E IMPAGINAZIONE: Francesca Di Fiore
COPERTINA: Emiliano Bernasconi
FOTOCOMPOSIZIONE: Lineacomp S.r.l. - Via Rosellini, 12 - 20124 Milano
STAMPA: Stabilimento Grafico Matarelli (MI)

*È indegno di uomini*
*dall'eccellente ingegno*
*lo sprecare ore nella*
*fatica del calcolare.*
*Leibnitz*

# SOMMARIO

**APPENDICI**

# PREFAZIONE

Circa trent'anni fa, uno studio condotto da autorevoli esperti dell'epoca indicava in qualche centinaio di unità il mercato mondiale complessivo degli elaboratori elettronici. Alla luce dei fatti, raramente una previsione si rivelò tanto errata. A discolpa degli autori, occorre però tener presente che cosa era allora un elaboratore elettronico: una macchina enorme, che dissipava grandi quantità di energia, che si guastava ogni momento, che richiedeva utenti di rara specializzazione.

Il costo, le prestazioni, il tipo di addetti, giustificavano l'impiego di tali macchine solo in un numero limitato di applicazioni, in accordo con la previsione di cui sopra.

Che cosa invece sia successo, è di dominio pubblico. L'elaboratore ha registrato un progresso spettacolare, che ne ha cambiato radicalmente la faccia e le prospettive. Da strumento di élite, riservato ad una ristretta casta di *sacerdoti* in camice bianco, è diventato lo strumento di lavoro giornaliero di milioni di persone, praticamente in ogni settore delle attività umane.

L'elaboratore è oggi addirittura strumento di una profonda trasformazione della società.

Esso è infatti un veicolo della transizione dalla attuale società industriale a quella post-industriale, qualificata come la società dell'informazione.

Nella *esplosione informatica*, un fattore determinante è certamente costituito dallo straordinario progresso delle tecnologie elettroniche. Basti pensare che la potenza di elaborazione di una delle mastodontiche apparecchiature di trent'anni fa, è oggi realizzata da un dispositivo integrato delle dimensioni di un francobollo.

In effetti, il progresso tecnologico è solo l'aspetto più appariscente della evoluzione complessiva dell'informatica. Questa evoluzione ha riguardato infatti non solo il modo di costruire l'elaboratore, ma anche la facilità di impiego. Sotto quest'ultimo profilo, di importanza fondamentale è stato il lavoro per superare o ridurre la *barriera del linguaggio*.

Per come è concepito, l'elaboratore possiede un linguaggio proprio che è assolutamente lontano dal linguaggio naturale dell'uomo. Un linguaggio di *uni* e *zeri*, che riflette la struttura interna della macchina. Eppure, anche se oggi sembra inconcepibile, questo fu il linguaggio in cui inizialmente si programmarono gli elaboratori.

Si può affermare che se fossimo rimasti a questo stadio, le previsioni di diffusione di cui si è parlato all'inizio sarebbero state sostanzialmente centrate.

Lo sviluppo dei linguaggi di programmazione costituisce da solo uno dei capitoli più ampi e interessanti dell'informatica. Dai linguaggi simbolici *uno a uno*, in cui l'uso di nomi e simboli mnemonici solleva il programmatore da alcuni degli aspetti più faticosi del lavoro, ai linguaggi evoluti, orientati ad ampie aree applicative o addirittura a specifici problemi, è stata fatta molta strada. Un cammino non sempre lineare, con molte varianti più o meno significative. Basti pensare che i linguaggi sviluppati, includendo i vari *dialetti*, si contano a centinaia.

In questa apparente babele dei linguaggi, emergono però pochi e precisi punti di riferimento. Uno di questi è, fuori di ogni dubbio, il FORTRAN. La grande maggioranza dei programmi per applicazioni scientifiche è stata scritta in questo linguaggio, e in tale ambito applicativo esso rimane tuttora il linguaggio più usato.

Data la sua popolarità, libri sul FORTRAN non mancano. Tuttavia questo nuovo libro va segnalato per una serie di caratteristiche che non è dato generalmente di trovare riunite: la chiarezza di esposizione e la gradualità con cui il lettore viene introdotto nell'argomento, la ricchezza di esempi e l'impostazione *top-down* seguita diffusamente nel testo, la completezza e l'aggiornamento del materiale. Dietro queste caratteristiche traspare l'ampia esperienza degli Autori, sia in campo didattico che nell'applicazione a problemi reali. Questo libro è perciò una utile guida per chiunque voglia iniziare o approfondire l'impiego di questo linguaggio.

Franco Filippazzi

# PREFAZIONE ALLA II EDIZIONE

Ai tempi del Liceo, un nostro professore di Disegno e Storia dell'arte continuava a ripetere che un disegno, come un quadro, come comunque un'opera di una qualche importanza, non è mai terminata, è sempre suscettibile di ritocchi, di aggiustamenti, di perfezionamento.

Rileggendo un libro come questo, sia pur a poco tempo dalla sua prima stesura, può accadere agli autori di riconoscere finalmente il valore di quei discorsi: è ovvio che qui non si tratta di opere d'arte e neppure di opere di chissà quale importanza, ma solo di un testo che pretende di insegnare qualcosa, eppure il principio è proprio lo stesso. Insomma mai nulla è finito e definitivo: tutto quanto va sempre rivisto, rimeditato, riadeguato ai tempi.

Dopo tutto, gli autori si sono accorti solo di questo. In effetti, affrontando il discorso da un altro punto di vista, è innegabile che la velocità dell'evoluzione informatica, ben nota a chi di essa si occupa, richieda aggiornamenti frequenti. Forse quello che più ci ha spinto ad una revisione del testo è proprio tale fatto: già in così pochi anni sono cambiati alcuni modi di lavorare e la maniera di affrontare certi problemi. Certo ha contato molto la volontà di eliminare le eventuali inesattezze inevitabilmente presenti, ma sopratutto il desiderio di impostare il discorso facendo riferimento a quei mezzi che la recente tecnologia ha messo a disposizione. Intendiamo il terminale ed il Personal Computer.

Forse, paragonando la vecchia e la nuova edizione, qualcuno si accorgerà che le aggiunte o le modifiche in proposito occupano ben poco spazio, tuttavia è il discorso nel suo complesso che assume un aspetto diverso. Tanto per fare un esempio: pochi anni fa si usavano essenzialmente le schede. Si trattava di qualcosa di concreto: scrivere una scheda significava comporre sull'apposità perforatrice una riga costituente l'"istruzione" (in senso lato) da passare al computer. Oggi la stessa cosa la facciamo battendo una riga che comparirà sul video di un terminale o di un Personal Computer. Poco male, dopotutto anche oggi chiamiamo "scheda" quella riga, però la trattiamo in maniera diversa, non la dobbiamo maneggiare, conservare in un armadio od altre cose di questo genere. Soprattutto, comunque, è il video stesso (sia esso quello del terminale o del PC) che ci ha portato ad un modo diverso di lavorare. Forse è semplicistico dire che è grazie ad esso che oggi possiamo scrivere programmi interattivi, ma sta di fatto che il colloquio uomo-macchina è diventato più facile e vantaggioso proprio contemporaneamente al suo diffondersi.

Con queste brevi note non abbiamo voluto, come si vede, fare una elencazione delle differenze che il lettore potrà trovare tra la vecchia e la nuova edizione, ma solo accennare ai motivi che ci hanno spinto ad aggiornare il testo. Siamo certi che il FORTRAN continuerà ancora a diffondersi come uno dei più vantaggiosi linguaggi tecnico-scientifici ed a divenire sempre più potente, quindi di revisioni ne saranno necessarie altre, forse molte. È certo che, nonostante la nascita di linguaggi magari più potenti ed eleganti, esso è rimasto sino ad oggi il più noto ed utilizzato tra chi deve servirsi degli strumenti informatici nella tecnologia e nella scienza; anzi propio recentemente dopo l'ondata commemorativa seguita al venticinquennale della nascita del linguaggio, esso sta vivendo una seconda giovinezza, di cui non è assolutamente possibile non tenere conto.

Anzi, anche per questo, va segnalata l'uscita, presso questa stessa Casa Editrice, di un Eserciziario "Esercizi di Fortran" degli stessi autori, che arricchisce e completa l'opera iniziata con questo libro. Esso contiene i sunti dei più importanti argomenti, una vasta massa di esercizi svolti, nonchè un buon numero di proposti ed ha il preciso scopo di facilitare l'apprendimento pratico del linguaggio.

Detto ciò, non possiamo che lasciare al lettore interessato l'utilizzo critico del presente volume.

Gli Autori

# INTRODUZIONE

**Dall'Abaco all'Informatica**

Appena l'uomo ebbe imparato a far di conto, volle costruirsi qualcosa che lo aiutasse. La cosa risale agli albori della storia e la prima delle "macchine" aiutanti fu l'abaco; ci vollero però dei secoli perché nascessero le prime *macchine calcolatrici* degne di tal nome. Forse la prima fu quella costruita da Pascal nel XVII secolo: essa però, in sostanza, non fu che un abaco un poco più complesso, nel quale alle varie palline erano state sostituite delle ruote dentate. Poi fu la volta di Leibnitz che ne estese l'uso e dopo ancora, di molti altri matematici: ognuno aggiungeva un granello al progresso dell'automazione nella fiduciosa ricerca di uno strumento vantaggioso e con la dichiarata intenzione di risparmiare fatica. Ci volle ancora del tempo, poi, all'inizio del nostro secolo, *macchine calcolatrici* maggiormente perfezionate entrarono nell'uso comune: ad esse era (ed è, giacchè ne esistono ancora) possibile far eseguire l'operazione aritmetica voluta, semplicemente pigiando dei tasti appropriati.

Ma non si era ancora giunti al momento della *rivoluzione* e della nascita dei cosiddetti *cervelli elettronici*. Forse fu la costruzione della prima bomba atomica americana a dare una spinta decisiva in quel senso: nella fase di progetto tutti i calcoli (e furono tanti) vennero eseguiti da squadre di giovani donne che, meccanicamente, li svolgevano secondo le richieste, utilizzando calcolatrici del tipo citato in precedenza. Perchè allora non costruire una macchina che eseguisse altrettanto meccanicamente le stesse azioni?

Certo non è bello pensare che lo stimolo all'evoluzione della scienza e della tecnica sia dovuto alle esigenze di distruzione di una guerra incombente od in atto.
Purtroppo, però, nel nostro caso fu certamente così: si pensò che una macchina capace di eseguire automaticamente calcoli complessi e ripetitivi avrebbe potuto ad esempio compilare facilmente le tavole di tiro. Le ragioni militari prevalsero e vennero trovati gli stanziamenti necessari: nasceva così l'ormai leggendario ENIAC, il primo vero *elaboratore*. Si era nel 1946. I militari ci rimisero. La guerra era terminata, dell'ENIAC si disinteressarono e l'elaboratore venne usato a scopi matematici e nella tecnica di pace.

Si era aperta una nuova strada: effettivamente ora non si doveva faticare troppo a calcolare. D'altro canto ora anche le informazioni di tipo non numerico potevano essere trattate, archiviate in poco spazio, rilette nel momento voluto e via dicendo: un'intera enciclopedia poteva venire raccolta in un angolo dell'elaboratore sempre a disposizione per ogni richiesta.

Intanto l'evoluzione dell'elettronica conduceva a progressi insperati: gli elaboratori divenivano meno imponenti e, soprattutto, più veloci. L'ingombrante ENIAC veniva messo in meritato riposo in un museo e la sua più potente progenie si diffondeva per il mondo con una velocità di espansione mai vista.

L'elaboratore perdeva il fantascientifico e scorretto, anche se affascinante, nome di *cervello elettronico* per diventare uno strumento di tutti i giorni nell'industria, nelle università, nelle banche e, perché no, anche nelle famiglie.

Contemporaneamente nasceva la scienza dell'informazione: l'*informatica*. Veniva studiato come migliorare il trattamento delle informazioni, come far funzionare l'elaboratore nella maniera più appropriata, come ottenerne i vantaggi maggiori.

L'informatica è oggi ben più di quanto presupponga la frase appena scritta. Non vogliamo, in questa sede, né darne una definizione, né tanto meno affrontarne in dettaglio le caratteristiche. Vogliamo però sottolineare che la complessità cui il mondo e la società sono giunti rende questo strumento irrinunciabile. L'elaboratore è di importanza vitale sia nella produzione, sia nella ricerca, sia nella gestione del tessuto sociale. Quanto tempo ci vorrebbe ad un ingegnere per eseguire i calcoli del progetto di un ponte solo con carta e matita? Quanto ci impiegherebbe un fisico a calcolare, sempre con carta e matita, la velocità e le energie delle sue particelle elementari e quanto tempo servirebbe ad un impiegato per andare a ritrovare il certificato di nascita di un cittadino di New York o Roma?

La risposta è comunque ovvia: troppo.

Il progresso ha, ormai, una sua velocità. Non restare al passo significa condurre il genere umano ad una rapida involuzione: non essere in grado di gestire i grandi numeri cui l'espansione demografica ci ha abituato, significa aprire la porta alla confusione ed alla degenerazione di ogni rapporto sociale.

L'informatica, insomma, a così pochi anni dalla sua nascita, non è già più soltanto un comodo strumento, ma è una scienza di cui non dobbiamo fare a meno.

## Usare l'elaboratore

Cosa significa in termini più precisi, usare l'elaboratore?

Prima di tutto sappiamo che si può fare eseguire ad una macchina quella parte noiosa e ripetitiva di calcoli e controlli che, in altri tempi, saremmo stati costretti ad eseguire manualmente noi stessi. Inoltre, in generale, lo si può

fare molto più velocemente. Essa è molte volte più rapida di noi nel compiere le operazioni e meno soggetta a commettere errori: non si trova certo coinvolta in problemi emozionali, di ripetitività, di noia, di stanchezza o disattenzione.

Va chiarito subito che con una macchina per procedimenti elaborativi automatizzati, o *computer* come lo denomineremo d'ora innanzi, non si può fare altro che quello che, in linea di principio, si può eseguire a mano o con l'aiuto di una normale calcolatrice contabile; solo che, trovarsi costretti ad eseguire manualmente calcoli complessi od a trattare una grossa mole di dati, significa inevitabilmente costringersi a dover superare grosse difficoltà: il lavoro risulta sempre molto oneroso, seppur fattibile, ed i risultati non possono essere molto affidabili, a causa di errori comunque assai difficilmente evitabili. Solo con una macchina *intelligente* (il nostro computer per l'appunto) sarà possibile elaborare con esattezza, affidabilità ed in modo sufficientemente economico, calcoli complessi, o sarà possibile trattare dati molto numerosi.

Abbiamo usato il termine macchina *intelligente* in un senso ben preciso, e vediamo di eliminare subito il rischio di possibili fraintendimenti: la macchina non è *mai* intelligente, ma abbiamo usato ugualmente il termine volendo con esso riferirci ad un dispositivo in grado di eseguire operazioni logiche od aritmetiche nel modo che noi gli chiediamo di fare e, tutto questo, con un solo intervento da parte di un operatore e non, come nel caso di una calcolatrice, intervenendo operazione per operazione.

Le informazioni su tutto quanto si vuol eseguire devono essere fornite alla macchina in modo opportuno: all'inizio dell'era dei computer, tali informazioni dovevano essere scritte ed organizzate unicamente tenendo conto del fatto che la macchina ha delle proprie modalità di lavoro, e quindi utilizza una certa *logica*.
Così *programmare* la macchina (fornirle cioè le informazioni necessarie per ottenere la soluzione automatizzata di un problema) era un lavoro arduo ed eseguibile da parte di pochi specialisti: essi scrivevano i loro programmi nel cosiddetto *linguaggio macchina*, cioè in un linguaggio che la macchina non aveva bisogno di tradurre per sapere che cosa eseguire: essa leggeva direttamente i comandi e poi ad uno ad uno li eseguiva.

Anche oggi è possibile procedere in questo modo, ma è diventato inutilmente faticoso, dopo la nascita dei *linguaggi simbolici*. I linguaggi simbolici non evoluti (in generale i cosiddetti linguaggi assemblatori) sono linguaggi che esprimono in maniera simbolica, con parole, quegli stessi comandi che in linguaggio macchina sarebbero scritti attraverso combinazioni numeriche direttamente comprensibili dal computer: della traduzione da simboli, o parole, ad istruzioni-macchina, si occuperà il computer stesso. L'evoluzione ha poi prodotto altre possibilità: ora possiamo comunicare al computer i nostri comandi in un linguaggio simbolico ancor più generalizzato, con frasi

molto vicine a quelle del nostro linguaggio comune, assegnando ancora alla macchina stessa il compito di eseguire la traduzione in simboli più semplici per lei fino ad arrivare al suo linguaggio macchina.

Questo processo di traduzione prende il nome di *compilazione* e vedremo che esso è un passo importantissimo nell'utilizzo dell'elaboratore. Per ora teniamo presente che l'insieme di istruzioni comunicato in questo modo alla macchina costituisce un programma codificato in un *linguaggio evoluto*, intendendo con tale termine un linguaggio assai vicino al nostro e (per noi) di facile utilizzo, composto di frasi semplici e concise.

Poter programmare in questo modo non è cosa da poco quando si pensi che, così facendo, la nostra attenzione può essere rivolta tutta alla logica del problema e non a quella della macchina e del suo funzionamento. In altre parole ciò di cui si deve preoccupare chi stabilisce le operazioni da far eseguire, cioè il *programmatore*, è di trovare esclusivamente la soluzione logica del problema che gli si prospetta, e quindi, grazie a metodi di facile apprendimento, codificare tale soluzione nel linguaggio evoluto più indicato.

Ma vediamo, con un semplice esempio, quanto in pratica significhi ciò che abbiamo esposto finora.

Consideriamo un insieme di 30 numero decimali: 127.823, 4612.009, ...., 87.003616 ed ammettiamo di dover moltiplicare ognuno di essi per 7. Pur utilizzando una calcolatrice, ci possiamo ben rendere conto che, a seconda della velocità ad operare ed a digitare correttamente i numeri, potrà venir impiegato un tempo più o meno lungo (mai brevissimo) e che il numero di errori commessi (e quindi di correzioni, ammesso che di essi ci si accorga) non potrà risultare trascurabile. Un qualsiasi computer compie tutte queste operazioni in poche frazioni di secondo. Ovviamente, per poter eseguire queste funzioni, occorrerà istruirlo in qualche modo.

Per l'esempio in esame, la cosa è abbastanza semplice: le operazioni da eseguire consistono nel considerare un numero, moltiplicarlo per 7, rendere in qualche modo disponibile il risultato, eseguire nuovamente queste operazioni per il numero successivo e così via, sino ad esaurire tutti i numeri da trattare.

Come si può osservare, la soluzione logica prescinde dal modo di funzionare della macchina: il nostro sforzo è (e così deve essere) indirizzato solo a comprendere la natura del problema ed a cercarne la soluzione.

Va inoltre fatta un'altra osservazione riguardo al riconoscere un'importante struttura del problema (e ciò dimostrerà ancora una volta quanto sia utile una elaborazione automatizzata): è possibile ideare la nostra soluzione come un insieme di cicli di operazioni ripetitive.

Abbiamo già detto che il linguaggio che utilizzeremo per programmare sarà un linguaggio evoluto, un linguaggio che non si allontani troppo dai nostri schemi mentali, che sia dunque per noi facilmente apprendibile; ma abbiamo anche detto che esso deve essere un linguaggio adatto al problema che dobbiamo trattare.

Cosa si intendeva con questo?

Esistono fondamentalmente due tipi diversi di problemi di cui si occupa l'Informatica: quelli di carattere gestionale e quelli di carattere scientifico, dei quali ultimi intendiamo preferibilmente occuparci in questo libro.

I primi riguardano le informazioni di carattere generale, il raggruppare, ordinare dati, l'ottenere sunti e riorganizzazioni di informazioni qualunque relative alla gestione delle più generiche situazioni immaginabili: tenere un conto delle merci e dei valori di un magazzino, sapere quando rifornire il magazzino stesso, oppure tenere, in una banca, il conteggio degli assegni, oppure ancora, in una ditta, tenere aggiornata la situazione del personale, e così via...

Il problema scientifico, invece, riguarderà più da vicino il calcolo su certi dati di partenza, l'elaborazione di metodologie matematiche applicate alla Fisica, all'Ingegneria, alla Tecnologia, la ricerca di soluzioni a problemi di tipo logico, le applicazioni di metodologie statistiche eccetera.

Si potrà sicuramente obiettare che il confine tra i due tipi di problemi non è poi così netto.

Ciò è senz'altro vero, tuttavia i problemi di tipo scientifico hanno una caratteristica che li contraddistingue in modo particolare e che deve risultare decisiva quando ci si domandi quale sia l'ambito mentale nel quale sia preferibile muoversi e quindi anche quale linguaggio di programmazione risulti più adatto.

Si tratta della caratteristica di algoritmicità: diciamo che un algoritmo è uno strumento che permette di trasformare certi dati di partenza, solitamente numerici, per mezzo di formule matematiche (o logiche) in altri dati numerici, in risultati ed informazioni nuove rispetto a quanto si disponeva in partenza.

Possiamo già anticipare a questo punto che il linguaggio evoluto forse più adatto, o comunque più diffuso, per la soluzione di questo tipo di problemi è appunto il FORTRAN, l'oggetto di studio di questo libro.

Prima però di prendere in esame il linguaggio, vanno spese ancora alcune parole su come si debba procedere nell'affrontare i problemi del tipo suddetto, giacchè è proprio tale punto di partenza a determinare in modo drastico la

distinzione tra il *bravo programmatore*, anzi vorremmo dire *risolutore*, ed il mediocre.

A volte l'algoritmo che dobbiamo usare per la soluzione del nostro problema è semplice, già pronto sotto forma di una banale formuletta che attende solo di essere applicata, ma più spesso esso va ricercato, va appositamente *inventato*.

Nei problemi di tipo gestionale solitamente esiste una sostanziale differenziazione tra la figura dell'*analista*, o ricercatore ed inventore del metodo, e la figura del programmatore; non succede altrettanto in campo scientifico dove spesso la possibilità di risolvere un problema è strettamente legata alla possibilità di trovare un algoritmo automaticamente elaborabile.

Per questo ci sentiamo in dovere di esaminare, anche se quanto più rapidamente possibile, come vada affrontata questa serie di problemi, e di fare una raccomandazione preliminare: il problema di tipo scientifico deve essere analizzato con accuratezza, ne vanno studiate le varie caratteristiche logiche e si deve giungere ad una corretta formulazione dell'algoritmo prima di iniziare ad effettuare la vera e propria programmazione.

Condizione prima sarà una perfetta conoscenza dei termini del problema che ci accingiamo ad affrontare, immediatamente dopo sarà necessario chiarire in modo inequivocabile le richieste, ossia le domande che ci siamo poste e per le quali il problema è nato, gli obiettivi che intendiamo raggiungere.

A questo punto o la conoscenza di metodologie già studiate e note o uno sforzo immaginativo (... buona fortuna, in questo caso!) ci condurrà all'algoritmo risolutivo.

Con la soluzione siamo certamente a buon punto, ma c'è ancora parecchio da fare: l'algoritmo va ora verificato in ogni sua parte ed in ogni possibilità, va controllata la sua fattibilità e va infine ridotto in termini simbolici semplici, cioè a dire, in formule o in sequenze di formule; e saranno proprio queste a costituire il nucleo del nostro programma e solo a questo punto potremo finalmente partire col vero e proprio lavoro di programmazione.

# PARTE I
# CONCETTI GENERALI

CAPITOLO 1

# IL PROGRAMMA E L'ELABORATORE

## 1.1 La Programmazione

Si è già detto che un programma è l'insieme delle istruzioni che vengono comunicate alla macchina perché essa esegua certe operazioni e che il programmare altro non è che l'attività di stesura di quelle istruzioni.

Prima di vedere come si procede a quest'ultima operazione, ci sembra opportuno avere le idee chiare su cosa si debba dire alla macchina perché essa svolga un certo lavoro e questo perché ad essa non dovranno essere comunicate soltanto le istruzioni riguardanti il nostro problema (ovvero il programma), ma anche qualcosa di più riguardante, ad esempio, la richiesta di compilazione ed esecuzione di quel programma. Queste istruzioni, non strettamente inerenti al problema che ci siamo proposti di risolvere con l'aiuto dell'elaboratore, fanno parte di quello che viene normalmente chiamato *linguaggio di controllo* (e su questo torneremo più avanti): il programma vero e proprio sarà costituito da quelle, e sole, istruzioni che risolvono il problema e ad esse solamente dovremo prestare attenzione nel lavoro di programmazione.

Questa precisazione, che a qualcuno potrà parere scontata, è però servita a ribadire un concetto, una linea di condotta che abbiamo scelto e che riteniamo fondamentale: il programmatore deve badare alla struttura logica del problema, alla sua organizzazione in termini chiari e precisi, prima di iniziare a pensare come fare poi eseguire alla macchina il suo compito.

Partendo da questo punto di vista il *programmare* diviene soprattutto *organizzare* in termini schematici secondo una ben precisa linea logica di sviluppo, prevista con la massima chiarezza e, possibilmente, con la massima semplicità.

Possiamo allora considerare il lavoro da farsi distinto in due fasi:

a) costruzione dello schema logico delle operazioni da eseguire (operazioni in senso generale e non solo aritmetiche), ovviamente ciò sarà accompagnato da un disegno dello schema che metterà in evidenza sia le operazioni da eseguire, sia soprattutto la linea logica di sviluppo: esso verrà chiamato *diagramma a blocchi* (o *flow-chart*)[1]; almeno fino a questo punto non dovrà essere necessario pensare a quale sarà il linguaggio di programmazione che verrà utilizzato;

b) traduzione del diagramma a blocchi in sequenza di istruzioni scritte nel linguaggio appropriato ed utilizzabile da parte dell'elaboratore; questa parte verrà chiamata *codifica* e condurrà alla stesura definitiva del programma.

Ci sentiamo di affermare che la prima di queste due fasi è la più importante (tanto che molto spesso si parla di programmazione riguardo solamente alla costruzione del diagramma a blocchi). Si era già notato che la macchina è capace di svolgere solo quelle operazioni che ognuno di noi potrebbe fare anche manualmente: essa non inventa nulla di suo. Sarà lo schematizzare in modo preciso e chiaro l'organizzazione del problema a permettere quella visione sintetica (linea generale di sviluppo) ed allo stesso tempo analitica (elencazione passo per passo delle cose da fare) che ci condurrà alla codifica del programma da passare alla macchina.

Vediamo ora di analizzare come si deva procedere.

Lo faremo tenendo sempre ben presente che si tratterà di chiarire, di puntualizzare e riassumere quello che normalmente faremmo anche senza l'aiuto dell'elaboratore (a mano, come si diceva prima).

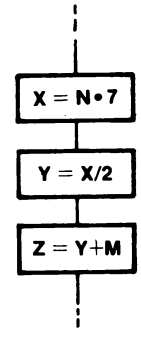
Ora, quando ci poniamo di fronte ad un problema e cerchiamo di risolverlo, quali sono le possibilità che abbiamo, o meglio, quali sono le cose che possiamo fare?

Sicuramente dovremo dedicare una certa attenzione alla *sequenza* delle operazioni da eseguire: per organizzarle in uno schema ordinato e facilmente leggibile utilizzeremo dei *blocchi* (da cui il nome di diagramma a blocchi), cioè dei simboli di varia forma contenenti l'indicazione dell'operazione stessa.

---

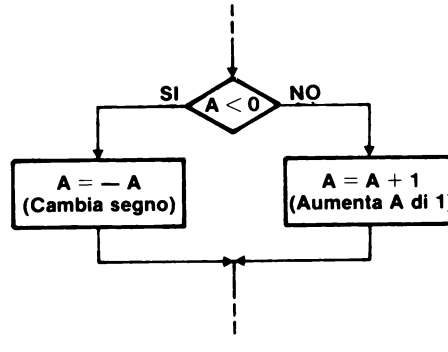
(1) Una trattazione approfondita dell'uso della diagrammazione a blocchi è presente nel volume "Logica e diagrammi a blocchi" di R. VIANO e R. FARABONE, edito dal Gruppo Editoriale Jackson.

Così, ad esempio, il problema di voler moltiplicare per 7 un certo numero N, dividerne il risultato per 2 ed infine sommare quanto ottenuto ad un numero M, potrà essere disegnato così:




dove, dunque, ogni operazione viene indicata all'interno di un rettangolo; tale struttura verrà detta **sequenza**.

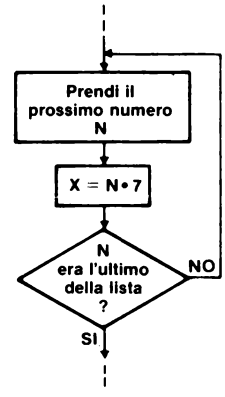
Durante la sequenza si possono presentare delle situazioni nelle quali le operazioni da eseguire sono alternative e richiedono che venga presa una decisione circa il fare una cosa piuttosto che un'altra e ciò in base ad un *test* (ovvero un'interrogazione, il controllo di una condizione, ecc.). Il test verrà simboleggiato da un rombo con due uscite, ad esempio:




il test è rappresentato dalla domanda: "Il numero A è negativo?", se la risposta è sì, dovremo cambiare di segno il numero, altrimenti il suo valore andrà aumentato di 1. Una struttura di questo tipo prende il nome di **alternativa**.

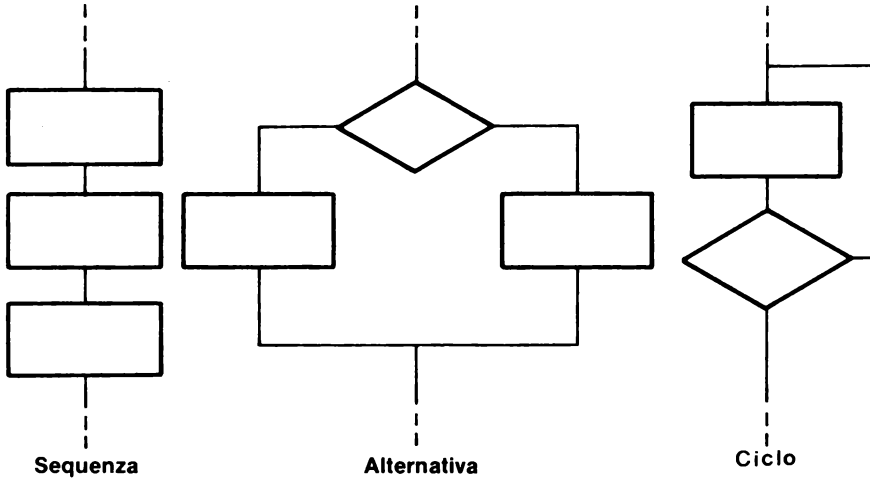
A questo punto va notato che accade spesso di dover ripetere un'operazione o una serie di operazioni: nell'analizzare il problema ci accorgiamo che una stessa cosa va fatta più volte e che dobbiamo decidere quante volte (o fino a

quando) continuare a rieseguire. Ad esempio nel caso dei 30 numeri decimali da moltiplicare per 7, potremo disegnare un diagramma come il seguente:




tale struttura viene detta di **ciclo** e, come si vede, in essa è presente (e lo sarà sempre in casi come questo) un test riguardante la fine della ripetizione. E' importante porre bene attenzione alla presenza, ma anche alla correttezza di tale test di fine ciclo, se si vuole evitare di incorrere in uno degli errori più tipici: quello di far eseguire al nostro programma delle operazioni indefinitamente, senza fermarsi più: la macchina, obbediente, tenterà di proseguire all'infinito il suo lavoro, entrando in quello che viene chiamato *loop* (1).

Negli esempi che abbiamo fin'ora analizzato, abbiamo messo in evidenza tre strutture, che chiamiamo di base: la *sequenza*, l'*alternativa*, il *ciclo*; esse sono dunque schematizzabili secondo i seguenti diagrammi:

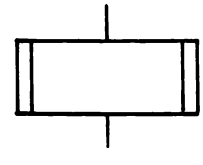



(1) L'elaboratore, in ogni caso, dispone di sistemi di controllo che interrompono l'esecuzione di un *loop* quando viene superato un certo intervallo di tempo.

La loro grande importanza sta nel fatto che esse servono e bastano a costruire il diagramma di un qualunque problema, comunque complesso esso possa essere, e, basta riflettervi un attimo, soddisfano a tutte le richieste logiche che un problema può presentare. Questo fatto è stato addirittura dimostrato: esiste infatti il teorema di Jacopini-Böhm (1), il quale afferma che ogni diagramma a blocchi di un qualsiasi problema, può essere riportato ad un diagramma costruito servendosi delle tre sole strutture di base. Anche se tutto ciò è fondamentale per il modo di procedere che intendiamo adottare, non spenderemo altre parole al proposito per non appesantire troppo il discorso con dimostrazioni che, almeno da un punto di vista strettamente applicativo, non possono risultare molto proficue; crediamo che, comunque, bastino i pochi esempi che seguono a farci facilmente rendere conto della verità del teorema e soprattutto della sua utilità, almeno per quanto riguarda la semplicità di ragionamento e l'economia logica.

Abbiamo già detto che, organizzando un problema, è bene badare alla linea logica che intendiamo seguire: gli obiettivi vengono sempre perseguiti tenendo una certa linea di condotta e ciò è assai importante nella costruzione del nostro diagramma a blocchi. Inizialmente è necessaria una visione del problema di tipo sintetico, si dovrà badare a quello che c'è da fare decidendo una *linea di flusso* lungo la quale sviluppare poi le azioni particolari da intraprendere. E' questo un punto che, ci teniamo a sottolinearlo, determina in maniera decisiva la facilità nel programmare: non dobbiamo mai pensare subito ai particolari, ma, una volta che si sappia con esattezza *cosa* si debba fare (conoscenza del problema, algoritmo e soluzione), procederemo per gradi, stabilendo prima per sommi capi l'organizzazione e la linea di flusso. Ciò è ottenibile in modo assai semplice e per capirlo è forse meglio aiutarci con un esempio.

Prima però un'avvertenza: useremo un nuovo simbolo, il rettangolo con due sbarre laterali (vedi figura), per indicare non più una semplice e generica operazione, ma un insieme di queste, un gruppo di istruzioni che possiamo vedere come un sottoprogramma. Ognuno di questi gruppi di operazioni verrà da noi battezzato con un nome mnemonico, magari inventato al momento, con il semplice ed unico scopo di riconoscerlo nell'uso che dovremo farne successivamente. Il gruppo di istruzioni potrà venire sviluppato compiutamente a parte in un secondo momento e, qualora lo si volesse, reinserito nel primo diagramma.



(1) Il teorema fu enunciato nel 1966 in un articolo comparso su "Communications of ACM".

Insomma in esso non dovremo preoccuparci di dettagliare alcunché: dovremo solo *indicare* ciò che vogliamo sia fatto. Dunque questo primo diagramma avrà la sola funzione di evidenziare quanto più possibile la linea di flusso ed al tempo stesso presenterà la grossa facilitazione di poter essere disegnato in maniera semplice e diretta: esso prenderà il nome di **diagramma di massima.**

Vediamo come primo esempio un caso estremamente semplice, ma basato su una logica *interattiva.* Prima di esporre il problema, due parole di spiegazione su questo termine: "diremo interattivi tutti quei programmi durante la cui esecuzione avviene un colloquio tra elaboratore ed utente"; ciò è come dire che taluni organi di input o di output sono strumenti adatti all'interazione uomo-macchina (come i *terminali* - vedi il relativo paragrafo) e che attraverso questi avviene un colloquio che fa sì che il programma porga delle domande ed attenda le relative risposte (è l'input dell'utente); queste a loro volta possono servire a guidare l'esecuzione determinando quali azioni intraprendere o, ad esempio, quali particolari calcoli eseguire.

Ammettiamo ora di voler scrivere un piccolo programma di conversione di valuta: lanciandolo esso dovrà dirci a quante lire corrisponda la cifra che gli diamo come input in dollari, o, viceversa, se gli diamo in input una cifra in lire, a quanti dollari corrisponda. Vogliamo anche che esso non termini subito dopo averci dato la risposta, ma possa proseguire e farci quante conversioni vogliamo; dovremo poter decidere la fine a nostra discrezione, ad esempio dandogli come input qualcosa di convenzionale (nulla di più comodo dello zero).

Le formule che ci servono sono ovviamente due e, posto che il cambio al momento che ci interessa fosse Lire 1927.25 = 1 Dollaro, molto semplicemente si dovrà calcolare, se X sono dollari:

$$X \cdot 1927.25$$

per ottenere le Lire; oppure, se X sono Lire:

$$\frac{X}{1927.25}$$

per ottenere i dollari.

Quello che è più interessante, però, è la sequenza delle operazioni che dovremo prevedere:

a) il programma dovrà chiederci quale delle due formule usare;

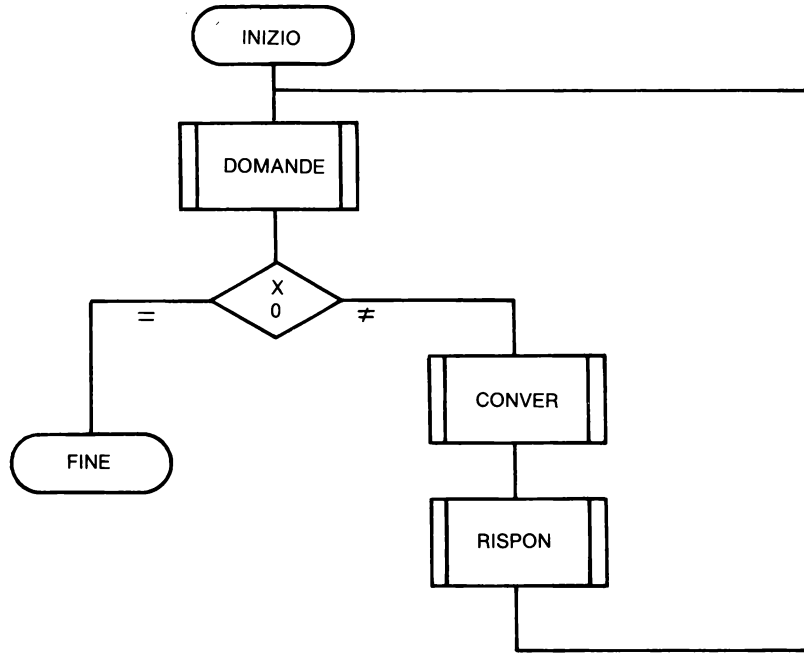
b) dovrà chiederci la cifra da convertire;

c) dovrà darci la risposta, ovvero la cifra convertita secondo quanto deciso in a);

d) dovrà ripetere quanto sopra quante volte vogliamo.

A ben pensarci le azioni indicate sono già di per sé il flow di massima; c'è però un punto un po' delicato: come gli diamo l'ordine di terminare? Abbiamo già notato che lo 0 è utile allo scopo: non sarà certo mai da convertire e dunque potremo utilizzarlo come segnale di fine.

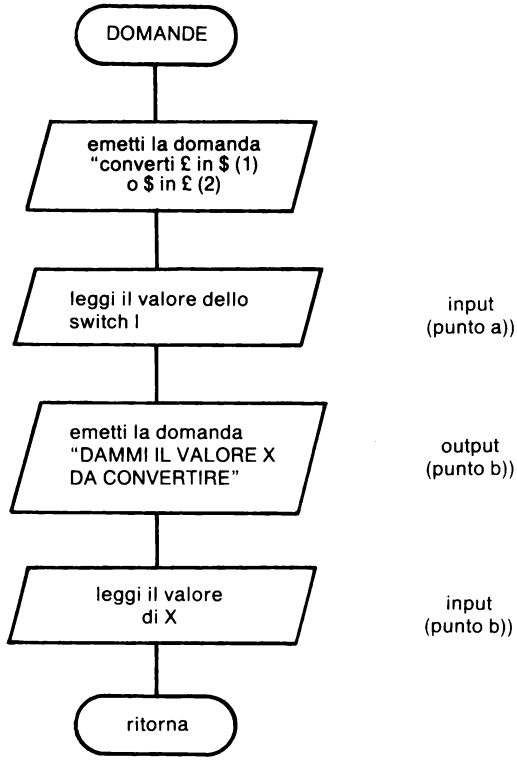
Allora il flow di massima potremo disegnarlo così:




Come si vede non ci siamo affatto preoccupati di analizzare come verranno fatte le domande, ma solo che verranno fatte: è il contenuto di DOMANDE e corrisponde ai punti a) e b) di cui sopra. Altrettanto dicasi per CONVER e RISPON in cui non ci siamo preoccupati di vedere come avviene la conversione e come vengono fornite le risposte (punto c)), ma solo del fatto che tali cose vengono elaborate in quel punto, o in quei momenti. Poi penseremo ai dettagli.

E' interessante notare subito, invece, come abbiamo realizzato la condizione di fine e la ripetizione (punto d)): se X è la cifra da convertire, quando ad essa assegnamo lo 0, il programma termina.

Passiamo finalmente ai dettagli: al blocco DOMANDE potremmo sostituire il seguente sviluppo:



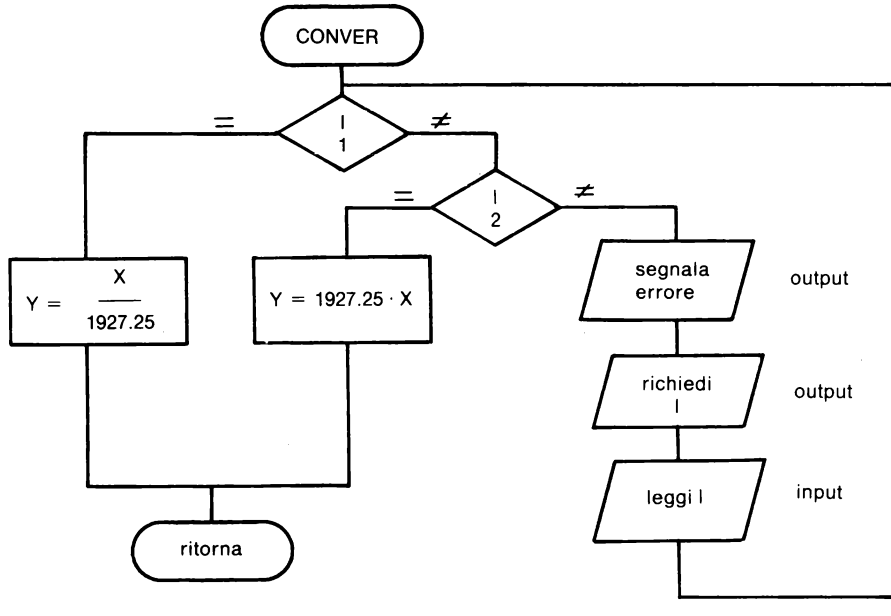

Come si vede, abbiamo distinto nettamente la fase di emissione delle domande (output) dalla lettura delle risposte che vengono fornite dall'utente (input). Non badiamo al formato delle domande stesse, notiamo invece che con I abbiamo fatto uso di ciò che si chiama **switch** (spesso anche **flag**): esso non è altro che un valore al quale noi assegnamo senso alternativo, che viene memorizzato in un punto e poi utilizzato più avanti, esso serve per *decidere* il da farsi più oltre, per imporre e memorizzare una condizione che solo in seguito dovrà essere tenuta presente. Va detto che nelle metodologie di programmazione più recenti si tende a non fare uso degli switch ed il perché è presto detto: se si può farne a meno si risparmia memoria e inoltre, e soprattutto, senza switch la lettura del programma è più semplice: ricordarsi, ad un certo punto, che valore gli avevamo dato può non essere cosa agevole.

Ci si può chiedere: si può farne a meno? La teoria vuole che la risposta sia "SI'" e la pratica non ha controesempi: di fatto qualunque programma può essere scritto senza l'uso di switch ed il consiglio è sicuramente di farne a meno.

Tuttavia, a volte, essi permettono di fornire soluzioni più semplici: volerne fare a meno può costringere ad involuzioni più o meno complicate, a sconvolgere la logica del programma facendoci perdere proprio quella linearità che ci premeva mantenere soprattutto e per la quale magari abbiamo voluto proprio evitare lo switch.
Quindi, attenzione a non usarli se se ne può fare a meno, ma non mettiamoci in grosse grane per evitarli!

Proprio per conoscerlo, dunque, abbiamo voluto introdurre subito lo switch in questo primo esempio: la prima domanda vuole che si risponda 1 per convertire Lire in Dollari, 2 per il viceversa; tale valore verrà memorizzato in I, successivamente lo "testeremo" per sapere quale formula utilizzare. Il lettore, per verificare quanto detto in generale sugli switch, provi al termine di questo esempio, a ridisegnare il flow senza farne uso (è possibile e facile scambiando di posto, ad esempio, le domande e le relative risposte).

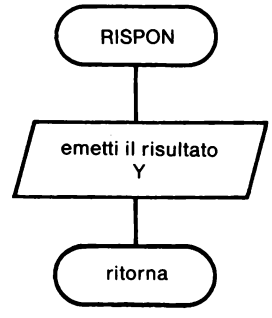
Passiamo ora a dettagliare CONVER: non c'è altro da fare che calcolare la conversione con la formula voluta:



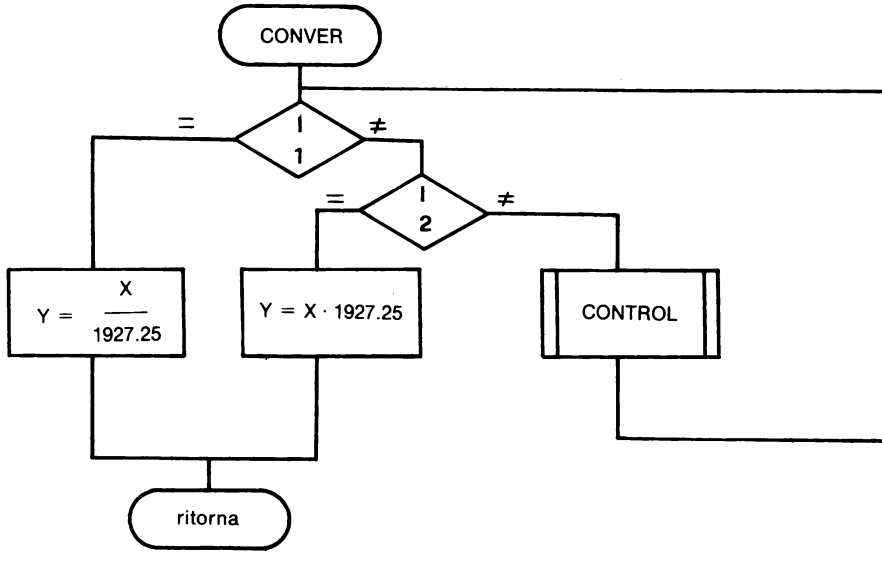

Abbiamo però voluto aggiungere qualcosa di più: un controllo che ad I non sia stato dato per sbaglio, in fase di input dell'utente, un valore diverso sia da 1 che da 2. Come si vede il controllo è realizzato assai semplicemente ed in

caso di errore si procede a rileggere I. Controlli di questo tipo sono indispensabili: non si dimentichi mai di inserirli.

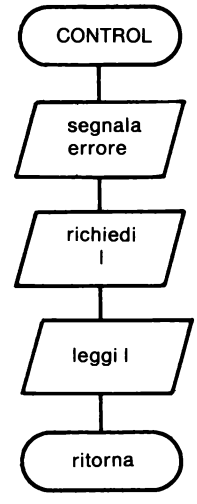
Infine RISPON è il caso più semplice, perchè è composto dal solo output del risultato:




Si noti che siamo passati in un solo colpo dal flow di massima a quelli di maggior dettaglio; nulla impedisce che i passaggi siano più d'uno; anzi in casi meno semplici di questo iniziale, ciò avviene sempre. Ma anche in questo esempio, almeno in un'occasione, si sarebbero potuti avere due passaggi: nel caso dei controlli di CONVER:



per poi sviluppare CONTROL.




Di sicuro il problema era assai semplice e non c'è da dubitare che molti intravvedano in questi successivi passaggi un inutile appesantimento. L'obiezione più ovvia è: "Non si sarebbe potuto fare tutto subito, in un unico disegno?" Certo, per ora la risposta è SI', ma ancora dobbiamo ribadire che prendendo l'abitudine a procedere secondo questo metodo, anche i problemi più complessi divengono di soluzione agevole e comunque appaiono al confronto assai più semplici.

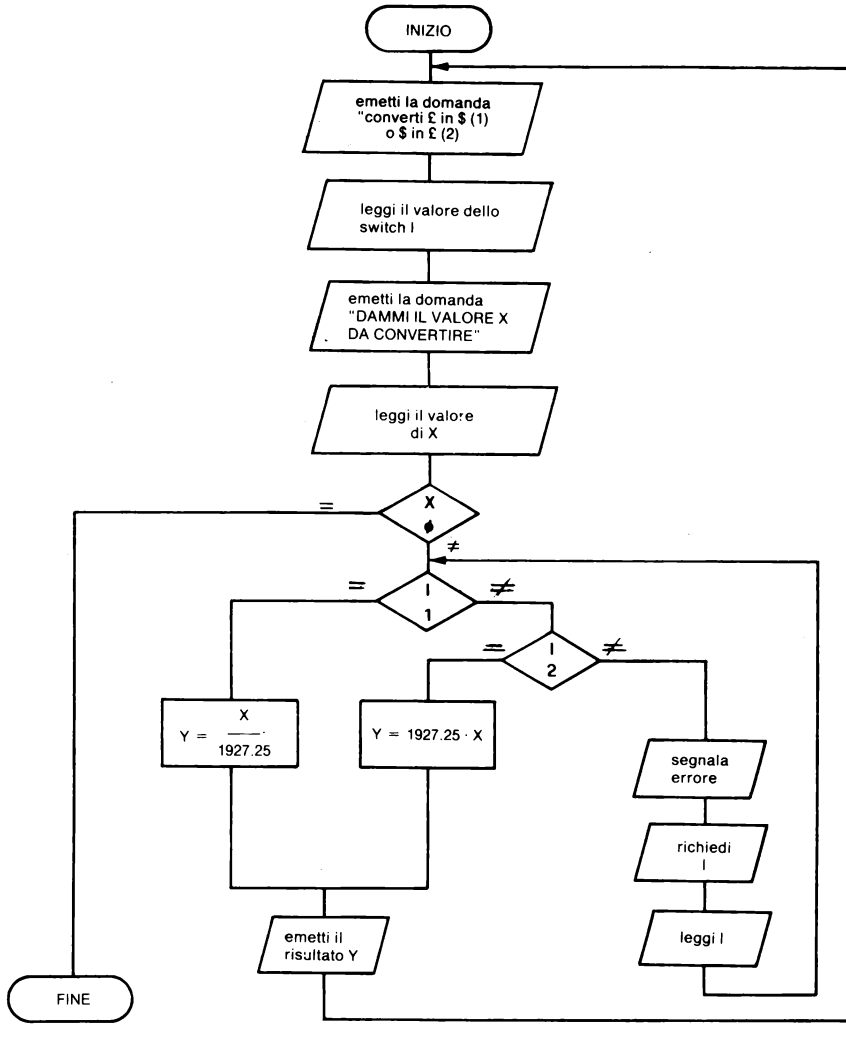
Nessuno nega che in casi più facili, il disegno completo, dettagliato al massimo livello, mostri una ineguagliabile leggibilità della linea logica. Anzi, vediamolo nel nostro esempio: prima di procedere il lettore riesegua tale disegno complessivo; esso non potrà che essere come quello di pag. 24.

Ma si pensi ai casi più complessi in cui il flow dovrebbe, così disegnato, occupare anche solo un paio di fogli di queste dimensioni: non sarebbe assai facile perdersi? Figuriamoci nella stragrande normalità dei casi, in cui cioè il flow di massimo dettaglio assume dimensioni che sono dalle 5 alle 10 volte quelle dell'esempio.

Passiamo ora ad un altro caso, di natura solo di poco più complessa in quanto riguarda anche l'uso di vettori. Esso è di carattere matematico ed è svolto, questa volta, in modo non interattivo.

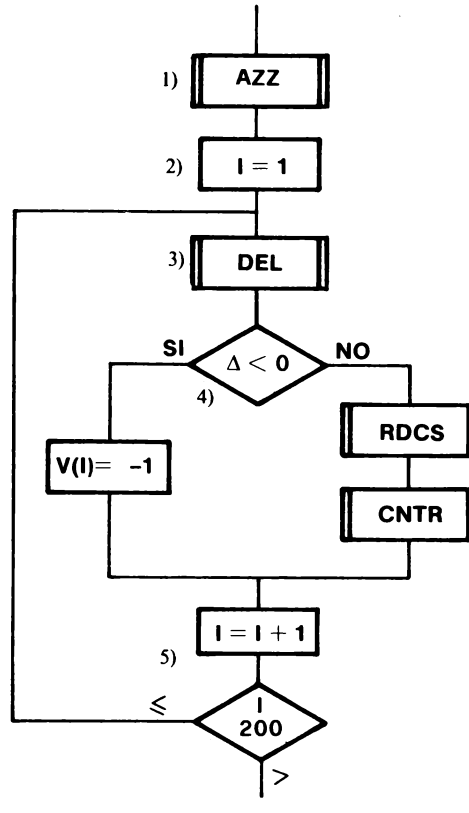
Nella memoria dell'elaboratore sono presenti tre vettori (1) A(I), B(I), C(I) le cui componenti di uguale ordine sono i coefficienti di un'equazione di

(1) Vedi Appendice E.




secondo grado, le equazioni sono 200 e dunque 200 è il numero delle componenti dei 3 vettori. Per ogni equazione dobbiamo calcolare le due radici, se sono reali e in tal caso memorizzarle in 2 vettori X1 e X2 (sempre di 200 elementi) inoltre dovremo determinare per quali equazioni la semisomma delle radici cade nel range [-1,1]. Useremo un vettore V(I) nel quale, dopo averlo azzerato, porremo a 1 la componente corrispondente ad un'equazione la cui semisomma delle radici cade nel range richiesto, a 0 se al di fuori di tale range, ed infine a -1 se l'equazione non ammette radici reali.

Si procede nel disegno come è indicato nella figura che segue:




1) La prima preoccupazione deve essere quella di azzerare tutto il vettore V(I): battezziamo queste inizializzazioni con un nome (AZZ) senza per ora preoccuparci delle singole operazioni;

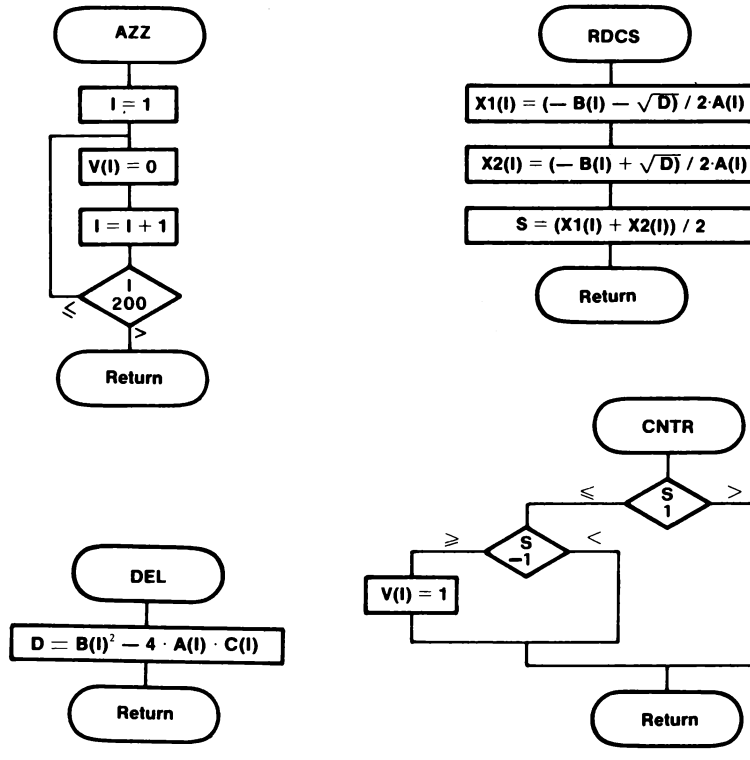
2) Il calcolo del discriminante e delle radici va ripetuto 200 volte (per ogni componente della terna di vettori per ottenere il ciclo useremo un indice I che inizialmente deve trovarsi a 1;

3) Prima di tutto va calcolato il discriminante (lo faremo in DEL sulle I-esime componenti dei vettori);

4) Bisogna controllare se il discriminante è positivo o meno: nel primo caso si dovrà procedere al calcolo delle radici (tale calcolo verrà successivamente sviluppato in RDCS) e vedere se esse cadono nel range richiesto (verrà fatto in CNTR e, sempre qui, verrà fatta l'assegnazione della componente di V(I); nel secondo caso dovrà semplicemente essere posta a -1 la componente di V(I);

5) Che si sia seguita l'una o l'altra strada, ora si deve incrementare l'indice di ciclo per passare alle prossime componenti; ovviamente ciò dovrà essere accompagnato da un controllo per vedere se si è terminato di scorrere i vettori.

A questo punto possiamo sviluppare ogni sottoprogramma (questa volta non aggiungiamo i commenti, poiché ci pare che il problema sia, tutto sommato, abbastanza semplice). Useremo un simbolo nuovo, l'ellissoide, per indicare inizio e fine di ogni programma o sottoprogramma: inizio e fine che non sono stati indicati nel diagramma di massima, poiché si sono volute sottintendere alcune istruzioni di relazione elaboratore-esterno, cui accenneremo tra poco. I vari sottoprogrammi si presenteranno nel seguente modo:




In DEL è presente una unica istruzione e quindi, ci si dirà, simboleggiare nel diagramma di massima addirittura un sottoprogramma per così poco è assurdo. In questo caso ciò è verissimo, perchè avremmo potuto facilmente prevedere che DEL sarebbe stato costituito da un'unica formuletta; ma, attenzione, non avviene sempre così: spesso domandarsi cosa c'è da fare in quel sottoprogramma è proprio quanto va evitato, allo scopo preciso di non perdere di vista la logica del problema ed è allora preferibile accorgersi solo dopo che lì dentro c'è un'unica istruzione: basterà, se proprio non possiamo farne a meno, rifare il disegno.

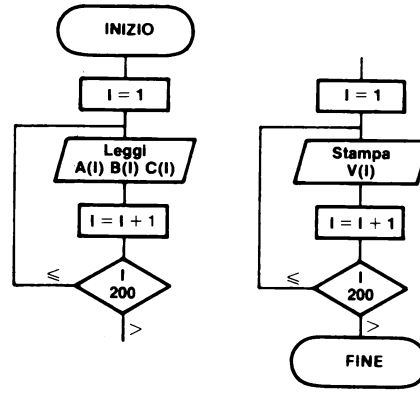
Ancora una precisazione: in CNTR non si pone V(I) a 0 quando la semisomma è fuori dal range, per la semplice ragione che V(I) è già stato tutto azzerato; ciò evidenzia un fatto importante: quando si scende nei particolari è

utile tenere ben presente tutto ciò che è stato fatto in precedenza e controllarlo pedissequamente. Anche se ciò può parere noioso, è però sicuramente indispensabile nei problemi più complessi.

L'esempio trattato è certamente abbastanza semplice, lo è almeno quel tanto che basta perchè il diagramma possa venir disegnato subito e tutto in una volta senza far ricorso ai sottoprogrammi. Tuttavia ciò che ci interessa è esemplificare un metodo che sicuramente faciliti la programmazione e che lasci poche possibilità di commettere errori di tipo logico. Un buon esercizio, a questo punto, potrebbe essere quello di ridisegnare tutto il diagramma di massima, sostituendo ai sottoprogrammi le istruzioni relative: ci si accorgerà allora, nonostante tutto, con quanta maggior (relativa) complessità si abbia a che fare; immaginiamo poi in un problema già di per sé complesso...

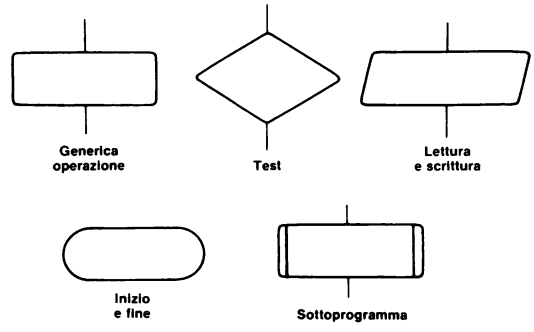
Abbiamo dato per scontato nell'esempio, che si sapesse cosa vuol dire memoria di un elaboratore. Ne parleremo un po' più diffusamente nel prossimo paragrafo, per ora continuiamo a ritenerla semplicemente un magazzino nel quale le informazioni possono venire messe o dal quale possono essere prelevate. Verrà detto in generale **input** l'operazione di trasferire in memoria delle informazioni dall'esterno; l'operazione di trasferimento inverso verrà detta **output**. Più in particolare parleremo spesso di **lettura** (l'elaboratore legge i dati, le informazioni di cui necessita) e di **scrittura** (i risultati vengono scritti, stampati).

Nell'esempio trattato, sicuramente i vettori A, B e C devono venir letti ed il vettore V stampato; nel diagramma a blocchi tali operazioni vengono indicate con un parallelogramma, così, supponendo di dover leggere terna per terna A(I), B(I) e C(I) e stampare V(I), potremo disegnare:




rispettivamente all'inizio e alla fine di tutto il diagramma a blocchi.

Possiamo riassumere ora tutti i simboli utili alla costruzione di un diagramma a blocchi:




Prima di proseguire, una raccomandazione: la vera difficoltà del programmare sta negli errori che si commettono (e non commetterne affatto è praticamente impossibile), quindi è necessario che il diagramma a blocchi sia chiaro e facilmente leggibile; esso dovrà essere controllato più volte e provato seguendo passo per passo, con dei dati scelti ad hoc (e di facile controllo), tutte le sue possibili vie. Ciò va fatto prima di iniziare la codifica: quando si dovesse scoprire un errore (di tipo logico) e ci si trovasse costretti a cercarne l'origine nel programma già codificato, si presenterebbero difficoltà notevoli, qualche volta insormontabili.

A questo punto può essere assai importante analizzare i seguenti due esempi, per prendere un minimo di confidenza con la tecnica di programmazione per mezzo dei diagrammi a blocchi.

## 1.2 Esempi

*Esempio 1*

Su di una scheda sono scritti i coefficienti di un sistema lineare in due incognite del tipo:

$$\begin{cases} ax + by = c \\ a'x + b'y = c' \end{cases}$$

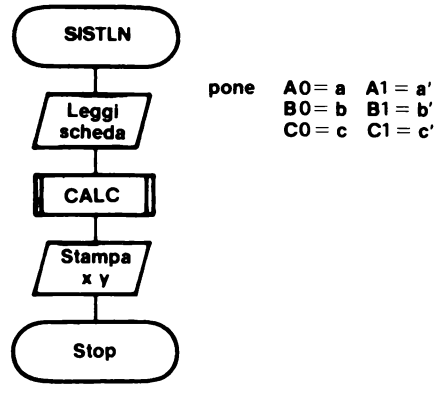
Proponiamoci di trovarne le radici con il metodo dei determinanti di CRAMER:

$$x = \frac{\begin{vmatrix} c & b \\ c' & b' \end{vmatrix}}{\begin{vmatrix} a & b \\ a' & b' \end{vmatrix}} = \frac{cb' - c'b}{ab' - a'b} \qquad y = \frac{\begin{vmatrix} a & c \\ a' & c' \end{vmatrix}}{\begin{vmatrix} a & b \\ a' & b' \end{vmatrix}} = \frac{ac' - a'c}{ab' - a'b}$$

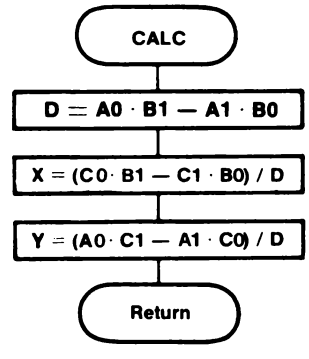
Poniamo che sulla scheda siano scritti ordinatamente i coefficienti:

a,b,c,a',b',c'.

Potremo procedere iniziando con un diagramma di massima del tipo (1):




in CALC verrà eseguito il calcolo delle radici:




Ovviamente, essendo tale problema di grande semplicità sarebbe stato possibile inserire tutto CALC direttamente nel primo diagramma ottenendone uno unico.

Come si osserverà, il calcolo dei determinanti viene effettuato più volte eseguendo sempre lo stesso tipo di operazione: se si potessero passare i dati ad

(1) Le notazioni che utilizzeremo nei diagrammi per le varie grandezze in gioco saranno lettere maiuscole o nomi (come si farà per la codifica FORTRAN), per le operazioni useremo invece gli usuali simboli algebrici.

un unico sottoprogramma il quale provvedesse al calcolo in modo abbastanza generalizzato, cioè prendendo come valori di partenza del calcolo stesso volta per volta le grandezze necessarie, il lavoro ne risulterebbe molto semplificato. Tutto ciò, se applicato a problemi di maggiore complessità, diventa certamente un metodo di procedere notevolmente vantaggioso.

A tale proposito, modifichiamo così il problema che ci siamo posti: si debbano risolvere 100 sistemi del tipo visto, i cui coefficienti si trovino ordinatamente scritti, come prima, in serie di sei ogni scheda:

a,b,c,a',b',c'.

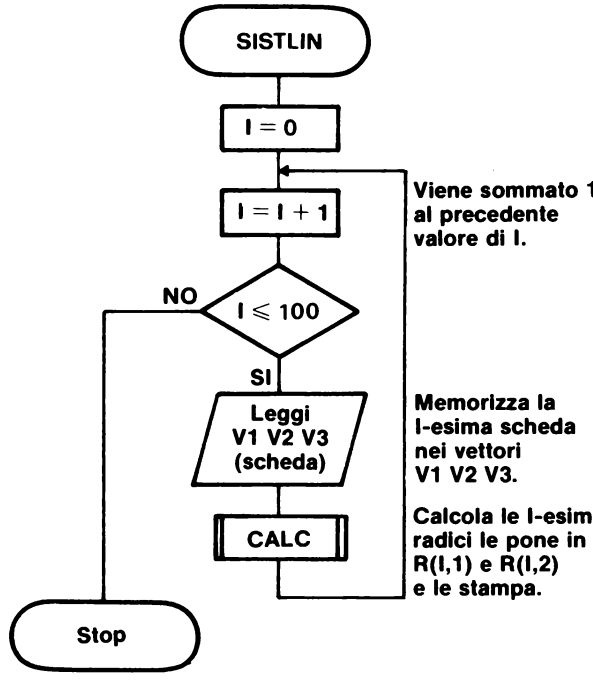
I sei valori vengono inizialmente memorizzati nei tre vettori:

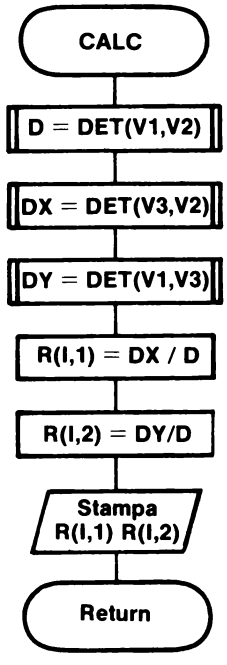
**V1** = [a,a']
**V2** = [b,b']
**V3** = [c,c']

per ognuno dei sistemi.

Intendiamo raccogliere le 100 coppie di radici x e y in una matrice R(100,2) emettendone subito ogni riga in stampa.

Si potrà procedere così:

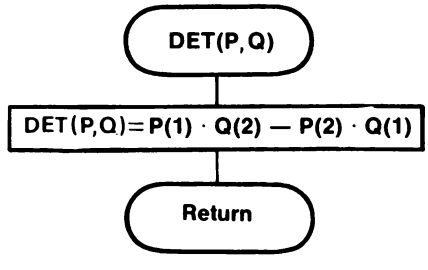





**Calcola il determinante su V1 e V2 e mette il risultato in D.**

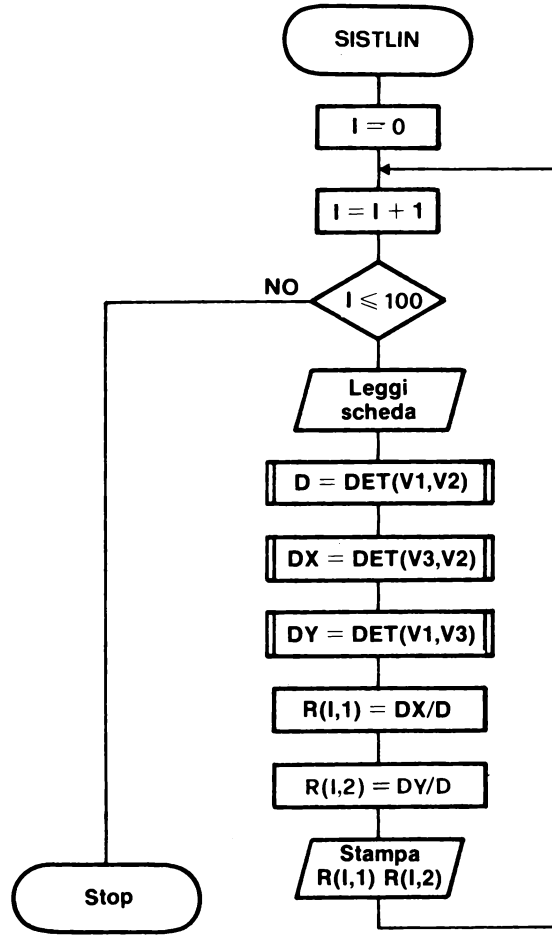
**Calcola il determinante su V3 e V2 e mette il risultato in DX.**

**Calcola il determinante su V1 e V3 e mette il risultato in DY.**




**In questo sottoprogramma P e Q saranno rispettivamente V1, V2 la prima volta che viene chiamato, V3, V2 la seconda e V1, V3 la terza.**

Come alternativa alla stesura vista, potremmo anche inserire i vari sottoprogrammi DET nel diagramma di massima al posto di CALC, ottenendo:




Anche questa volta il problema era abbastanza semplice per poter disegnare un unico diagramma con inseriti i vari sottoprogrammi. Tuttavia il procedere

in questo modo ci ha permesso di mettere in evidenza due fatti importanti (di uno almeno dei quali si è già diffusamento detto): prima di tutto procedere per diagramma di massima e sottoprogrammi è più facile, semplifica sia lo sforzo inventivo durante l'atto del disegno, sia un'eventuale rilettura dello stesso; in secondo luogo un medesimo sottoprogramma può venire scritto una sola volta e poi essere richiamato in quante occasioni si desideri (e facendolo lavorare su valori anche diversi) ottenendo un certo risparmio non solo sul tempo di codifica, ma anche sulla ampiezza del programma.

*Esempio 2*

Su un flusso di schede è scritto un campione statistico di 1000 numeri x, un numero per scheda da memorizzare in un vettore X(1000); si vuole calcolarne la media XM:

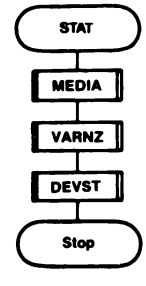
$$x_m = \bar{x} = \frac{\sum_{i=1}^{1000} x_i}{1000}$$

la varianza v e la deviazione standard σ:

$$v = \frac{\sum_{i=1}^{1000} (x_i - x)^2}{999} \qquad \sigma = \sqrt{v}$$

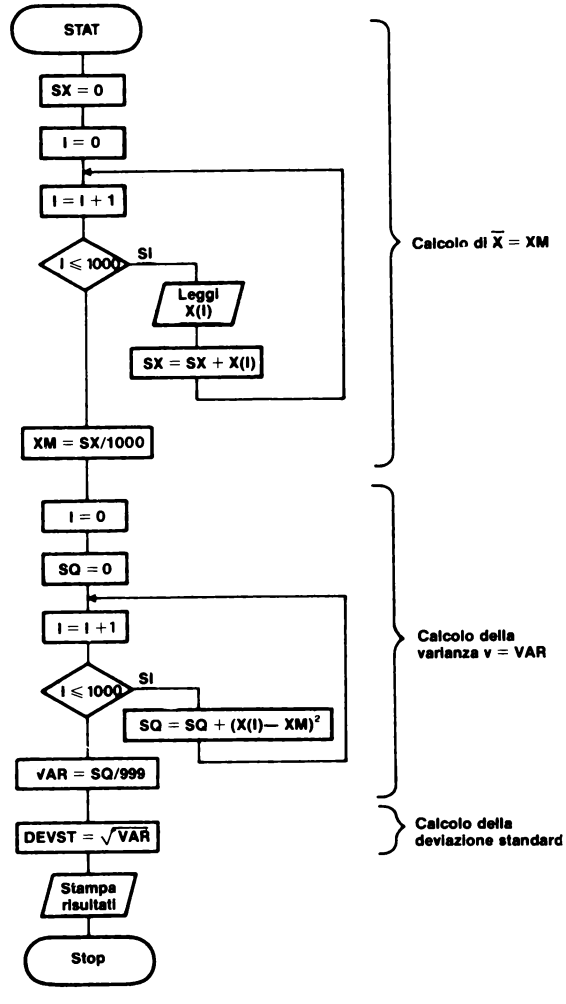
stampando infine i risultati.

Possiamo costruire un diagramma di massima molto semplicemente come segue:



Questa volta, esso serve solamente ad evidenziare le parti in cui è logicamente suddivisa la procedura: in realtà poiché le varie operazioni vanno eseguite per così dire *in cascata*, cioè secondo una normale sequenza, e con dei cicli del tutto indipendenti l'uno dall'altro e poiché, infine, non esiste la possibilità di ripetere più volte la stessa operazione (o serie di operazioni), possiamo tranquillamente disegnare un diagramma completo tutto in una volta. Tuttavia, va aggiunto che, anche in casi come questo, qualora si presenti il minimo dubbio, sarà sempre preferibile fissare le idee per mezzo di un semplice diagramma di massima, piuttosto che rischiare di confondere i termini del problema procurandosi inutili perdite di tempo.

Si tratterebbe, comunque, di disegnare un diagramma come il seguente:



## 1.3 La struttura di un elaboratore

Un elaboratore è una macchina assai complessa la cui descrizione richiederebbe da sola un intero trattato. Non ci proponiamo, ovviamente, nulla di simile; sarà comunque indispensabile avere una idea generale, abbastanza precisa, della macchina che intendiamo utilizzare, quale ne sia la natura, quali gli organi principali, quale il modo di funzionare.

Si suole suddividere ciò che concerne l'elaboratore in **hardware** e **software**: con il termine hardware si intende tutto ciò che, nella macchina, è strettamente fisico come le parti elettroniche, le parti circuitali, quelle meccaniche e così via; con il termine software si intende invece la parte programmata, le istruzioni da interpretare ed eseguire e, in definitiva, tutti quei modi di funzionare predisposti dall'uomo.

Del software faranno parte, dunque, anche i nostri programmi (*programmi utente*), ma va subito detto che il termine è ben più ampio: ad un elaboratore, affinchè agisca secondo i nostri desideri, non bastano le istruzioni che gli comunichiamo ed i circuiti per eseguirle. Ogni istruzione prima di essere portata a compimento deve venire interpretata e tradotta; un dato numerico, un'informazione qualunque che intendiamo trattare devono, allo stesso modo, seguire una serie di passaggi e interpretazioni, prima di venir attivamente utilizzati; a tutto ciò provvede una serie di programmi, o **routine**, predisposti una volta per tutte e ai quali l'elaboratore ha accesso libero ed immediato.

In altre parole, ogni elaboratore ha in sé un software predisposto che costituisce una parte importantissima del funzionamento della macchina nel suo complesso.

A questo punto, anzi, va aggiunto che esiste un gruppo di programmi adatti (costituente, per così dire, il nucleo centrale del software dell'elaboratore) che ha il compito di determinare il funzionamento dell'elaboratore stesso in ogni sua parte, di gestire tutti gli altri programmi, compresi quelli *utente*, di distribuire le varie risorse di cui la macchina dispone,di controllare quanto avviene colloquiando con l'utente per mezzo di quel linguaggio di controllo di cui si è già dato un cenno.

L'insieme di programmi svolgenti tale funzione viene chiamato **sistema operativo** e determina sovente la potenza del sistema.

Abbiamo or ora utilizzato un termine nuovo: quello di **sistema**. Con esso si intende sia l'hardware che il software di una macchina: si intende, insomma, l'elaboratore nella totalità della sua costituzione fisica e capacità esecutiva.

La descrizione che all'inizio ci siamo proposti, di un elaboratore, o meglio, come diremo d'ora in avanti, di un sistema *tipo*, inizierà dall'hardware e più precisamente dall'*architettura*, cioè dalla composizione dei vari moduli fisici.

Tali moduli verranno spesso denominati **unità**, dove con tale termine si intenderanno parti specificatamente ideate per svolgere funzioni particolari.

Tra queste è bene individuarne immediatamente tre: quella di memorizzazione, quella di svolgimento delle varie operazioni o attività e quella di passaggio delle informazioni dall'interno all'esterno del sistema e viceversa: la **memoria** va pensata come un vero e proprio magazzino delle informazioni (dati e programmi vi vengono ritenuti per poter essere attivamente trattati); l'esecuzione delle attività è devoluta ad un modulo che prende il nome di **unità centrale** che ha il duplice compito di effettuare sia un controllo generale delle varie operazioni da svolgere che l'esecuzione vera e propria delle stesse; infine lo scambio di informazioni con l'esterno avviene per mezzo di un'**unità di input-output** (input = ingresso delle informazioni, output = uscita) che ha, al pari della memoria, un carattere essenzialmente passivo essendo, per così dire, al servizio dell'unità centrale.

Possiamo, allora, eseguire una prima schematizzazione del nostro sistema tipo:


Essa va necessariamente puntualizzata: lo faremo prendendo in esame ogni singola unità.

## 1.4 Memoria

Si è soliti descrivere la memoria tramite l'esempio di quella a nuclei: in memoria sono contenuti numerossimi piccoli nuclei magnetizzabili in un senso od in quello opposto per mezzo del passaggio di corrente elettrica. In questo modo sono state costruite memorie potenti e sicure; oggi tuttavia si tende a sostituire il nucleo con circuiti o microcircuiti elettronici (più veloci e meno ingombranti). Non vogliamo qui affrontare l'argomento da un punto di vista tecnologico: quello che ci interessa è la funzione cui il nucleo od il circuito sono devoluti: ognuno di essi viene posto in uno stato A o B, acceso o spento, aperto o chiuso ovvero, come d'ora in avanti diremo, 1 o 0. In altre parole ognuno di essi può memorizzare uno stato binario che ci permette, attraverso opportune codifiche, di scrivere o leggere informazioni e dati. Ogni singolo nucleo o circuito viene detto **bit** (1).

---

(1) Dall'inglese BInary digiT.

Immaginiamo allora la nostra memoria composta da una serie di bit assai estesa: se desideriamo memorizzare il numero 6 sarà sufficiente tradurlo in binario 110 e porre tre bit negli stati corrispondenti; se desideriamo memorizzare il carattere A sarà sufficiente stabilire un codice il quale faccia corrispondere, ad esempio, la configurazione 001000001 alla nostra lettera e porre negli stati corrispondenti il numero necessario di bit... Possiamo procedere allora, come si intuisce, a memorizzare informazioni complesse, numeri più grandi, caratteri di ogni tipo, parole (1).

Quando una di tali informazioni deve venire prelevata, il sistema accede in memoria ai bit corrispondenti per mezzo dell'indirizzo della loro posizione. Tale indirizzo è conosciuto dalla macchina per via *hardware*, ma è specificato via *software*, cioè da come tale informazione è definita, ovvero battezzata con un nome, con un simbolo nel programma. Il sistema provvede automaticamente a collegare ad ogni nome l'indirizzo appropriato. Tale indirizzo fa sempre riferimento ad un gruppo di bit. L'unità minima di indirizzamento dipende dal tipo di sistema e da come esso è stato progettato: spesso tale unità è un gruppo di bit che viene denominato **parola**. Essa è costituita normalmente da 36 o 32 bit. Altrettanto spesso essa può essere vista suddivisa in gruppetti di 9 od 8 bit, ciascuno recante la memorizzazione di un carattere (si parla allora di **byte**), o può essere vista nella sua interezza a contenere la codifica binaria (secondo varie modalità) di un numero.

Particolarmente interessante è il fatto che, usualmente, una parola di memoria consente l'immagazzinamento di una singola istruzione di programma (scritta nel linguaggio macchina, direttamente comprensibile da parte del sistema).

Va infatti tenuto presente che ogni programma, per essere eseguito, deve trovarsi in memoria: di qui viene letta ogni singola istruzione ed eseguita nell'unità centrale. Il programma sorgente (scritto in un linguaggio evoluto) viene sempre tradotto, come vedremo meglio più avanti, in una serie di

| | | | | |
|---|---|---|---|---|
| dato 1 | dato 2 | ..... | ..... | ..... |
| ..... | dato i | | ..... | ..... |
| ..... | ..... | ..... | dato n | ..... |
| istruz.1 | istruz.2 | ..... | ..... | ..... |
| ..... | ..... | ..... | ..... | ..... |
| ..... | ..... | ..... | ..... | istruz.m |

(1) Vedi Appendice D.

istruzioni semplici (scritte in linguaggio macchina) ognuna delle quali occupa, appunto, una parola: è questa **l'unità minima di esecuzione**.

Comunque, per quanto ci riguarda, potremo pensare alla memoria come ad una serie di celle contenenti le varie informazioni che servono al nostro programma (dati, numeri, caratteri...) nonchè l'intero programma, come si può vedere nella figura di pagina precedente.
Dal programma stesso avremo la possibilità di accedere a quelle aree di memoria che intendiamo utilizzare assegnando valori, leggendone contenuti e così via.

Ultima cosa da notare è che la memoria è un organo di tipo *passivo* e ciò nel senso che gli accessi che vi vengono fatti e la operazione che vi viene eseguita (scrittura in, lettura da) sono comunque gestiti dalla unità di controllo: è questa che impartisce i comandi e *controlla*, appunto, i prelievi o i depositi relativi al nostro magazzino di informazioni.

### 1.5 Unità di input-output

Proprio per il modo in cui le varie informazioni stanno in memoria (come abbiamo visto sono codificate in maniera opportuna), è necessario che quando comunichiamo un dato al sistema, esso venga tradotto secondo precisi criteri: è importante che venga riconosciuto il modo in cui è scritto e che il passaggio stesso dall'esterno verso l'interno o viceversa sia guidato e gestito in maniera compatibile alle varie altre operazioni che la macchina deve svolgere.

Di tutto ciò si occupa, sotto diretto controllo della unità centrale, quella che abbiamo chiamato **unità di input-output**: essa è, come la memoria, un organo passivo, nel senso che anch'essa è attivata e gestita dall'unità di controllo in modo diretto.

Vediamo più da vicino quali sono i compiti che deve svolgere per adempiere alle sue funzioni: il colloquio tra esterno ed interno del sistema (facciamo conto di avere un flusso di informazioni che devono essere depositate in memoria) avviene ponendo le informazioni su di un supporto leggibile da parte di l sistema (si tratta delle schede, dei nastri, dei dischi magnetici o del terminale di cui parleremo tra poco).

Per la lettura da tali supporti esistono degli organi appositi, detti **periferiche**, che, ad un segnale inviato loro dall'unità di input-output, provvedono alla lettura *fisica* del supporto trasformando le informazioni in segnali che via via vengono passati all'unità stessa.

A questo punto l'unità di input-output provvederà (assieme all'unità di elaborazione e sempre passando per l'unità di controllo) alla traduzione di

quei segnali nel codice di memorizzazione prefissato ed infine l'informazione codificata verrà trasportata in memoria (ancora sotto la gestione dell'unità di controllo).

Naturalmente non è detto che un sistema possa trattare un unico flusso di informazioni per volta, normalmente, anzi, avviene il contrario; né che l'operazione di input-output debba essere l'unica eseguibile in quel momento: magari contemporaneamente può venire elaborato qualcosa di diverso (1).
La gestione di questo alternarsi delle informazioni o delle operazioni da svolgere sarà ancora devoluta alla unità di controllo, ma l'esecuzione vera e propria, e quindi l'attivazione della periferica appropriata, il porla in attesa e così via, sarà compito dell'unità di input-output. Questa avrà anche la funzione di collegamento con la **console di sistema**: si tratta di un terminale (solitamente video) con tastiera, che colloquia direttamente col sistema e che viene utilizzato o per leggere ciò che vi avviene istante per istante o per permettere l'intervento umano diretto: l'operatore può lanciare da qui direttamente, a tutto il sistema, dei comandi di attivazione o arresto di particolari funzioni, di abilitazione o disattivazione di certi organismi, di lancio di determinati programmi, ecc.

## 1.6 Unità Centrale

Abbiamo già detto che la parte effettivamente attiva dell'elaboratore è l'unità centrale: essa può immaginarsi composta da due parti: l'**unità di controllo** e **l'unità di elaborazione**. Le sue funzioni possono riassumersi brevemente in questo: ogni operazione, ed in particolare ogni istruzione, viene esaminata, ed inviata in esecuzione nella corretta sequenza, da parte dell'unità di controllo; il lavoro effettivo è a carico dell'unità di elaborazione (proprio per questo, tale unità prende spesso il nome di unità algebrico-logica).

Perché queste funzioni possano essere svolte, si rende necessario l'utilizzo di alcuni **registri**: essi vanno considerati come dei depositi (hanno solitamente la struttura di parole di memoria ma non sono localizzati in questa unità) nei quali momento per momento il sistema deporrà l'istruzione da eseguire, il suo indirizzo in memoria, un dato da elaborare e così via...

Ad esempio, per eseguire una serie di istruzioni, il sistema procederà così: in un registro, detto **registro indirizzi**, verrà posto l'indirizzo della prima istruzione da eseguire, l'unità di controllo provvederà a prelevare dalla memoria quella istruzione servendosi di tale indirizzo e la deporrà nel **registro istruzione** dal quale essa verrà decodificata e passata all'unità di elaborazione.

---

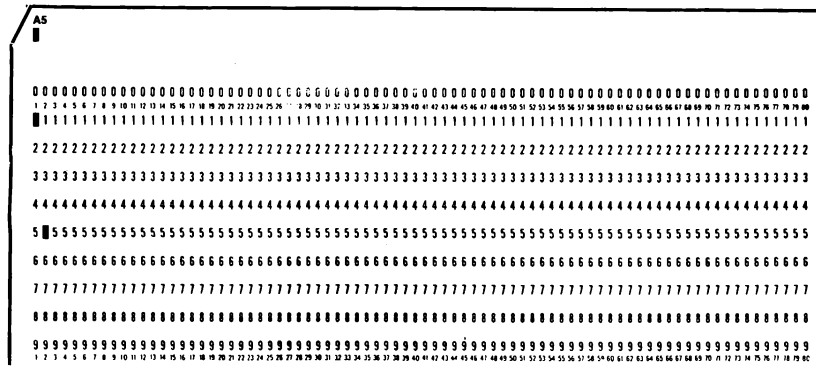
(1) Normalmente molti sistemi si occupano concorrentemente di programmi diversi. Ad esempio, mentre di un primo programma viene eseguita qualche operazione, di un secondo può essere effettuato l'input-output. Si dice, allora, che il sistema lavora in **multiprogrammazione**.

Questa la eseguirà, servendosi a sua volta di un certo numero di registri (registri di somma, di quoziente, esponente ecc...), contemporaneamente un registro (contatore istruzioni) verrà incrementato di uno e nel registro istruzioni verrà posta l'istruzione successiva, un eventuale risultato dell'operazione eseguita, che debba essere memorizzato, verrà depositato in memoria (sempre a carico dell'unità di controllo); poi partirà l'esecuzione della seconda istruzione e così di seguito per tutto il programma.

## 1.7 Le Periferiche

Abbiamo detto che l'input-output è guidato dall'unità, appunto, di input-output e che, perché questo avvenga effettivamente, sono necessari degli appositi organi detti **periferiche**.

Queste hanno ad esempio il compito di trasferire le informazioni dal supporto su cui sono scritte all'unità di input-output (e quindi in memoria). Il supporto almeno storicamente più importante è la **scheda**: si tratta di un cartoncino di circa 8x19 cm che va pensato suddiviso in 80 colonne di 12 elementi ciascuna. Il carattere viene ottenuto perforando in modo opportuno le varie colonne, così, ad esempio, nella figura che segue, la prima colonna porta perforato il carattere A, la seconda il numero 5 e così via nel codice Hollerith (1) che è quello più normalmente usato:



Tali schede vengono perforate per mezzo di una macchina detta **perforatrice di schede** (si pensi ad una macchina da scrivere che invece che stampare i caratteri produca delle perforazioni) e quindi data in lettura al sistema il quale utilizza per questo la periferica detta **lettore di schede**. Questa avrà il compito di verificare in quali posizioni sono presenti le perforazioni, traducendo così il codice Hollerith in segnali che l'unità di input-output utilizzerà per una traduzione, a sua volta, nel codice di memorizzazione.

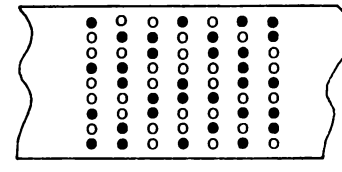
---

(1) Dal nome dell'inventore della scheda perforata (Vedi Appendice D).

Un'altra periferica in grado di trattare le schede è il **perforatore**, il quale provvede in output alla perforazione, sempre sotto il controllo dell'unità di input/output.

Si è detto che le schede sono il supporto più importante dal punto di vista storico in quanto questo è stato il primo a venire utilizzato. Oggi, tuttavia, hanno preso piede metodi più rapidi di input e output i quali utilizzano i supporti magnetici: le informazioni sono messe in input attraverso vari mezzi (ad esempio **terminali** (1) collegati al sistema) e registrate su **disco** o su **nastro**.

Il **nastro** altro non è che un normale nastro magnetizzabile come quello dei comuni magnetofoni, in esso le informazioni sono registrate con lo stesso formato che avrebbero in memoria, nel senso che la presenza o meno di magnetizzazione viene fatta corrispondere al bit; tali registrazioni sono altamente condensate, ottenendo il vantaggio di un grande volume di informazioni su uno spazio relativamente esiguo: alcuni tipi di nastri oggi possono giungere a contenere diversi milioni di caratteri.



pallino nero = magnetizzazione =1
pallino bianco = assenza di magnetizzazione =0

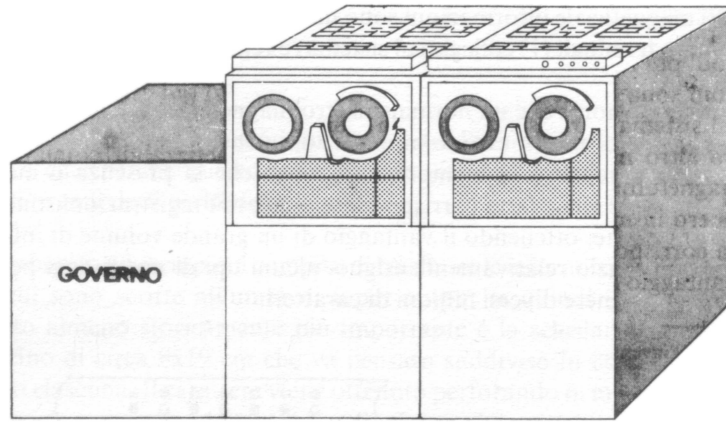
I nastri magnetici vengono letti o scritti dall'**unità nastro**. Questa è dotata di una testina magnetica, che provvede appunto alla magnetizzazione o alla verifica della presenza di questa, in modo assai veloce: si giunge a circa 500000 caratteri/sec.

Fisicamente l'unità nastro è poi composta di vari organismi (per il trascinamento delle bobine in modo regolare, per la protezione della testina e delle bobine stesse da agenti esterni, per il riscontro dell'inizio e fine del nastro) e da un **governo** che presiede al suo funzionamento (un solo governo provvede generalmente a più nastri).

Sullo stesso principio della magnetizzazione sono realizzati i dischi. Tale supporto viene normalmente chiamato **diskpack** ed è costituito da un certo

(1) Un **terminale** non è altro che uno strumento per comunicare a distanza col sistema. Esso di solito è costituito da uno schermo video per ricevere le informazioni e da una tastiera per inviarle.

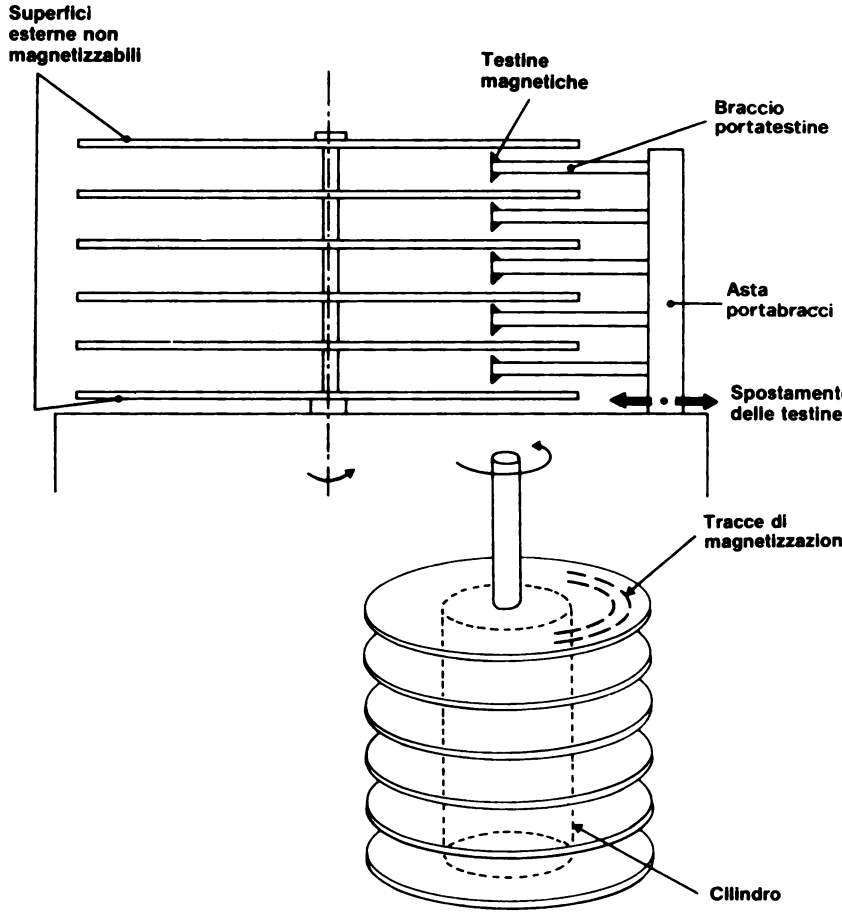
numero di piatti ricoperti, come i nastri, da una sostanza magnetizzabile. Sovrapposti su un perno comune, essi vengono letti (o scritti) da una serie di testine magnetiche, disposte a pettine, che possono spostarsi in senso longitudinale sulla superficie stessa dei piatti mantenuti in costante rapida rotazione.



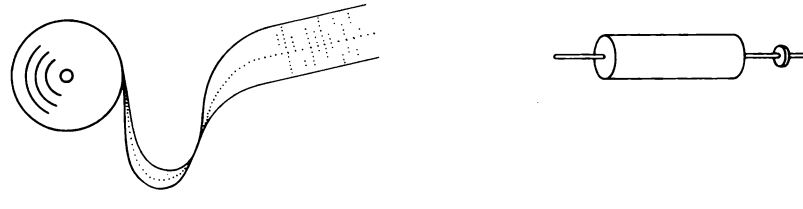

La registrazione (o la lettura) avviene su **tracce** concentriche in modo del tutto analogo ai nastri, al punto, anzi, che si può pensare ad una traccia né più né meno, come ad un nastro disposto su una circoferenza del piatto. La differenza sostanziale consiste nel fatto che su di un nastro le informazioni sono disposte in modo consecutivo, una dopo l'altra, mentre qui sono disposte diversamente: su di un cilindro immaginario composto dalle tracce corrispondenti sui vari piani dei piatti.

Come per i nastri, esiste per i dischi un'apposita unità, l'**unità dischi**, sulla quale viene montato il diskpack, dotata di un governo (un governo dischi può normalmente provvedere a più dischi) e di vari organismi di controllo (della rotazione, della stabilità e del movimento delle testine, ecc...).

Allo stesso modo dei dischi funzionano le cosiddette **diskette**: si tratta di dischi di piccole dimensioni (tra i 15 ed i 25 cm di diametro) e composti di un solo piatto magnetizzabile: la registrazione è ad alta densità ed il numero di tracce è sufficientemente elevato, tanto da permettere la memorizzazione di una notevole quantità di dati. Questo supporto va sempre più diffondendosi su tutti i tipi di elaboratori per il minimo ingombro e la facilità d'uso, ma è sicuramente il più diffuso sui più piccoli, i cosiddetti mini o microelaboratori, nei quali si rendono così disponibili tutte quelle funzionalità proprie dei dischi.




Va dato un cenno anche a due tipi di supporto che tendono, ormai, a cadere in disuso, ma che possono venire utilizzati in alcuni sistemi: si tratta della banda perforata e del cilindro. Per quanto riguarda la banda perforata

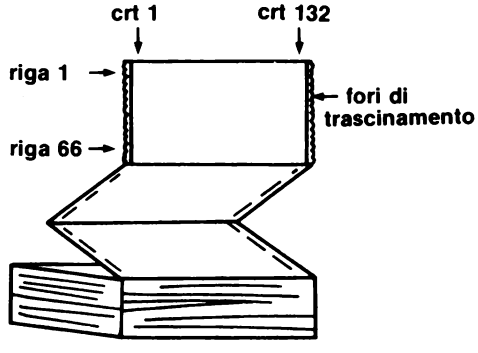


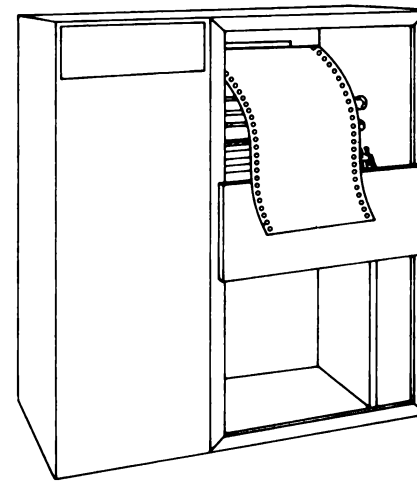
basterà dire che viene sfruttato lo stesso principio delle schede: un nastro di carta viene svolto sotto un lettore e l'informazione, perforata secondo un opportuno codice è letta, oppure, passando sotto un perforatore, viene

scritta. Il cilindro, altro non è che una superficie, appunto cilindrica, magnetizzabile al pari dei nastri e dischi.

Tralasciando ulteriori dettagli su questi ultimi due supporti, ci resta da descrivere l'unità di output per eccellenza: la **stampante**.

Questa è destinata a ricevere l'output delle varie esecuzioni e stamparlo su carta in maniera molto veloce (si possono raggiungere le 2000 righe di stampa al min.) e secondo i criteri scelti dall'utente (spaziatura tra le righe e salti di pagina guidati da programma, ecc.). La carta più normalmente utilizzata è destinata a raccogliere 66 righe per pagina, ognuna di 132 caratteri.



Per quanto riguarda le periferiche è opportuno spendere alcune parole anche sul **terminale**: si tratta di un dispositivo che permette funzioni di I/O di vario tipo, ma per le sue possibilità e per i vantaggi che offre in varie occasioni (o modi di lavorare), ci pare vada trattato a parte. Perciò rimandiamo alcuni dettagli al prossimo paragrafo, e per ora teniamo presente che nella configurazione più tipica esso può, almeno come termine lato, essere assimilato ad una *console lontana*, priva di funzioni di "controllo" effettive, ma capace di un particolare I/O.

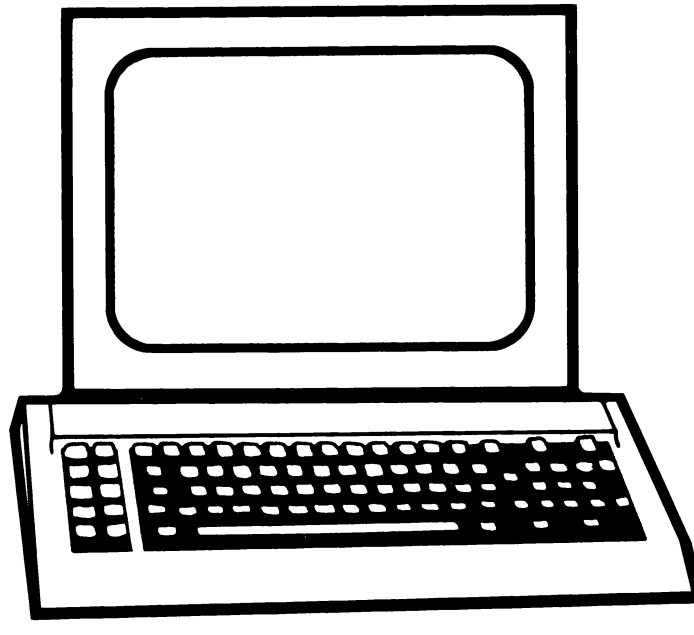
### 1.8 Il Terminale

Nel paragrafo precedente si è voluto descrivere il terminale come una *console lontana*, ovvero non posta vicino all'unità centrale od al sistema. In realtà ciò non è poi così importante. La posizione può essere scelta come aggrada; quello che è importante è che esso, pur non dovendo essere visto

come un organismo di controllo, deve essere considerato una interfaccia col sistema di tipo diretto ed immediato per le funzioni e le possibilità che vengono permesse all'utente.

Prima di procedere va anche detto che esso è attualmente uno strumento primario di lavoro e che la tendenza generale è quella di servirsene sempre di più a scapito delle altre periferiche. Almeno in alcuni casi (e lo vedremo tra breve) esse ne possono venire sostituite più che utilmente.

Per comprendere bene come ciò possa accadere, va dato, forse prima di tutto, uno sguardo a quella che è la sua struttura fisica (o hardware, almeno in senso lato). Si tratta di un video (come quello dei normali televisori domestici) dotato di una tastiera: l'utente può comportarsi come se avesse di fronte la solita macchina da scrivere: ciò che batte compare sul video, invece che sulla carta.



La cosa più interessante è che tale *messaggio* può venire direttamente inviato al sistema e che questo può *rispondere*, magari ancora sul video, fornendo il risultato di una elaborazione.

Si consideri, comunque, quanto detto a proposito della scheda: anche in quel caso si trattava di una *scrittura su tastiera*, però il messaggio andava successivamente passato al sistema attraverso l'apposita periferica: il lettore di schede. Nel caso del terminale, invece, quel messaggio (che compare sul video solo per nostra comodità) viene passato al sistema in modo diretto ed immediato, senza, per così dire, altri intermediari.

Altrettanto dicasi per l'output: la risposta del sistema, qualunque essa sia, comparirà sul video permettendoci di agire *al momento* come se stessimo parlando direttamente con esso.

Il fatto che il terminale possa agevolmente sostituire le schede è facile da comprendere quando si pensi che, battendo la perforazione su scheda, non facciamo altro che costruire una **riga messaggio** che successivamente l'apposita periferica (appunto il lettore di schede) interpreta e passa al sistema. Con il terminale non facciamo altro che battere e passare immediatamente al sistema la stessa **riga messaggio**, lasciando al terminale stesso il compito di transcodificarla.

Esistono poi, varie possibilità di lavoro. Una delle più importanti è forse quella di **edizione**: possiamo generare dei testi, ad esempio scrivere un intero programma, controllarlo e correggerlo sul video, e poi, con semplici comandi, chiederne l'esecuzione.

Tralasciamo le più o meno sofisticate possibilità che i vari tipi di terminali esistenti permettono, per notare invece come, sedendoci di fronte ad un terminale, non possiamo che trovarci *di fronte al sistema*, quasi esso fosse del tutto a nostra disposizione: possiamo parlare con esso senza intermediari e come se fosse a noi *dedicato*.
Un grande sistema svolgerà magari del lavoro anche per altri, ma noi avremo la sensazione che esso stia lavorando solo per noi (e si ricordi ancora quanto detto a proposito dell'interattività nel paragrafo 1.1).

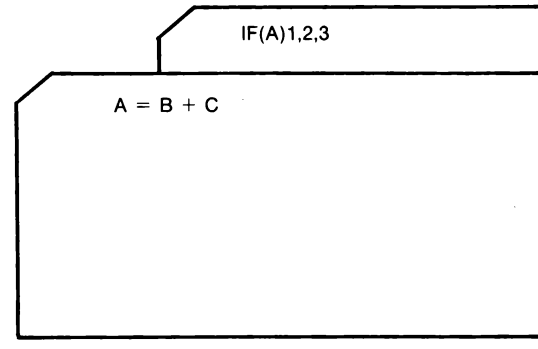
Da alcuni anni a questa parte vanno prendendo sempre più piede, ed a ragione, i **personal computer**. Essi altro non sono che dei sistemi di dimensioni assai ridotte, ma non per questo poco utili, che riuniscono in poco spazio le funzionalità di quello che abbiamo descritto come un *sistema tipo*. Ebbene, essi si presentano praticamente come un terminale che riunisce in sé varie componenti: l'unità centrale, le periferiche di stampa, un video; a volte anche qualcosa di più. Tuttavia la cosa di maggior interesse è che essi dispongono di un modo di interagire diretto ed immediato e che l'utente non ha altro da fare che sedersi di fronte a quella famosa *macchina da scrivere* che elabora e fornisce risposte di elaborazione in modo attivo ed immediato.

Prima di concludere, un'ultima considerazione che sarà utile approfondire prima di proseguire nella lettura: vediamo da vicino cosa significhi *battere* quella famosa *riga messaggio* su scheda o su terminale.

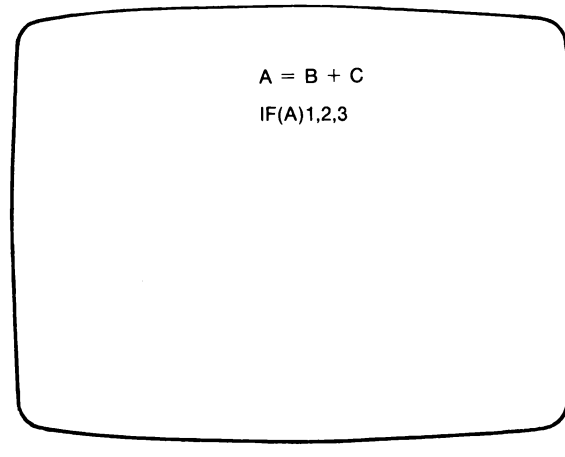
Se volessimo scrivere un'istruzione di linguaggio FORTRAN (ma, attenzione, delle istruzioni in sé parleremo più oltre), dovremmo generare, ad esempio, per le "righe messaggio":

```
A = B + C
IF (A) 1,2,3
```

delle schede come le seguenti:




mentre sul terminale comparirebbero, previa nostra battitura su tastiera, le righe:

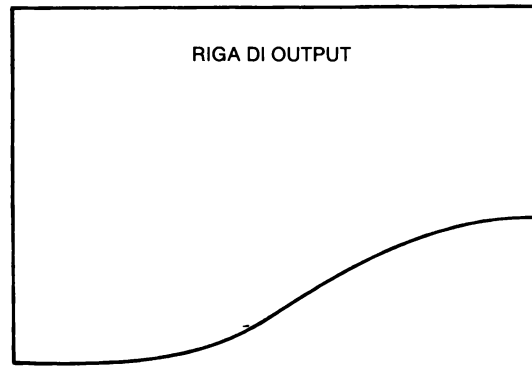



Insomma, ad ogni scheda corrisponde una riga di scrittura su terminale.

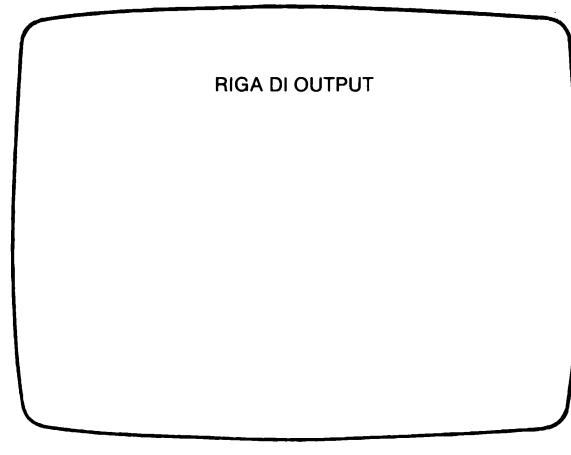
Ciò è assai importante per quanto seguirà: spesso parleremo di **scheda** per intendere, così è invalso nell'uso, la **riga messaggio**, o la riga di battitura su terminale (questo sarà messo in evidenza al massimo, ove ciò sia possibile).

E' ovvio che altrettanto vale per l'output: una riga di output potrebbe essere destinata al video del terminale e messaggi come quello che possiamo vedere

su carta (tale è il simbolo della figura che segue):

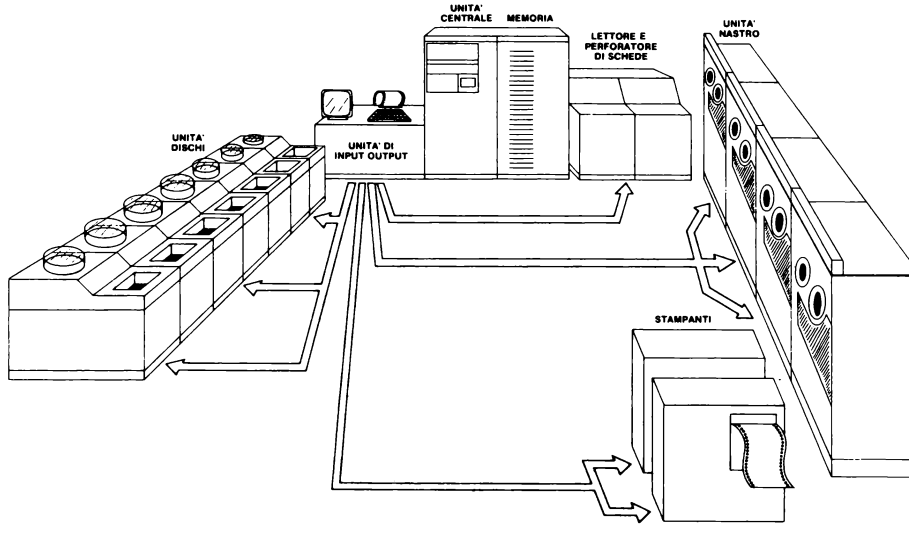



potrebbero anche essere inviati sul video definendo semplicemente da programma la loro destinazione. Insomma, il voler vedere comparire sul video:




non rappresenterà, per il programmatore, nessuna difficoltà e, soprattutto, nessun formalismo di programmazione diverso da quella che può essere la definizione della "destinazione" stessa dell'output.

Possiamo ora dare uno sguardo d'assieme al sistema tipo che ci eravamo proposti di descrivere: sempre tenendo presente che ogni configurazione è studiata su misura delle esigenze dell'utente, che il numero di periferiche, la disposizione e la potenza delle stesse, la capacità di memoria ecc. sono diverse caso per caso, riteniamo che un sistema medio possa venire configurato come nella figura seguente.




## 1.9 Il programma dalla sua progettazione alla sua esecuzione

Una volta visto come si debba procedere di fronte al nostro problema, e quali siano gli strumenti che siamo in grado di utilizzare, resta da analizzare la strada che il programma deve percorrere dalla sua progettazione fino all'esecuzione. Tutto questo deve essere compiuto in maniera completa, chiarendo gli aspetti collaterali anche se non riguardano direttamente il programmatore nell'applicazione che egli si propone. Tali aspetti lo riguardano, però, dal punto di vista di una coerente conoscenza di ciò che accade e di ciò che va fatto una volta che il programma entri in fase operativa.

Si è detto e ripetuto che il primo passo è costituito dallo studio del problema, dalla ricerca e conferma dell'algoritmo solutivo; questo va poi messo in termini schematici chiarendone e precisandone gli aspetti logici in un diagramma a blocchi e, soltanto a questo punto, si passa alla codifica.

Si è anche già detto che tale codifica consiste nella traduzione delle operazioni indicate, in modo abbastanza generico nel diagramma a blocchi, in un linguaggio preciso che la macchina è in grado di comprendere e a sua volta tradurre in operazioni da eseguire. Aggiungiamo subito che, per effettuare un tale passo, è ovviamente necessario essere padroni del linguaggio che si intende utilizzare: tale studio sarà il nucleo della seconda parte di questo libro. Ciò che per ora intendiamo esaminare è, ancora, del tutto indipendente dal linguaggio (è relativo al modo di funzionare dell'elaboratore) e va comunque appreso prima di iniziare a programmare effettivamente.

Supponiamo dunque di aver scritto una minuta del nostro programma traducendo ogni istruzione del diagramma a blocchi nelle frasi più o meno fisse, comunque precise, del linguaggio di programmazione scelto (ad es. FORTRAN).

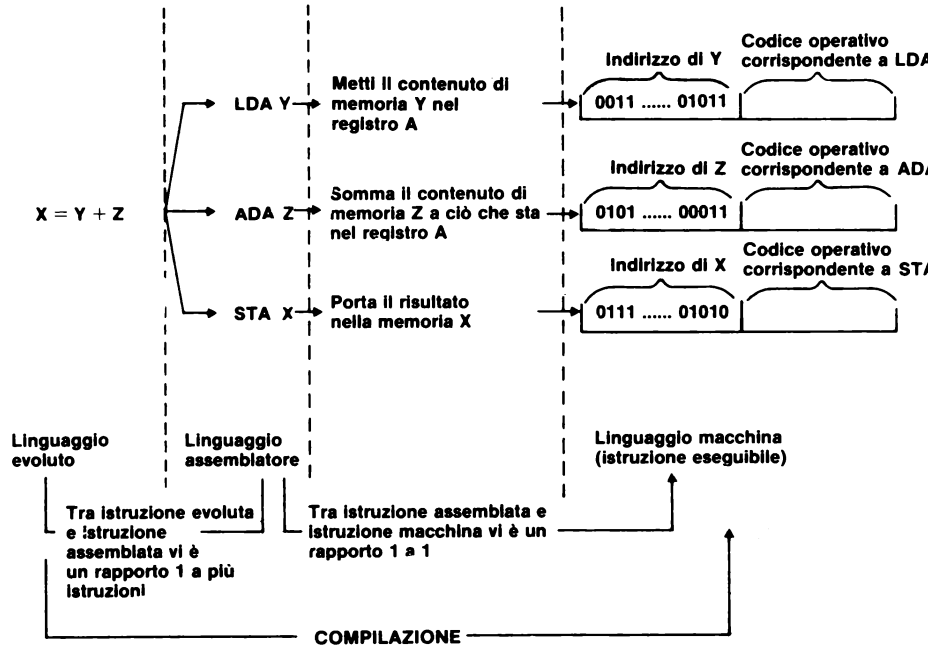
La parola *minuta* non è stata scelta a caso (anzi a volte si parla di minutazione piuttosto che di codifica), nel senso che quello di cui ora disponiamo è, effettivamente, una brutta copia scritta a mano di ciò che dobbiamo comunicare alla macchina e deve servire come base per la bella copia che alla fine si troverà sul supporto leggibile da parte dell'elaboratore.
Che la minuta sia scritta su fogliacci o su moduli prestampati in cui ogni carattere occupa una bella casellina tutta sua (come nelle parole crociate) poco importa: nel momento in cui il testo verrà copiato sul supporto scelto come input per l'elaboratore, ad es. le schede, si verificheranno sicuramente degli errori di copiatura. Dunque è importante, ancora una volta, la chiarezza, l'ordine e la precisione.

Ma partiamo dal presupposto di aver scritto sul supporto più adatto il programma codificato: si tratterà, ora, di vedere cosa farne. Permettiamoci subito una precisazione: si è detto che tale supporto potrebbe essere costituito dalle schede perforate, ciò tuttavia è un caso particolare: nulla impedisce che il programma venga scritto su un disco o su un nastro (e ciò con i sistemi più diversi).
Procederemo, comunque, come se disponessimo del pacchetto di schede con il nostro programma ivi perforato.

La prima cosa da fare sarà **compilare**. Si è già detto a grandi linee cosa significhi, ma ripetiamolo ed aggiungiamo qualche precisazione: il programma è scritto in istruzioni in un linguaggio più o meno evoluto (nel caso del FORTRAN sicuramente evoluto), l'elaboratore dovrà per prima cosa prendere ogni istruzione e smembrarla nelle varie operazioni particolari che la compongono in modo da tradurre ognuna in una serie di *operazioni macchina* eseguibili.
Vorremmo fare un esempio perché si possa capire bene cosa significhi questo lavoro di traduzione: immaginiamo un'istruzione di somma che, come vedremo, in FORTRAN è codificata, ad es., così: X = Y + Z.

Per eseguire delle operazioni aritmetiche l'elaboratore si serve di una area apposita, nell'unità di elaborazione, nella quale viene messo il dato preso dalla memoria; qui viene eseguita l'operazione addizionandovi l'altro addendo e, infine, il risultato viene portato in memoria (l'area dell'unità di elaborazione vien detta registro di somma e normalmente viene chiamato registro A).

Così l'istruzione X = Y + Z può venir tradotta in modo assai simile a quello del seguente schema (dipendente dal tipo di macchina):

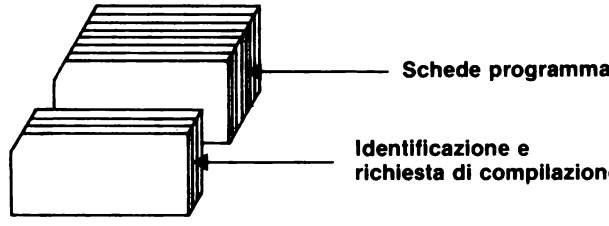



Mentre vien fatta tale traduzione, l'elaboratore controlla che le istruzioni del linguaggio siano scritte in maniera corretta secondo le regole e le richieste del linguaggio stesso. Se così non è segnala l'errore specificandone la gravità. Di solito esistono tre gradi di gravità: il primo e meno importante riguarda quelle istruzioni che non sono errate di per se stesse, ma che fanno eseguire del lavoro in più alla macchina, la segnalazione avverte che dovrebbe essere possibile agire in modo diverso risparmiando tempo ed evitando sprechi (la presenza di questo tipo di errori dipende strettamente dal tipo di macchina e non ci riguarda molto da vicino), tali errori vengono detti *di efficienza*; il secondo tipo di errori è legato al fatto che spesso l'operazione (aritmetica o meno che sia) è sicuramente possibile, ma in certi casi può condurre ad errori grossolani, tali errori vengono detti *warning* (avvertenza); il terzo tipo di errori è quello riguardante la forma e la sintassi del linguaggio, essi vengono definiti *fatal* (fatale, rovinoso) proprio perché la macchina, non comprendendo cosa vi sia da fare, dato che l'istruzione è scritta male, pur terminando la compilazione non passerà mai il programma in esecuzione.

Compilare permetterà di giungere non solo ad una completa traduzione, ma anche ad una completa serie di segnalazioni di tutti gli errori commessi in modo che si possa agevolmente correggerli.

La prima cosa da fare, dunque, sarà compilare il programma ripetutamente fino ad ottenere un *compilato* privo di errori: ad ogni compilazione si ottiene il

**listing** (ovvero la lista del programma, una sua stampa completa) con le segnalazioni degli errori: dovremo correggere almeno i fatal e ricompilare, fino ad ottenere un programma che ne sia privo. Si tratterà (sempre nell'ipotesi delle schede) di dare in input all'elaboratore un pacchetto di informazioni così fatto:




oppure digitando le corrispondenti righe-messaggio.

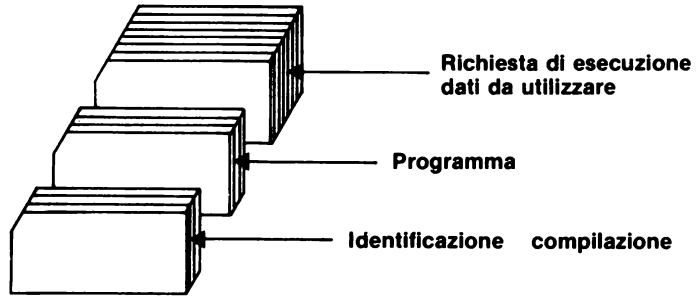
Come si vede andranno premesse al programma delle schede, che vengono dette di **controllo**, che rappresentano le nostre richieste all'elaboratore: nel caso specifico eseguire la compilazione. Ma come fa l'elaboratore a compilare? Servendosi, molto semplicemente, di un programma (modulo compilatore, uno dei *moduli di sistema*) registrato e permanentemente a disposizione della macchina in qualunque momento: la nostra richiesta, presente nelle schede controllo, provvederà a richiamare proprio quel programma **compilatore** che si incaricherà della traduzione del nostro programma e della segnalazione di errori.

Esiste un certo numero di programmi che, come il compilatore, risiedono su disco permanentemente registrati e che svolgono funzioni diverse: vi faremo sempre riferimento col nome di **moduli di sistema** (come sono tutti quelli che costituiscono il sistema operativo).

Una volta ottenuta una traduzione senza che vi siano state segnalazioni d'errore, questa sarà costituita da una serie di istruzioni scritte in **linguaggio macchina**, che rappresentano ciò che dovrà poi venir eseguito. Il programma così compilato, o, se si preferisce, così tradotto, viene usualmente chiamato **programma oggetto**. Esso non è caratterizzato soltanto dal fatto di contenere tutte le istruzioni di partenza tradotte in linguaggio macchina, ma contiene qualcosa di più: alcune istruzioni, a volte, per poter essere eseguite hanno bisogno di essere sostituite con una vera e propria serie di sottoprogrammi caratteristici e tipici che l'elaboratore ha a sua disposizione preregistrati: li chiameremo **routine di sistema**. Queste routine di sistema hanno un'importanza tutta particolare: esse sono già scritte in linguaggio macchina e quando viene fatta la compilazione alcune operazioni passeranno il controllo proprio a quelle di queste routine che servono in quel caso particolare.
Così il **programma caricabile** (quello che dovrà essere effettivamente eseguito) che risulta dalla compilazione, non sarà costituito solamente dalla traduzione

del nostro programma di partenza (**programma sorgente**), ma da questa e dalle routine di sistema necessarie alla sua esecuzione.

A questo punto perché il programma possa venir eseguito sarà necessario dare all'elaboratore un'istruzione apposita: otterremo ciò con altre schede controllo.




In quest'ultimo pacchetto di schede che aggiungiamo al lavoro che intendiamo eseguire, saranno dunque presenti almeno queste informazioni: richiesta di esecuzione, dati di input che il programma deve utilizzare, indicazioni relative a quali dischi, nastri o altre risorse sono necessarie.

La prima cosa che viene fatta nel momento in cui l'elaboratore legge la richiesta di esecuzione è il caricamento in memoria del programma oggetto con i relativi agganci alle routine di sistema necessarie. Tale operazione viene fatta da un modulo di sistema di nome **caricatore** il quale una volta terminata la sua funzione passa il controllo alla prima istruzione del programma, il che dà inizio all'esecuzione effettiva. Un lavoro come quello che abbiamo descritto nell'ultimo schema prende il nome di **compilazione e lancio** ed è forse il caso più tipico (e che viene più spesso adottato dai principianti); non dimentichiamo, però, che un programma (ed è ciò che poi viene normalmente fatto nei centri di elaborazione) può essere prima soltanto compilato, fino alla eliminazione degli errori, perchè poi se ne conservi una copia (magari registrata su disco) del solo programma oggetto, che verrà poi lanciato quante volte si vuole in esecuzione.

Detta così la cosa, sembra che le preoccupazioni siano terminate, ma non è vero: sia che siamo nel caso compilazione-lancio o meno, le prime esecuzioni, dopo l'eliminazione degli errori formali non è detto che siano esatte, anzi ciò accade piuttosto raramente. A questo punto possiamo infatti rilevare quegli errori logici che abbiamo messo nel programma, per cui... credendo di fargli fare una cosa, gliene abbiamo fatta fare un'altra. E' pur vero che durante la stesura del diagramma a blocchi abbiamo prestato attenzione ad evitare errori di tipo logico, ma non possiamo esserne certi; oppure, pur avendoli evitati, non possiamo essere certi di avere previsto tutti i casi particolari; oppure, ancora, possiamo aver sbagliato il modo di stampare i risultati, etc...

Proprio per questi motivi sarà importante che le prime esecuzioni vengano lanciate con *dati-prova* scelti appositamente, di facile verifica, che conducano a risultati facilmente controllabili. Tali dati dovranno inoltre essere costruiti in modo che il programma, per elaborarli, segua tutte le possibili vie logiche del diagramma a blocchi. Preparare un *gioco-prova* che risponda a questi requisiti non è sempre facile, ma certamente potremo raggiungere una certa sicurezza che il nostro programma elabori in modo corretto, solo dopo un'esecuzione su dati siffatti.

Anche a questo punto, tuttavia, non potremo avere la certezza matematica di aver previsto tutto: potrà sempre verificarsi un caso particolarissimo che manda in *terminazione anormale (abort)* (1) un programma prima funzionante correttamente.

Proprio per questo motivo è importante codificare il programma inserendo delle istruzioni che magari non compaiono nel diagramma a blocchi e che forniscono delle scritture, delle stampe, delle segnalazioni di ciò che sta avvenendo in quel momento: delle istruzioni, insomma, che ci permettano di controllare dei risultati parziali e di verificare che non si stiano commettendo errori. Ciò può essere ottenuto utilizzando istruzioni del tutto normali, ma esistono anche delle istruzioni particolari, proprie del tipo di macchina che stiamo utilizzando (e del linguaggio di programmazione usato in codifica) che possono spesso semplificare questo lavoro.

Non ci soffermiamo su queste ultime: vogliamo sottolineare invece l'importanza di questo modo di procedere, poiché è nella norma che un programma appena lanciato in esecuzione presenti le prime volte degli errori. Se così non fosse, tanto meglio; se invece dovessimo eseguire quei controlli di cui si diceva prima (ciò che prende il nome di **debug** del programma), dovremo farlo esaminando tutte le segnalazioni inserite e controllando, parallelamente, il diagramma a blocchi.

Una volta che il programma sia stato corretto e non presenti più errori, potremo anche togliere quelle istruzioni sovrabbondanti di controllo, passando il programma a quello che viene detto *funzionamento a regime*. Anche in tal caso, tuttavia, sarà bene non eliminarle tutte, poiché, come si era già notato, nulla toglie che un caso particolarissimo non conduca ad errori anche in questa fase. Sarà bene allora avere ancora a disposizione quelle istruzioni che ci permettono di effettuare il debug più strettamente necessario. Non solo, ma in un caso del genere sarà anche assai utile disporre ancora di quanto è stato fatto precedentemente, in particolare almeno del diagramma a blocchi, di commenti appropriati alle varie operazioni e dei risultati delle prove. Di qui la necessità di una documentazione completa ed esauriente sia del progetto della nostra procedura nella sua interezza, sia dei vari passi compiuti nel portarla a termine.

---

(1) Abort è il termine gergale ricavato dall'inglese ABnORmal Termination.

Indubbiamente l'esperienza è la migliore maestra per quanto riguarda il destreggiarsi nei casi sopraccitati, tuttavia gli avvertimenti che abbiamo fornito sono forse i più elementari, ma quasi sempre sufficienti perché un programmatore non debba perdersi in un lavoro troppo faticoso, in un labirinto inestricabile di istruzioni, perdendo così i vantaggi dell'uso dell'elaboratore.

# PARTE II
# FORTRAN IV

CAPITOLO 1

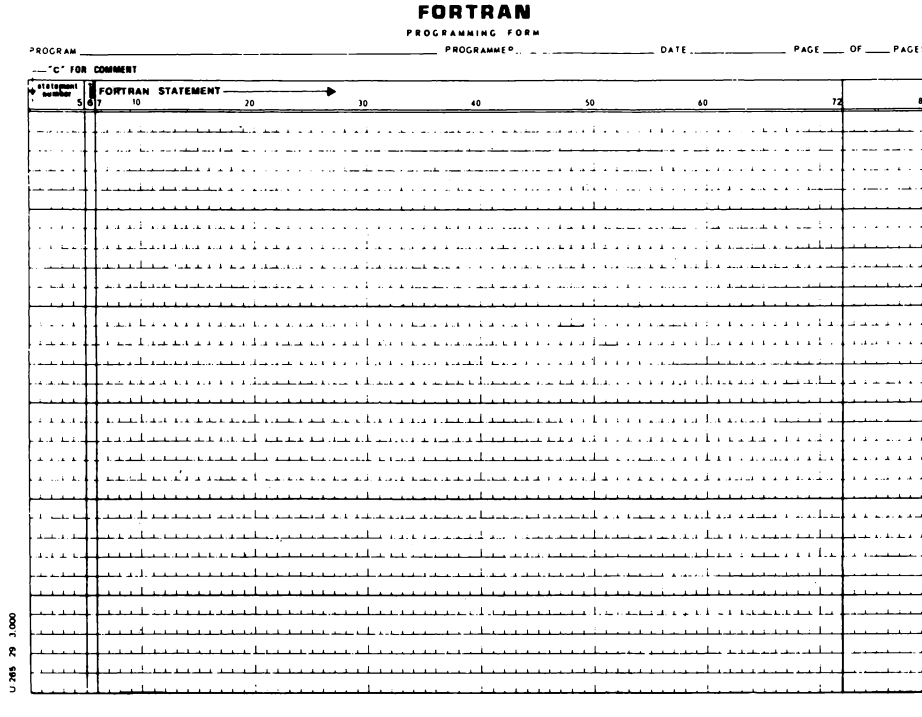
# I DATI E LE GRANDEZZE

## 1.1 Modalità di codifica

Come abbiamo visto, il passo che ci attende dopo la stesura del flow-chart (tramite il quale siamo giunti alla soluzione del problema che ci siamo proposti), è quello della minutazione.

Per ogni tipo di linguaggio di cui si faccia uso, e quindi anche per il FORTRAN, esiste un certo insieme di regole formali da rispettare, affinchè le istruzioni siano scritte in un modo comprensibile ed accettabile dal compilatore. Per rendere al programmatore più agevole il lavoro di scrittura, sono stati introdotti i *moduli di programmazione*: questi sono dei fogli (come in figura) costituiti da varie linee parallele suddivise in 80 caselle (o colonne) e, a seconda del linguaggio utilizzato, sono solitamente presenti varii delimitatori, in grassetto, che separano i diversi tipi di informazioni ccodificabili.

Osserviamo ora il foglio di codifica FORTRAN.



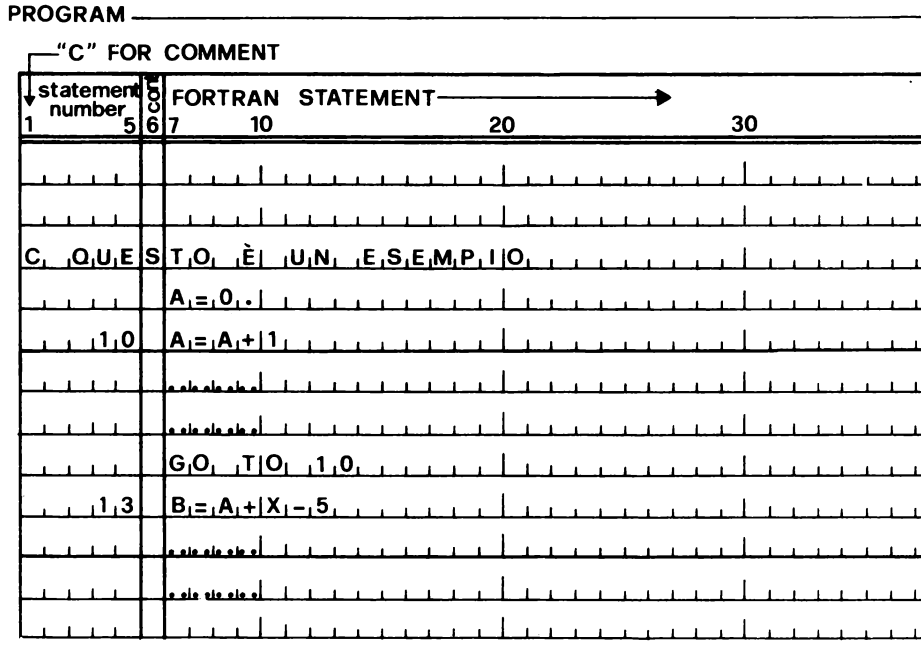
Possiamo vedere che sono individuabili essenzialmente 4 zone: la prima da colonna 1 a colonna 5, costituente il cosiddetto **campo label**, la seconda a colonna 6 nota come **campo di continuazione**, la terza da colonna 7 a colonna 72 che è il **campo operandi** e infine la quarta zona corrispondente al **campo identificazione**.

Nel campo label, ordinariamente, scriveremo delle *etichette* (label per l'appunto) che ci consentiranno di fare riferimento all'istruzione che si trova sulla medesima linea. In FORTRAN tali label sono costituite da un insieme di cifre (da una a cinque) solitamente accostate sulla destra del campo; occorre avere l'avvertenza di non assegnare mai a due istruzioni la stessa label: questo è dovuto al fatto che durante la fase di compilazione non dovrà mai presentarsi il pericolo di non riuscire ad individuare univocamente un'istruzione cui ci si voglia riferire, come accadrebbe in caso di uguaglianza delle label. Va chiarito che accostare le label sulla destra del campo è solo un accorgimento per rendere più uniforme la scrittura e la lettura da parte dell'uomo. In realtà in FORTRAN, esclusa qualche eccezione di cui parleremo, gli spazi (o blank) sono del tutto facoltativi ed utilizzabili a nostro piacimento.
Così tutte queste scritture nel campo label sono del tutto equivalenti e verranno interpretate allo stesso modo:

```
col.      5        5        5        5        5
          123  1 2 3    1  23 1  2 3    123
```

Ovviamente anche i blank iniziali, così come gli zeri iniziali, non alterano minimamente il *valore* di una label. Abbiamo messo in corsivo il termine *valore*, in quanto dobbiamo avere ben presente che i numeri che noi associamo ad un'istruzione hanno il solo significato di individuarla in mezzo alle altre, di costituire un possibile punto di richiamo nel programma: se ricordiamo la tecnica dei flowchart, sappiamo che, ad esempio, in un ciclo si ha la necessità di tornare ad una precedente istruzione: essa, quindi, dovrà essere richiamabile e le label soddisferanno proprio tale esigenza. Non ci sarà alcuna necessità di metterle in un ordine particolare, né le istruzioni con valore di label maggiore verranno eseguite prima o dopo altre con diverso valore di label. Ricordiamo dunque che queste sono null'altro che dei **nomi** (tutti diversi) associabili alle istruzioni.

Il campo label può inoltre essere usato anche per un'altra funzione: se in colonna 1 viene messa la lettera C, si vuole intendere che la linea di codifica in considerazione è una linea di commento, cioè non contiene istruzioni da elaborare, ma solo informazioni che noi vogliamo inserire per rendere chiaro cosa si sta facendo nel programma. Naturalmente tale linea di codifica non verrà presa in considerazione dal compilatore, ma verrà semplicemente stampata assieme a tutte le altre linee di codifica nel *listing* di compilazione.
Il campo che, nel foglio di codifica, segue quello che abbiamo appena esaminato è il **campo di continuazione**. La sua funzione è essenzialmente quella di rendere noto, al programma compilatore, che la linea di codifica che




si sta esaminando non è nuova, bensì è il proseguimento di una linea precedente contenente istruzioni. Quando verrà usato questo campo? Tutte le volte che una parola, una istruzione o una costante vengono interrotte in una riga e proseguite in quella successiva. Cosa dovremo inserire in questo campo? Una cifra, con l'accorgimento che, se si ha a che fare con più linee di continuazione consecutive, i numeri da inserire in questo campo devono essere in ordine non decrescente: ossia il numero presente in una linea di continuazione deve essere uguale o maggiore del numero presente nella linea precedente.

Quante di queste linee si possono inserire? Esiste chiaramente un numero massimo, che però dipende dal calcolatore in uso; in generale possono esserci la linea iniziale e 19 linee di continuazione.

E' il caso di mettere in evidenza il fatto che se si interrompe una linea di commento, quella che segue per completarla, non è una linea di continuazione, bensì un'altra linea di commento, che avrà quindi una C a colonna 1.

Un'altra cosa da tenere presente, a proposito delle linee di continuazione, è che quando si ha a che fare con una di tali linee, il compilatore ignora il contenuto del campo label, che può così essere utilizzato per inserire dei commenti.

Il terzo campo che consideriamo è il **campo operandi** che va da col. 7 a 72. E'

questo il campo in cui vengono inserite le istruzioni che costituiscono il programma (1).

L'ultimo campo da esaminare è il **campo di identificazione** che occupa le colonne da 73 a 80: è anche questo un campo che viene ignorato dal compilatore e che solitamente contiene dei caratteri che servono a identificare tutte le linee di codifica di uno stesso programma. Un possibile utilizzo è, ad esempio, quello di usare 4 caratteri per identificare il programma e gli altri 4 a formare un numero progressivo per linea, così da poter tenere conto anche della sequenza delle diverse righe di codifica.

## 1.2 Costanti, Variabili, Array

Come si è già affermato diverse volte in precedenza, il nocciolo centrale per la soluzione di un problema scientifico, è la determinazione dell'algoritmo relativo. Abbiamo altresì osservato che esso è dato, generalmente, da una o più relazioni analitiche o logiche. Sarà quindi opportuno vedere quali siano le informazioni utilizzabili e riconoscibili in un programma FORTRAN e renderci quindi conto di cosa incontreremo nella costruzione dei programmi. Dato che il linguaggio che vogliamo usare deve permetterci di esprimere tutte le informazioni presenti in una formula, possiamo subito individuare tutto ciò che deve essere rappresentabile in FORTRAN e precisamente costanti o variabili.

## 1.3 Le Costanti

Con il termine *costante* faremo riferimento ad ogni valore (numerico, logico o carattere) che viene definito come tale in fase di stesura del programma e mantiene quindi inalterato il suo valore durante tutta l'elaborazione.

In altri termini, se in una formula (ad es. $Y = X+2$) compare un valore costante (nel nostro caso il numero 2), esso verrà immagazzinato in una cella di memoria e non verrà mai alterato.

Poiché in una generica elaborazione ci si potrà trovare nella situazione di avere informazioni numeriche, logiche o alfanumeriche (formate cioè da insiemi di caratteri qualsiasi), una prima grossa suddivisione può appunto essere tra costanti numeriche, costanti logiche e costanti alfanumeriche. Considerando poi che i numeri utilizzabili nelle espressioni possono essere interi, razionali, reali o complessi, ne conseguirà che anche le costanti numeriche dovranno rispecchiare una suddivisione analoga, ed avremo allora costanti intere, costanti reali (in questo gruppo vengono conglobate sia quelle

(1) Non vi sono cose particolari da dire su questo campo, tranne che, con alcuni compilatori, è possibile inserire dei commenti dopo l'istruzione, separandoli tramite un apposito carattere.

reali che quelle razionali a cui, in verità, le precedenti vengono ridotte) e costanti complesse.

Esaminiamo ora in dettaglio le caratteristiche delle varie costanti, ricordando che in FORTRAN per separare la parte intera di un numero da quella decimale si usa il punto.

**COSTANTI**

| | | |
|---|---|---|
| **NUMERICHE** | **INTERE** | **numeri interi: 1, 100, 2, 1000 ...** |
| | **REALI** | **numeri razionali: 1.5, 2.7, 3.0, .01 ...** |
| | **COMPLESSE** | **numeri complessi: 2 + 3 i ...** |
| **LOGICHE** | | **valori alternativi: .TRUE.o.FALSE.** |
| **ALFANUMERICHE** | | **accostamento di caratteri qualsiasi** |

*Tab. 1*

*COSTANTI INTERE* - Viene considerato costante intera e memorizzato in una parola di memoria (1) un qualunque numero intero dotato di segno esplicito o implicito (in quest'ultimo caso solo se positivo (2)). In generale una costante intera, per essere immagazzinata come tale, non dovrà possedere più di 11 cifre.

Vediamo quindi alcuni esempi di costanti intere valide e non valide ed in quest'ultimo caso il relativo motivo.

| Costanti intere valide | Costanti intere non valide | Motivo |
|---|---|---|
| **12345** | | |
| **0** | | |
| **—1298** | | |
| **+3** | | |
| **—0** | | |
| | **1,234** | **Presenza del separatore delle migliaia** |
| | **128—** | **Segno dopo la costante** |
| | **123456789012** | **Numero con più di 11 cifre** |
| | **127.15** | **Presenza del punto decimale** |
| | **36.** | **Presenza del punto decimale** |

*Tab. 2*

(1) Vedi paragrafo 1.4 parte I.

2) Il valore massimo della costante dipende dalla dimensione della parola dell'elaboratore di cui si fa uso.

COSTANTI REALI - Per costante reale intendiamo qualsiasi numero dotato esplicitamente o implicitamente di segno, formato da una parte intera e da una decimale separate da un punto (punto decimale).

Tale costante viene immagazzinata in una parola di memoria e può essere costituita al massimo da un numero di cifre significative dipendenti dal calcolatore (che in ogni caso possiamo ritenere sia almeno di 7 cifre).

Le costanti reali possono essere anche scritte secondo un altro formato, e precisamente quello esponenziale: questo significa avere un numero, con eventuale segno esplicito ed eventuale punto decimale, moltiplicato per una potenza di 10.

Per indicare questa forma, si fa seguire il numero da una **E** (ad indicare l'esponente) e da un massimo di due cifre con eventuale segno esplicito. Vediamo anche in questo caso alcuni esempi di costanti valide e non valide con la relativa motivazione.

| Costanti reali valide | Costanti reali non valide | Motivo |
|---|---|---|
| 0. | | |
| —1.12 | | |
| 128.34 | | |
| +1280.186 | | |
| .0128E — 4 = .00000128 | | |
| 12.348E + 2 = 1234.8 | | |
| | 12 | Manca il punto decimale |
| | —127 E —138 | Numero di cifre dell'esponente superiore a due |
| | 12.R + 06 | Carattere indicante l'elevamento a potenza diverso da E |
| | 231412345678E + 02 | Troppe cifre significative prima dell'esponente |

*Tab. 3*

*COSTANTI REALI DOPPIA PRECISIONE* - Sono analoghe alle costanti reali, solo che vengono immagazzinate in 2 parole consecutive di memoria e quindi, in generale, permettono di rappresentare anche numeri con almeno il doppio di cifre significative rispetto alle costanti reali. Nella forma esponenziale, la lettera utilizzata è la **D**, invece della **E** e, con certi compilatori si possono avere anche 3 cifre di esponente. Vediamo ancora, nella Tab. 4, alcuni possibili esempi di costanti reali doppia precisione valide e non valide.

| Costanti reali doppia precisione valide | Costanti reali doppia precisione non valide | Motivo |
|---|---|---|
| 12345678901234.<br>—0.D0<br>+123D112 | | |
| | 12345678901234567890 | Troppe cifre |
| | 1.D1238 | Esponente troppo grosso |
| | 1.F12 | Simbolo dell'esponente errato |

*Tab. 4*

*COSTANTI COMPLESSE* - Poiché, come sappiamo, accade sovente nelle elaborazioni di carattere scientifico di avere a che fare con numeri complessi, è ovvio che debba rendersi possibile memorizzare anche tali quantità. In generale una costante complessa viene immagazzinata in due locazioni consecutive di memoria, di cui la prima contiene il numero che ne rappresenta la parte reale, mentre la seconda contiene il coefficiente della parte immaginaria. Per scrivere un numero complesso, se ne danno le due componenti, ossia la parte reale ed il coefficiente della parte immaginaria, separate da una virgola e racchiuse entro parentesi tonde. Ogni singola parte deve essere scritta come numero reale secondo una qualsiasi delle notazioni ammesse per tali numeri.

Presentiamo anche in questo caso, nella Tab. 5, alcune costanti complesse valide e non valide, con la relativa spiegazione.

| Costanti complesse valide | Costanti complesse non valide | Motivo |
|---|---|---|
| (—0., +0.)<br>(—1.E—2,2.1415) | | |
| | 1.,—2. | Mancano le parentesi |
| | ((1.,0.),(3.3,0.)) | Componenti a loro volta costanti complesse |
| | (A,B) | Componenti non numeriche |
| | (1.2.) | Manca la virgola tra le due componenti |

*Tab. 5*

*COSTANTI LOGICHE* - Oltre ai dati numerici, in FORTRAN è possibile maneggiare anche dei dati di tipo logico: dovranno quindi esistere i relativi tipi di costanti. Cosa si intenderà allora con tale termine? Una costante che dovrà indicare uno dei due valori logici possibili: ossia *vero o falso*. Come

sempre ci interesserà vedere come sono memorizzate tali informazioni e come esse possono essere scritte. Potremo allora dire che il valore di una costante logica viene mantenuto in una singola zona di memoria centrale, secondo modalità dipendenti dall'elaboratore, mentre la rappresentazione da parte del programmatore consisterà nella scrittura **.TRUE.** o **.FALSE.**. In essa occorre prestare attenzione al fatto che i punti precedenti e seguenti la parola sono parte integrante della costante. Per indicare una costante cui si vuol far corrispondere il valore logico *vero*, si userà la scrittura **.TRUE.**, in caso contrario si userà **.FALSE.** .

Vediamo ancora nella seguente Tab. 6 alcuni esempi di costanti logiche valide, non valide e il relativo motivo.

| Costanti logiche valide | Costanti logiche non valide | Motivo |
|---|---|---|
| .TRUE.<br>.FALSE. | | |
| | .VERO. | Valore non permesso come costante logica |
| | TRUE | Mancano i punti prima e dopo il nome della costante |
| | .FALSE | Manca il punto finale |

*Tab. 6*

*COSTANTI ALFANUMERICHE* - Vengono dette anche **costanti Hollerith** e servono per rappresentare insiemi di caratteri (spesso intestazioni) che devono essere memorizzati inalterati in locazioni consecutive di memoria. Queste costanti possono essere scritte in due modi diversi: il primo come l'insieme di caratteri costituenti la costante, racchiuso tra apici, il secondo tramite la scrittura $\mathbf{nHc_1c_2...c_n}$ dove **n** è il numero di caratteri che compongono la costante, l'allineamento $\mathbf{c_1\ c_2...c_n}$ è la costante in oggetto, costituita per l'appunto da **n** caratteri; ovviamente **H** è l'ottava lettera dell'alfabeto maiuscolo e serve per indicare che si ha a che fare con costanti Hollerith.

Se per esempio volessimo utilizzare i due metodi anzidetti per rappresentare la parola MILANO, potremmo scrivere 'MILANO' o 6HMILANO rispettivamente nel primo e nel secondo caso.

Quando è opportuno utilizzare una forma e quando l'altra? In effetti non vi è alcuna differenza ed il tutto dipende dalle preferenze del programmatore. Chiaramente col primo metodo non vi è il problema di dover contare preventivamente il numero dei caratteri costituenti la costante. Bisogna però prestare attenzione ad un fatto: se per qualche motivo all'interno della costante è

presente un apice, col secondo metodo non vi è alcun problema, mentre col primo vi è una difficoltà in quanto l'apice è proprio il delimitatore delle costanti alfanumeriche. Tale difficoltà è però sormontabile ricordando che se si vuole inserire l'apice all'interno di una costante espressa col primo metodo, basta inserire tale apice due volte consecutive. Così, se si volesse scrivere la parola L'AQUILA, basterebbe scrivere 'L"AQUILA', oppure, ovviamente, 8HL'AQUILA.

Va notato esplicitamente che quello delle costanti alfanumeriche è il caso in cui, come si accennava in precedenza, ha importanza il blank come carattere. Infatti, questa volta, il blank non ha la funzione di delimitatore tra i vari termini FORTRAN, ma è un carattere come qualsiasi altro all'interno della costante. Quindi, scrivendo una costante alfanumerica, non si può interromperla a piacere in un qualsiasi punto della linea di codifica e proseguirla nella linea successiva, mettendo un carattere di continuazione a colonna 7. Così facendo, l'insieme delle colonne a blank viene interpretato come facente parte della costante. Ne consegue che, volendo scrivere una costante alfanumerica che non sta su una sola riga, essa dovrà in ogni modo essere scritta sino ad occupare la colonna 72, prima di poter proseguire sulla nuova linea di codifica.

A questo punto ci si potrebbe chiedere: "Ma quando si usano tali costanti?". Vedremo in seguito che esse sono essenzialmente usate in concomitanza di un particolare enunciato (FORMAT) e permettono di costruire intestazioni utilizzabili in fase di stampa.

### 1.4 Le Variabili e il loro nome

Per giungere ad un certo risultato che non sia banale, in una qualunque elaborazione non si avranno solo dati costanti, ma, in base all'algoritmo risolutore del problema, si calcoleranno, e quindi si troveranno, dei valori che saranno la nostra soluzione finale. Poiché è noto che tutte le informazioni elaborate si trovano in celle di memoria, necessariamente accadrà che in alcune di queste le informazioni presenti muteranno valore o, se si preferisce, varieranno.

Queste celle di memoria in cui stanno le informazioni suscettibili di cambiamento, vengono per l'appunto dette **variabili**. Nel nostro programma noi le individueremo assegnando loro un nome univoco: è ovvio che mai il compilatore si dovrà trovare di fronte a situazioni dubbie e quindi ogni variabile dovrà avere un suo nome ovviamente diverso da quello di ogni altra. Sorge allora il problema di sapere come si scrive il nome di una variabile, quanto può essere lungo, quali caratteri si possono usare. La risposta è che i nomi possono essere composti da 1 a 6 caratteri alfanumerici (1), di cui il primo

---

(1) Esclusi: caratteri speciali quali, ad esempio, —, /, etc.

alfabetico. La scelta dei nomi è completamente libera, però è consigliabile usare nomi che ricordino la funzione per cui è utilizzata la variabile: così, se si dovesse usare una cella di memoria per un calcolo di un totale, non sarebbe opportuno chiamare ROSA tale cella (quantunque nessuno lo impedisca), ma sarebbe molto più chiaro il suo uso se la si chiamasse TOT o TOTALE.

Abbiamo detto che le variabili conterranno dei dati il cui valore muterà durante l'elaborazione. D'altronde sappiamo che i dati possono essere di diversi tipi (l'abbiamo appena visto esaminando le costanti), quindi occorrerà poter informare il compilatore di quali dati dovranno essere memorizzati in ogni variabile, per sapere come poter trattare le informazioni ivi contenute. Questo scopo è raggiungibile con tre modalità diverse, che ora esamineremo in dettaglio, e che prendono il nome di *regola del nome, definizione esplicita, definizione implicita.*

*REGOLA DEL NOME* - Abbiamo ora visto che ad ogni variabile bisogna assegnare un nome univoco, il cui primo carattere deve essere alfabetico. Se tale primo carattere è una delle lettere

**I,J,K,L,M,N**

allora significa che la variabile è da ritenersi intera (salvo eventuali altre definizioni contrarie, come vedremo in seguito). Se, viceversa, il primo carattere del nome della variabile è una lettera qualunque diversa da queste, si vorrà far riferimento ad una variabile reale (sempre salvo eventuali altre definizioni contrarie).

In questo modo, cioè con la regola del nome, possiamo individuare le variabili intere e quelle reali. Sappiamo però che le grandezze numeriche non sono solamente intere o reali, ma anche doppia precisione o complesse; inoltre esistono anche le grandezze logiche. Come sarà allora possibile informare il compilatore che un nome di variabile deve essere associato ad uno dei tipi ora detti? Questo sarà realizzabile tramite la cosiddetta *definizione esplicita.*

*DEFINIZIONE ESPLICITA* - Come dice il nome stesso, la definizione esplicita permette di associare ad una variabile un tipo. Esso viene espresso tramite il nome del tipo seguito dai nomi delle diverse variabili, separate tra loro da virgole.

| | |
|---|---|
| Tipo intero | INTEGER |
| Tipo reale | REAL |
| Tipo d.p. | DOUBLE PRECISION |
| Tipo complesso | COMPLEX |
| Tipo logico | LOGICAL |

Ad es. se si vogliono definire le variabili A, B, C come intere, le IX,JU come

reali, le DV, IV, MX come doppia precisione, ALFA, BETA, GAMMA come complesse e LOGC come logica, bisogna scrivere:

```
INTEGER A, B, C

REAL IX, JU

DOUBLE PRECISION DV, IV, MX

COMPLEX ALFA, BETA, GAMMA

LOGICAL LOGC
```

Va evidenziato in modo chiaro che, quando si usa la definizione esplicita, le variabili che si elencano hanno i nomi che devono soddisfare la sola condizione di non superare i 6 caratteri con il primo alfabetico, senza alcuna altra restrizione.

Ma i metodi per assegnare i tipi alle variabili non sono ancora terminati: vediamo ora l'ultimo.

*DEFINIZIONE IMPLICITA* - Quando si assegna esplicitamente un tipo a diverse variabili, è giocoforza elencare tutte le variabili di quel tipo, così se questo elenco è molto lungo si può avere il problema di scriverne esplicitamente tutti i nomi. E' possibile allora rendere meno complessa tale scrittura dando a tutte le variabili dello stesso tipo certe lettere iniziali che, in un'apposita istruzione verranno definite appunto come le iniziali per quel tipo. Tale assegnazione viene eseguita scrivendo:

IMPLICIT tipo ($C_1$, $C_2$, ... $C_n$)

dove: tipo è uno qualsiasi tra INTEGER, REAL, DOUBLE PRECISION, COMPLEX, LOGICAL

$C_1$, ... $C_n$ sono quei caratteri che, se vengono trovati come iniziale di un nome di variabile, fanno sì che ad essa venga associato il tipo precedentemente specificato. I caratteri possono essere messi in ordine qualunque; se si hanno più lettere dell'alfabeto consecutive, non è necessario scriverle tutte, ma basta mettere la prima e l'ultima separate da un trattino: in quest'ultimo caso bisogna ricordare che la prima lettera deve essere, nell'ordine alfabetico, precedente la seconda.

Così:

IMPLICIT tipo-1 ($i_1$, $i_2$, ... $i_n$), ....., tipo-k ($j_1$, $j_2$, ... $j_m$)
nomi che iniziano con $i_1$, $i_2$, ... $i_n$ sono di tipo-1
.....
nomi che iniziano con $j_1$, $j_2$, ... $j_m$ sono di tipo-k

ad esempio:

IMPLICIT INTEGER (A, B, C), REAL (M-Q)

fa sì che variabili con nome

A
ALFA
BIG
C
CIX

vengano considerate intere, mentre nomi come

MIX
N
OPS
QUA

vengano considerati di variabili reali.

Abbiamo visto i diversi modi di assegnare il tipo alle variabili: non abbiamo però mai detto che si sia obbligati a scegliere uno specifico metodo per fare le assegnazioni di tipo. Ne consegue che in uno stesso programma possono essere presenti tutti i diversi metodi che abbiamo ora visto. Per poter procedere senza equivoci occorrerà che tali metodi siano valutati con peso diverso, altrimenti si potrebbe facilmente verificare qualche condizione contraddittoria. In effetti, procedendo dal peso maggiore a quello minore, abbiamo che innanzi tutto vale la definizione esplicita, poi la definizione implicita e, per ultima, la regola del nome.

Ne consegue che se, ad esempio, una variabile CMPX è stata definita di tipo complesso in modo esplicito:

COMPLEX CMPX

esso sarà ritenuta di tale tipo anche qualora fosse presente una definizione come

IMPLICIT DOUBLE PRECISION (C-E)

che condurrebbe ad una definizione di tipo doppia precisione per tutte le variabili che iniziano per C (appunto), D ed E.

Nella seguente Tab. 7 possiamo vedere alcuni altri esempi di dichiarazioni di tipo:

| Esempi | Tipo | Osservazioni |
|---|---|---|
| I | Intero | |
| TOT | Reale | |
| COMPLEX ALFA, BETA, IPS | Complesso | |
| REAL INT, A | Reale | La variabile A sarebbe stata considerata in ogni modo reale |
| INTEGER I,RHO | Intero | La variabile I sarebbe stata considerata in ogni modo intera |
| ITOT | | |
| LAMBDA | | |
| REAL MON, NUM | | |
| IMPLICIT DOUBLE PRECISION (A,L—R) | | Sono considerate in doppia precisione le variabili il cui nome inizia con A,L,M,N,O,P,Q,R, escluse le variabili MON e NUM che sono reali. Ne consegue che ITOT è intera, mentre LAMBDA è reale doppia precisione. |

*Tab. 7*

## 1.5 Array

Sappiamo che, specialmente nel calcolo scientifico, ci si trova spesso a dover trattare quantità come i vettori e le matrici: di conseguenza si dovrà possedere anche in FORTRAN qualche mezzo che ci permetta di utilizzare tali enti (1).

In effetti è possibile definire, leggere, scrivere, elaborare sia vettori che matrici (che saranno denominati nel loro complesso **array**) e vedremo che le modalità con cui sarà possibile farlo, saranno abbastanza analoghe a quanto viene fatto nell'ordinario linguaggio matematico.

Vediamo di capire prima di tutto cosa possa voler dire avere un array in memoria.

Se ricordiamo che un vettore è dato da un insieme ordinato di n × m numeri (componenti del vettore) ed una matrice a n righe ed m colonne è una tabella ordinata di n × m numeri e così via, vediamo che è fondamentale poter individuare in modo univoco un insieme di locazioni di memoria con associato un nome e con la possibilità di accedere ad ogni suo singolo elemento. Questo obiettivo viene raggiunto assegnando un nome all'array (con la solita convenzione che il primo carattere sia alfabetico ed il tutto non superi i 6 caratteri) e mettendo entro parentesi tonda uno o più indici (a seconda delle

(1) Vedi appendice E

dimensioni dell'array), separati tra loro tramite virgole, per poter individuare il singolo elemento dell'array. Così col solo nome dell'array individuiamo un insieme di locazioni consecutive di memoria, costituenti nel loro complesso l'array, mentre col nome dell'array seguito dagli indici entro parentesi, si farà riferimento al singolo elemento. A loro volta gli indici potranno essere o costanti intere o nomi di variabili intere per le quali dovremo essere certi che il contenuto non sia inferiore a 1, né superiore al numero complessivo di elementi dell'array.

Possiamo quindi constatare che, prescindendo da piccole variazioni di carattere formale, il modo di individuare gli array o i suoi elementi è del tutto corrispondente a quello usato nella matematica ordinaria: infatti, se come solitamente in matematica, indichiamo con A B C delle matrici, in FORTRAN potremo ancora individuarle rispettivamente con A B C; se con $A_i$, $B_{k,l}$ indichiamo gli elementi di un vettore e di una matrice, la notazione corrispondente FORTRAN sarà A(I), B(K,L): come si vede c'è una analogia evidente.

Per quanto riguarda gli array in generale, vi sono alcune particolarità da tenere presenti: innanzi tutto le aree in cui dovranno essere memorizzati gli elementi dell'array saranno tutte dello stesso tipo, cioè se un elemento conterrà un intero, altrettanto varrà per tutti gli altri elementi dell'array, così pure dicasi per elementi reali, complessi ed il tipo di ogni singolo elemento dell'array, sarà il tipo dell'array nel suo insieme: quindi tramite il nome di questo si specificherà il tipo dei suoi elementi. Ovviamente anche per l'array vale l'insieme di regole atte a specificare il tipo, e cioè la regola del nome, la definizione implicita, la definizione esplicita.

Bisogna però ricordare che per attribuire il tipo viene dichiarato il nome dell'array nel suo complesso, e non un suo singolo elemento.

C'è poi un'altra cosa importante: in fase elaborativa si può pensare che, a seconda delle necessità, un array contenga un numero di elementi piuttosto che un altro. Al momento però in cui si scrive il programma, non si può restare nel vago o nell'incerto: occorre sempre informare il compilatore di quante locazioni di memoria dovrà essere costituito ogni array: di conseguenza dovrà esserci un enunciato, una cosiddetta *direttiva al compilatore*, in grado di svolgere questa funzione. L'enunciato in questione è il DIMENSION: il suo formato (ossia il modo con cui scriverlo) è:

DIMENSION $\alpha_1(n_1, n_2, ..., n_k), ..., \alpha_j(m_1, m_2, ..., m_h)$

dove $\alpha_1, ... \alpha_j$ sono nomi di altrettanti array di cui vogliamo assegnare le dimensioni

$k, ..., h$ rappresentano il numero di dimensioni di ogni array, e, a seconda dei compilatori, possono arrivare sino a 3 o sino a 7.

$n_1, ..., n_k$

$m_1, ..., m_h$ sono costanti intere senza segno, che rappresentano il numero di elementi relativi ad ogni dimensione dell'array.

Così, se vogliamo dimensionare un array A monodimensionale (vettore) di 15 elementi, scriveremo:

DIMENSION A(15)

se vorremo svolgere la stessa funzione per un array B a due dimensioni, di 20 righe e 30 colonne, e per un array C tridimensionale di 10 righe, 20 colonne e 30 "pagine", scriveremo:

DIMENSION B(20,30), C(10,20,30)

E' chiaro che, con questi enunciati, non c'è alcun pericolo di confusione col riferimento al singolo elemento dell'array, proprio perchè è presente la direttiva DIMENSION.

Abbiamo detto che con l'enunciato DIMENSION si devono mettere i nomi degli array ed entro parentesi, separati da virgole, le costanti intere che danno il numero degli elementi per ogni dimensione dell'array. Non è possibile mettere invece delle variabili intere cui, a seconda delle necessità, si facciano corrispondere certi valori piuttosto che certi altri? Diciamo subito che questo, in generale, non è possibile, proprio perchè in fase di compilazione occorre sapere esattamente quanta memoria centrale dovrà essere riservata per immagazzinare l'array. C'è però un'apparente eccezione a quanto ora detto, ed è il caso del DIMENSION variabile, che però verrà esaminato al momento opportuno, durante la descrizione dei sottoprogrammi.

Abbiamo visto per ora come sia possibile precisare al compilatore l'occupazione di memoria di un array e il tipo di dati che esso deve contenere. Vediamo come riferirci agli elementi che ci possono interessare.

Abbiamo già anticipato che ci sono due modi possibili per individuare un elemento: uno consiste nell'indicare il nome dell'array ed entro parentesi una o più costanti intere separate da virgole: in tal caso il riferimento sarà fisso e verrà individuato uno e un solo elemento, e precisamente quello il cui posto si troverà in corrispondenza delle posizioni indicate dagli indici. Così con la scrittura A (6,3,8) individueremo l'elemento che si trova nella sesta riga, terza colonna e ottava pagina (oppure ... all'incrocio della sesta riga e della terza colonna dell'ottava pagina).

L'altro modo, invece, consiste nell'usare come indici non delle costanti, ma dei nomi di variabili. Con tale metodo si ha ovviamente la possibilità di indirizzarsi ad elementi diversi, semplicemente cambiando il contenuto delle variabili che fungono da indice. In realtà si hanno possibilità un po' più ampie, rispetto a quelle viste sinora, per assegnare gli indici di un elemento di array: quelle descritte in precedenza si possono ritenere le forme *base* degli indici. In effetti è possibile usare come indice un'espressione aritmetica intera, purchè siano seguite certe regole e precisamente: se con $c_1$ e $c_2$ indichiamo due

costanti intere e con v una variabile intera, le forme possibili per gli indici sono:

$c_1$
v
$v \pm c_2$
$c_1 * v$
$c_1 * v \pm c_2$

Con questo metodo le costanti $c_1$ e $c_2$ non sono più costrette a variare tra 1 ed il valore massimo previsto per l'indice corrispondente, ma a tale condizione dovranno sottostare i valori delle espressioni costituenti gli indici.

Vediamo ancora una volta nella seguente Tab. 8 alcuni esempi di scritture di array e di elementi, sia esatte che errate:

| Scrittura valida | Scrittura non valida | Motivo |
|---|---|---|
| **DIMENSION A(10)** | | |
| **B(5,20,4)** | | |
| | **DIMENSION A** | |
| | **(1,5,7,3,4,2,8,9,6)** | **Troppe dimensioni** |
| | **DIMENSION A(10, B*I)** | **Presenza di un'espressione tra le dimensioni** |
| **A(1)** | | |
| **B(3,1,2)** | | |
| **B(I,K,L)** | | |
| **A(2*I – 3)** | | |
| | **A(1 + K)** | **Forma errata dell'indice** |
| | **A(I*2 – 3)** | **Forma errata dell'indice** |
| | **A(2*I – K)** | **Forma errata dell'indice** |
| | **A(–3 + 2*I)** | **Forma errata dell'indice** |
| | **A(B(I))** | **Indice a sua volta elemento di array** |

*Tab. 8*

### 1.6 Enunciati di assegnazione ed operatori

Esaminando il procedimento risolutivo di un problema tramite la tecnica dei diagrammi a blocchi si è potuto osservare che nei blocchi nei quali sono presenti delle operazioni (blocchi rettangolari), vi sono delle scritture, tipo A = —A, A = A + 1...che, in ultima analisi, possono ricondursi alla forma tipica:

$$V = e$$

dove:

V è un nome di variabile o di elemento di array (quindi NON una costante)

e è una qualunque espressione aritmetica o logica.

Tali scritture prendono il nome di **enunciati di assegnazione** ed hanno la funzione di aggiornare il valore della variabile al primo membro, tramite il valore calcolato per l'espressione al secondo membro. Ovviamente tale enunciato non va interpretato come uguaglianza tra il primo e il secondo membro ($A = A + 1$ non avrebbe senso); bensì come una sostituzione, tra il valore attualmente presente nella variabile al primo membro, ed il risultato dell'espressione al secondo membro. Ora diviene subito chiaro il significato dell'enunciato $A = A + 1$: viene considerato il valore attualmente contenuto in A, lo si incrementa di 1 ed il risultato viene posto ancora nella cella di memoria associata al nome A.

Come si vede, il tutto è piuttosto semplice, sennonchè occorre dare qualche ulteriore chiarimento sulle espressioni che costituiscono il secondo membro di un enunciato di assegnazione.

Si può ritenere infatti che esse abbiano in generale la forma:

$$t_1\ op_1\ t_2\ op_2 \ldots op_m\ t_{m+1}$$

dove ogni $t_i$ è un elemento costituente l'espressione e $op_i$ è un operatore che indica quale tipo di azione si intende svolgere tra i due termini. Così ad esempio $A + 2 — B$ è un'espressione nella quale A, 2, B sono i termini, mentre i simboli + e — sono gli operatori.

Anche le espressioni possono essere di diversi tipi, che vedremo in dettaglio in seguito: per ora ci accontentiamo semplicemente di individuarli e di darne una breve descrizione.

Innanzi tutto vi sono le **espressioni aritmetiche**, in cui i termini sono o costanti o variabili numeriche o, a loro volta, altre espressioni aritmetiche, mentre gli operatori sono gli usuali **operatori aritmetici**:

| | |
|---|---|
| + | simbolo di addizione |
| — | sottrazione |
| * | moltiplicazione |
| / | divisione |
| ** | elevamento a potenza |

Seguono poi le **espressioni logiche** in cui i termini sono, questa volta, costanti o variabili logiche, mentre gli operatori sono **operatori logici**:

| | |
|---|---|
| .NOT. | Simbolo di negazione logica |
| .AND. | Simbolo di intersezione logica |
| .OR. | Simbolo di unione logica |

Infine vi sono le **espressioni di relazione** nelle quali i termini sono costanti o variabili numeriche od espressioni aritmetiche, mentre gli operatori sono i

cosiddetti **operatori di relazione**:

| Operatore | Significato | Simbolo aritmetico |
|---|---|---|
| .EQ. | uguale | = |
| .NE. | diverso | ≠ |
| .GT. | maggiore di | > |
| .GE. | maggiore o uguale | ≥ |
| .LT. | minore di | < |
| .LE. | minore o uguale | ⩽ |

Vediamo ora in particolare ognuno di questi tipi di espressioni.

### 1.7 Espressioni aritmetiche

Come abbiamo già anticipato, un'espressione aritmetica è fatta collegando tra loro costanti o variabili numeriche o elementi di array tramite operatori aritmetici. Questi ultimi, come abbiamo visto, vengono rappresentati con i simboli: +, —, *, /, **. Ovviamente in una espressione possono essere contemporaneamente presenti tutti questi simboli: si presenta allora il problema: secondo quale modalità viene valutata l'espressione? Da sinistra verso destra? O viceversa? Oppure ancora partendo da qualche operatore particolare? E inoltre, come dovrà essere scritta in FORTRAN una espressione data secondo l'usuale formalismo del calcolo algebrico?

Risponderemo brevemente a tutte queste domande, partendo proprio da quest'ultima.

Siamo abituati, in matematica, a vedere espressioni tipo

$$a^2 + 3bx + 4 - cd^2y^3$$

ed ormai ci siamo talmente familiarizzati con esse, che non ci soffermiamo più ad analizzare cosa significhi in realtà questa scrittura, cosa invece molto importante qualora si decida di usarla per fornire informazioni durante la stesura di un programma. Prima di tutto, termini tipo $a^2$(1) possono essere interpretati sia come potenza di esponente 2 della base a, sia come prodotto di a per se stesso ed analogamente $d^2$ o $y^3$; inoltre nel monomio 3bx sottintendiamo sempre il simbolo di moltiplicazione, cosa che non è possibile fare nella scrittura dell'equivalente FORTRAN: il simbolo * va sempre esplicitato e non può essere sottinteso.

(1) O anche i risultati di certi sottoprogrammi, quali i sottoprogrammi FUNCTION che vedremo nel Cap. 5.

Dopo queste brevi premesse, vediamo allora come sarà possibile scrivere l'equivalente FORTRAN della formula sopra vista.

Ricordiamo che ogni dato che utilizzeremo nel programma, sarà memorizzato in un'opportuna cella di memoria, che è individuata (e quindi disponibile) tramite il suo nome. Se ricordiamo quanto detto in precedenza, i nomi delle variabili sono a discrezione del programmatore: nulla quindi ci impedisce di chiamare la variabile corrispondente ad una data lettera di un'espressione letterale con la stessa lettera, però in stampatello maiuscolo (ripetiamo che tutto quello che deve essere scritto sul foglio di codifica va in stampatello maiuscolo). Tenendo presente che i vari fattori di un monomio vanno interpretati come se ci fosse un simbolo di moltiplicazione, l'espressione prima vista va scritta:

$$A**2 + 3*B*X + 4 - C*D**2*Y**3$$

E' chiaro ora quale sia il modo per scrivere le espressioni di tipo aritmetico, tuttavia non è, per ora, altrettanto chiaro come esse vengano valutate. In concreto: esaminando l'espressione e, dimenticandoci per un attimo come è stata costruita, essa verrà considerata come se fosse:

$$a^2 + 3bx + 4 - cd^{2y^3}$$

oppure:

$$a^2 + 3bx + (4 - cd)^{2y^3}$$

... o che altro?

In base a cosa si può stabilire quale sia l'espressione che le ha dato origine? Diciamo subito che il filo conduttore che ci porterà a determinare in modo esatto le espressioni è la cosiddetta **gerarchia degli operatori** (Tab. 9).

Elevamento a potenza (**)
Moltiplicazione (*) e Divisione (/)
Addizione (+) e Sottrazione (+)

*Tab. 9*

In base a questa vengono eseguite per prime le operazioni relative ad operatori di grado gerarchico più elevato, poi quelle di grado man mano inferiore. E quando sono presenti più operatori dello stesso livello gerarchico (quelli che si trovano sulla stessa riga in tabella), essi vengono normalmente eseguiti da sinistra verso destra.

Ad esempio:

A/B*C equivale a $\frac{a}{b}$ c

$$A/B/C \text{ equivale a } \frac{\frac{a}{b}}{c}$$

$$A+B*C\text{—}D/E**2+1 \text{ equivale a } a+b\,c-\frac{d}{e^2}+1$$

Va notato esplicitamente che *non* è possibile usare consecutivamente due o più operatori aritmetici, non sarebbe cioè permesso scrivere A + —B.

In base a quanto appena scritto, vediamo subito che l'espressione esaminata risulta effettivamente la trascrizione FORTRAN dell'espressione aritmetica

$$a^2+3bx+4\text{—}cd^2y^3$$

senza alcun possibile dubbio.

Vediamo allora, tenuto conto di quanto dato in Tab. 9, di rappresentare in FORTRAN l'espressione

$$\frac{a-b}{c+d}$$

Se scrivessimo: A — B/C + D

tenuto conto delle regole ora viste, tale espressione FORTRAN sarebbe in realtà la trascrizione di

$$a-\frac{b}{c}+d$$

Si presenta quindi il problema di far riconoscere al compilatore che tutto un complesso di operazioni vogliamo venga eseguito in un certo ordine, prescindendo dal livello gerarchico degli operatori presenti. E' in effetti possibile ottenere un tale risultato introducendo le parentesi a riunire le sub-espressioni che noi vogliamo far calcolare con certe precedenze. Le regole per l'uso delle parentesi sono le stesse di quelle dell'ordinario calcolo letterale, tenendo presente però che sono utilizzabili solo parentesi tonde e che non c'è limite al numero di parentesi che possono essere utilizzate in un'espressione.

Tenuto conto di quanto ora detto, l'espressione:

$$\frac{a - b}{c + d}$$

viene tradotta in FORTRAN:

(A—B)/(C+D)

senza che possa sorgere alcuna possibilità di equivoco.

Possiamo allora modificare la precedente Tab. 9, in modo da tenere presenti anche le parentesi, ed ottenere quindi gli operatori, in ordine decrescente di livello gerarchico, nella seguente Tabella 10:

| |
|---|
| ( ) |
| ** |
| * / |
| + — |

*Tab. 10*

### 1.8 Tipo delle espressioni

In precedenza abbiamo visto che ogni costante o variabile numerica ha associato un *tipo* che permette di trattare in maniera adeguata il dato immagazzinato.

Poiché il risultato di un'espressione aritmetica dovrà essere un dato numerico, anche ad esso dovrà essere associato il relativo tipo; di conseguenza l'espressione aritmetica da cui deriva dovrà in qualche modo possedere la medesima caratteristica.

Come è possibile che si verifichi una simile circostanza?

Una prima risposta è che gli operandi presenti nell'espressione siano tutti dello stesso tipo, che è quindi quello del risultato.

Una simile condizione è però abbastanza restrittiva, ed in effetti la maggior parte degli odierni compilatori FORTRAN permette di non sottostare a tale limitazione.
Avremo quindi la possibilità di trattare espressioni di tipo misto, ossia espressioni nelle quali i diversi operandi possono essere di tipo diverso.

Qualora non si sia in queste favorevoli condizioni, le Tab. 11 e Tab. 12 ci forniscono l'insieme dei casi compatibili o meno, che si possono presentare

| $X_1$ \ $X_2$ | I | R | D | C |
|---|---|---|---|---|
| I | I | R | D | C |
| R | R | R | D | C |
| D | D | D | D | NO |
| C | C | C | NO | C |

*Tab. 11 - La presente tabella indica tra quali tipi di operandi siano possibili le quattro operazioni aritmetiche +, -, *, / (il NO indica quelle non permesse) per la maggior parte dei compilatori. Essa indica anche il tipo del risultato. (I = intero, R = reale, D = doppia precisione, C = complesso).*

| $X_1$ \ $X_2$ | I | R | D | C |
|---|---|---|---|---|
| I | I | R | D | NO |
| R | R | R | D | NO |
| D | D | D | D | NO |
| C | C | C | NO | C |

*Tab. 12 - La presente tabella indica tra quali tipi di operandi sia possibile l'operazione $X_1$ * * $X_2$ ( il NO indica i casi non permessi) per la maggior parte dei compilatori. Essa indica anche il tipo del risultato (I = intero, R = reale, D = doppia precisione, C = complesso).*

rispettivamente per le quattro operazioni fondamentali e per l'elevamento a potenza.

Negli altri casi, ormai più usuali, di compilatori che permettono una gestione più flessibile delle espressioni, ci sarà pur sempre il problema di determinare il tipo del risultato (o dell'espressione) una volta noto quello di ognuno degli operandi.

Se questi sono tutti dello stesso tipo, non c'è problema, e si ricade in quanto visto prima, mentre se gli operandi sono di tipo diverso, le cose saranno un po' più complesse. Infatti prima che venga valutata l'espressione, tutti gli operandi verranno convertiti nel tipo di ordine gerarchico maggiore, quindi verranno svolte le operazioni, secondo l'ordine visto in precedenza.

Attenzione però esiste un'eccezione: in generale il rapporto tra due grandezze intere viene calcolato come intero (7/3 = 2 !) anche se si trova all'interno di un'espressione di tipo gerarchicamente più elevato, ad esempio reale.

Da quanto detto consegue immediatamente che anche nei tipi esiste un livello gerarchico, che in ordine decrescente, è:

doppia precisione o complesso
reale
intero

Una precisazione particolare va fatta per quanto riguarda espressioni in cui compaiano operandi in doppia precisione e operandi complessi: con certi compilatori esse sono formalmente proibite, (come abbiamo specificato nelle

tabelle precedenti), mentre con altri gli operandi vengono convertiti in operandi **complessi in doppia precisione**, in cui sia la parte reale che il coefficiente dell'immaginario occupano ciascuno 2 parole di memoria. Ovviamente per sapere se certi tipi di espressioni sono permesse, occorrerà conoscere cosa consenta il compilatore che si usa, in ogni caso se si tiene conto di quanto espresso nelle due tabelle precedenti saremo senz'altro nelle condizioni minimali accettabili da tutti gli odierni compilatori.

Abbiamo quindi potuto vedere che oltre le costanti, le variabili e gli array, anche le espressioni hanno associato un tipo: è una caratteristica da tenere ben presente, in quanto ci tornerà utile tra breve.

### 1.9 Espressioni logiche

Sappiamo che sia le costanti che le variabili utilizzate in FORTRAN possono essere, oltre che aritmetiche (dei diversi tipi), anche logiche. Sarà allora possibile, utilizzando queste ultime, costruire, tramite opportuni operatori, delle espressioni logiche il cui risultato, una volta risolte, sarà un valore logico.

Ovviamente tutte le variabili coinvolte nell'espressione, dovranno essere precedentemente dichiarate logiche o esplicitamente (tramite l'enunciato LOGICAL) o implicitamente (tramite l'enunciato IMPLICIT LOGICAL).

Un'espressione logica sarà allora costruita combinando in modo opportuno costanti o variabili logiche ed operatori logici.

Gli operatori logici a cui è possibile fare ricorso sono: .NOT., .AND. e .OR.: è bene notare esplicitamente che il punto prima e quello dopo le lettere fanno espressamente parte dell'operatore logico, analogamente a quello che avevamo visto accadere per le costanti logiche .TRUE. e .FALSE..

Il modo di agire di un operatore logico può esse può essere individuato tramite una tabella, la cosiddetta **tabella di verità**, che mostra il risultato, una volta noti gli operandi e l'operatore.

Va subito ricordato che in generale non è possibile scrivere due operatori logici consecutivamente, con l'eccezione dell'operatore .NOT. che può essere messo di seguito ad uno qualsiasi degli operatori logici.

Ma vediamo in dettaglio come funzionano questi operatori, costruendo per ognuno di essi la relativa **tabella di verità**.

Operatore .NOT. — Questo operatore ha la funzione di cambiare il valore di verità di un'espressione logica. In Tab; 13 ne possiamo vedere la **tabella di verità**.

| A | .NOT.A |
|---|---|
| .TRUE. | .FALSE. |
| .FALSE. | .TRUE. |

*Tab. 13 - .NOT. (A è una qualunque espressione logica)*

*Operatore .AND.* - Questo operatore ha la funzione di collegare tra loro due espressioni logiche e di fornire il risultato logico .TRUE. solo se entrambe le espressioni logiche hanno tale valore .TRUE., e di fornire .FALSE. in caso contrario.

Vediamo compendiosamente quanto ora detto, nella seguente **tabella di verità**:

| A | B | A.AND.B |
|---|---|---|
| .TRUE. | .TRUE. | .TRUE. |
| .TRUE. | .FALSE. | .FALSE. |
| .FALSE. | .TRUE. | .FALSE. |
| .FALSE. | .FALSE. | .FALSE. |

*Tab. 14 - .AND.*

*Operatore .OR.* - Questo operatore permette di collegare tra loro due espressioni logiche e fornisce il valore logico .FALSE. solo se entrambe le espressioni assumono tale valore .FALSE.

Vediamo anche in questo caso compendiato il suo funzionamento tramite la **tabella di verità**:

| A | B | A.OR.B |
|---|---|---|
| .TRUE. | .TRUE. | .TRUE. |
| .TRUE. | .FALSE. | .TRUE. |
| .FALSE. | .TRUE. | .TRUE. |
| .FALSE. | .FALSE. | .FALSE. |

*Tab. 15 - .OR.*

Le espressioni logiche costruibili possono essere semplici (due operandi collegati tramite un operatore), oppure composte (successioni di operandi collegati tramite operatori). Nel primo caso valutare il valore di verità dell'espressione è una cosa piuttosto facile e non presenta alcun problema, mentre nel secondo ci troveremo ancora nella necessità di dover stabilire un criterio che ci permetta di valutare in modo univoco le espressioni rappresentate.

Ancora una volta ci soccorrerà il concetto di **gerarchia degli operatori**, in base al quale verranno valutate per prime le espressioni collegate tramite operatori di ordine gerarchico via via più basso.

Gli operatori dall'ordine gerarchico più elevato a quello meno elevato sono rispettivamente .NOT., .AND. e .OR. A parità di livello gerarchico degli operatori, l'espressione viene valutata da sinistra a destra.

A questo punto abbiamo una volta ancora un problema: se si trovasse scritta una espressione come

A.AND.B.OR.C.AND.D

e si volesse che questa venisse valutata vera solo se A e D hanno contemporaneamente valore .TRUE. e ancora fossero veri (sempre contemporaneamente) almeno uno tra B e C?

Chiaramente, secondo l'ordine gerarchico degli operatori, verrebbe valutata prima A.AND.B, poi C.AND.D, infine verrebbe fatto il collegamento dei due risultati, tramite l'operatore .OR.

Per quanto detto sinora, sarà sufficiente che una delle due sub-espressioni A.AND.B e C.AND.D risulti vera, perché tale sia il valore di verità dell'espressione complessiva. Si vede subito che ciò vale per A e B aventi entrambi il valore .TRUE. e qualsiasi sia il valore di C e D, oppure per C e D entrambe con valore .TRUE. e qualsiasi siano A e B: non era affatto questo quanto ci eravamo proposti scrivendo l'espressione.

A questo punto ci soccorrerà ancora una volta l'uso delle parentesi: infatti verranno valutate per prime le sub-espressioni poste entro parentesi e, nel caso di più parentesi l'una interna all'altra, la valutazione partirà dalle parentesi più interne.

All'interno di ogni coppia di parentesi le sub-espressioni verranno valutate secondo l'ordine gerarchico degli operatori utilizzati.

Così possiamo vedere nella seguente Tab. 16, in ordine decrescente di gerarchia, gli operatori che possono essere presenti in un'espressione logica.

( )
.NOT.
.AND.
.OR.

*Tab. 16*

Fatte tali precisazioni, possiamo agevolmente esprimere quanto si desiderava tramite l'espressione:

A.AND.(B.OR.C).AND.D

### 1.10 Espressioni di relazione

Queste espressioni, come dice il loro nome, indicano una relazione tra coppie di operandi. Gli operandi sono a loro volta espressioni aritmetiche, mentre gli operatori utilizzabili sono dati nella seguente Tabella:

| Operatore | Significa | Simbolo aritmetico |
|---|---|---|
| .EQ. | uguale | = |
| .NE. | diverso | ≠ |
| .GT. | maggiore di | > |
| .GE. | maggiore o uguale | ⩾ |
| .LT. | minore di | < |
| .LE. | minore o uguale | ≤ |

Tab. 17

Tutti gli operatori indicati hanno lo stesso valore gerarchico, quindi non si avrà più alcun problema di priorità, e le diverse parti dell'espressione verranno valutate da sinistra verso destra.

Anche se gli operandi sono espressioni aritmetiche, il risultato dell'espressione di relazione è un valore logico, **quindi essa può essere utilizzata come**

**operando in un'espressione logica** ed eventualmente per assegnare un valore ad una variabile logica.

L'espressione di relazione viene valutata calcolando prima di tutto il valore dell'espressione aritmetica di ognuno degli operandi, quindi viene valutata la relazione espressa dall'operatore e, se tale relazione è soddisfatta, il risultato assume il valore logico .TRUE., altrimenti assume il valore logico .FALSE..

| I | A | C | Espressione | Risultato |
|---|---|---|---|---|
| 2 | 3.1 | | I.LE.A. | .TRUE. |
| 8 | 2.7 | 5.3 | I.GT.A.+C | .FALSE. |
| 8 | 2.7 | 5.3 | I.GE.A.+C | .TRUE. |
| 8 | 2.7 | 5.3 | I—A.EQ.C | .TRUE. |
| 4 | 4.0 | | I.NE.A | .FALSE. |
| 2 | 5. | 7.8 | I*A.LT.C. | .FALSE. |

Ovviamente l'espressione più generale che si possa scrivere, può contenere sia operatori aritmetici, che logici, che di relazione; quindi in un'espressione comunque composta o complicata, la valutazione tiene conto dell'ordine gerarchico completo degli operatori, che è dato dalla seguente tabella:

**( )**

******

*** /**

**+ —**

**.EQ. .NE. .GT. .GE. .LT. .LE.**

**.NOT.**

**.AND.**

**.OR.**

*Tab. 18*

Come esempio, vediamo di valutare lo stato di verità della seguente espressione:

(I+1)*I+1.GT.K+K/K*K**K.OR..NOT.1/(K+1/K/(1+I))
.LT.K*(—K)**((K—K)**2).AND..NOT.I+K*K+I.EQ.I—K+1** K

per K = 1 e I = 0.

Per la valutazione richiesta in tale espressione mista, cominciamo prima di tutto a risolvere le diverse sub-espressioni, sostituendo i valori assegnati a K ed I.
Avremo allora successivamente:

(0+1)*0+1.GT.1+1/1*1**1.OR..NOT.1/(1+1/1/1(1+0)).LT.
1*(—1)**((1—1)**2).AND..NOT.0+1*1+0.EQ.0—1+1**1

1.GT.1+1.OR..NOT.1/(1+1).LT.1*(—1)**(0**2)).AND..NOT.1
.EQ.—1+1

1.GT.2.OR..NOT.1/2.LT.1.AND..NOT.1.EQ.0

.FALSE..OR..NOT..TRUE..AND..NOT..FALSE.

.FALSE..OR..FALSE..AND..TRUE.

.FALSE..OR..FALSE.

.FALSE.

Ovviamente i passaggi dettagliati vengono eseguiti quando le espressioni sono complicate: nei casi più frequenti basta farsi un po' di pratica con l'esercizio, per valutare rapidamente lo stato di verità di un'espressione.

Siamo così finalmente in grado di poter comprendere completamente come funziona un enunciato di assegnazione: quando si scrive una relazione tipo:

V = e

verrà eseguito quanto previsto dall'espressione a secondo membro ed il risultato verrà posto (cioè *assegnato*) nella variabile V a primo membro.

Il discorso potrebbe apparire concluso. In realtà non è così. Infatti, come abbiamo visto, l'espressione a secondo membro può essere di diversi tipi: aritmetica, logica, di relazione, oppure mista, cioè con sub-espressioni di ognuno di questi tipi. Ovviamente la variabile a primo membro dovrà coerentemente permettere di immagazzinare il risultato dell'espressione a secondo membro.

Possiamo quindi vedere nella seguente Tab. 19 la compatibilità dei tipi delle espressioni (e) e delle variabili di destinazione (V).

| V \ e | ARITMETICA | LOGICA | DI RELAZIONE | MISTA (LOG. E DI REL.) |
|---|---|---|---|---|
| ARITMETICA | SI | NO | NO | NO |
| LOGICA | NO | SI | SI | SI |

*Tab. 19*

Occorre dire ancora qualche parola nel caso in cui sia la variabile di destinazione che l'espressione al secondo membro siano di tipo aritmetico.

Come abbiamo già potuto osservare, in un'espressione aritmetica possono esservi operandi tutti dello stesso tipo, oppure di tipo diverso: nel primo caso l'espressione viene calcolata fornendo un risultato che è dello stesso tipo di quello degli operandi, mentre nel secondo caso il dato contenuto in ognuno degli operandi viene trasformato nel valore corrispondente, ma del tipo che nell'ordine gerarchico è il maggiore tra quelli esistenti nei diversi operandi. Non è però detto che il tipo dell'espressione così individuata, coincida con quello della variabile di destinazione. Come procederanno allora le cose? L'espressione verrà calcolata come abbiamo appena indicato, quindi il *risultato* verrà convertito corrispondentemente al tipo della variabile al primo membro. Conviene allora domandarsi se saranno possibili tutti i casi prevedibili a priori.

In generale possiamo ritenere valida la seguente tabella:

| V \ e | Intero | Reale | D.P. | Complesso |
|---|---|---|---|---|
| Intero | sì | sì | sì | no |
| Reale | sì | sì | sì | no |
| D.P. | sì | sì | sì | no |
| Complesso | no | no | no | sì |

*Tab. 20*

Vediamo ora con alcuni semplici esempi come funziona quanto appena detto.

Pensiamo di seguire la Regola del Nome e determiniamo:

$$I = K/L \qquad \text{per} \qquad K = 3 \qquad e \qquad L = 2$$

Si vede subito che varrà I = 1: infatti K è una variabile intera, L è anch'essa intera, dunque l'espressione viene valutata come intera ed il risultato viene posto nella variabile I (intera).

Sia invece A=K/L per K=3 e L=2

Questa volta avremo A = 1.: infatti l'espressione è identica alla precedente, ma questa volta la variabile di destinazione A è reale.

Se dovessimo invece calcolare:

I=B/C con B=3. e C=2.

il risultato sarebbe I = 1; in quanto l'espressione a secondo membro verrebbe calcolata come reale (sia A che B sono reali), mentre la variabile di destinazione è intera.
E infine, calcolando:

A=B/C con B=3. e C=2.

si otterrà A=1.5

E ancora, se dovessimo calcolare (1):

I=K/L+B/C con K=3 L=2 B=3. e C=2.

il risultato sarebbe

I=2

In modo del tutto analogo il calcolo di

A=K/L+B/C

con gli stessi valori degli operandi del caso precedente, fornirebbe

A=2.5.

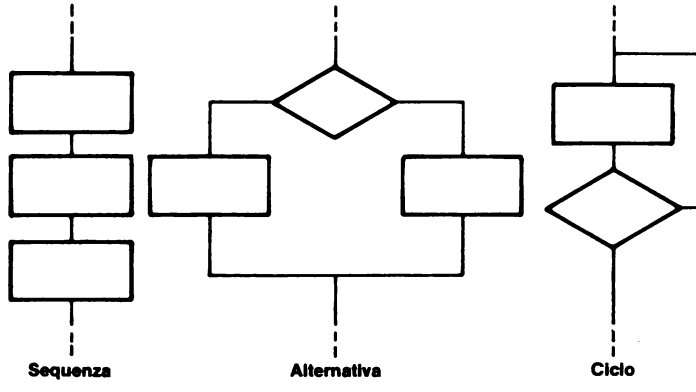
(1) Vedi paragrafo 1.8: Tipo delle espressioni.

CAPITOLO 2

# ENUNCIATI DI CONTROLLO

## 2.1 Il diagramma a blocchi e gli enunciati di controllo

Come abbiamo potuto notare nella costruzione dei flow-chart, durante la soluzione logica di un problema, ci si trova a seguire strutture di vario tipo e precisamente: sequenziale, alternativa e ciclo.




Dal punto di vista della codifica in linguaggio FORTRAN dovrà allora esistere qualche verbo che permetta di ottenere le funzioni previste da queste strutture e, più in generale, di far fronte a qualunque richiesta logica che preveda qualcosa di diverso da una stretta sequenzialità delle istruzioni.

Diciamo subito che il FORTRAN permette di ottenere la struttura sequenziale tramite, ad esempio, un insieme consecutivo di enunciati di assegnazione, mentre per le strutture di alternativa e di ciclo mette a disposizione quelli che vengono detti **enunciati di controllo**.

## 2.2 GO TO incondizionato

Il primo enunciato di controllo che esaminiamo è il GO TO incondizionato. Il suo formato è:

GO TO n

dove: n è un numero di label (numero che compare nelle colonne 1-5) di un'istruzione eseguibile, cioè di un'istruzione che prevede una vera e propria azione e non una dichiarazione di tipo, assegnazione di area o altro (vedi Appendice B).

La funzione di tale enunciato, una volta che venga incontrato in fase esecutiva, è quella di far saltare senz'altro all'istruzione con associato come label il numero che viene indicato dopo il GO TO.

*Esempio:*

```
      . . .

      . . .

10    A=A+.5

      . . .

      . . .

      . . .

      GO TO 10

      . . .

      . . .

      . . .
```

una volta giunti ad eseguire GO TO 10, si ritorna ad eseguire l'istruzione A = A + .5.

Va tenuto presente che un comando GO TO incondizionato **non** deve richiamare se stesso.

### 2.3 GO TO calcolato

Anche questa è un'istruzione di salto: ci permette cioè di uscire dalla stretta sequenzialità di esecuzione delle istruzioni, però è condizionata nella sua azione dal verificarsi o meno di certi eventi.

Il suo formato è:

$$\text{GO TO } (n_1 , n_2 ,\ldots,n_k ), i$$

dove $n_1, n_2, ..., n_k$ sono k label numeriche, anche non distinte tra loro, corrispondenti ad altrettante linee di codifica;
i è il nome di una variabile intera.

Qual'è l'effetto di una tale istruzione?

La spiegazione è molto semplice: in fase di esecuzione, se il valore della variabile i è 1, si salterà alla linea di codifica associata alla label $n_1$; se i vale 2, si salterà alla linea di codifica associata alla label $n_2$, se è 3 si salterà a $n_3$ e così via. Se il valore di i non è né 1, né 2,..., né k, si andrà in sequenza. Come si vede tale comando è particolarmente utile per codificare un test a più uscite, quando si può associare in qualche modo ad ogni uscita un risultato intero compreso in un certo intervallo di numeri consecutivi.

Se tutti gli n fossero uguali, il comando sarebbe *quasi* equivalente ad un GO TO incondizionato. Ovviamente vale quel *quasi*, in quanto se l'intero i è diverso da 1, 2,...,k si va in sequenza.

Vediamo di chiarire meglio con qualche esempio:

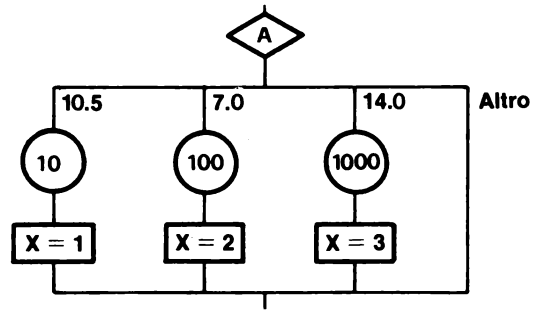
GO TO (10,20,10,30,30,40), I

significa che se I vale 1 si salta all'istruzione associata alla label 10,
se I vale 2 si salta all'istruzione associata alla label 20
se I vale 3 si salta all'istruzione associata alla label 10
se I vale 4 si salta all'istruzione associata alla label 30
se I vale 5 si salta all'istruzione associata alla label 30
se I vale 6 si salta all'istruzione associata alla label 40

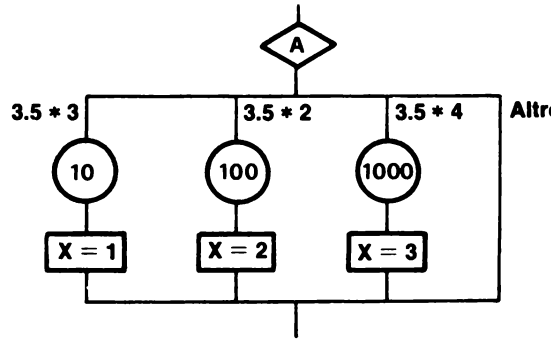
se I assume un valore diverso dai precedenti, si va in sequenza.

GO TO (10,10,10,10), I fa saltare all'istruzione associata alla label 10 se I risulta essere 1 o 2 o 3 o 4, in caso contrario si va all'istruzione che segue immediatamente il GO TO.

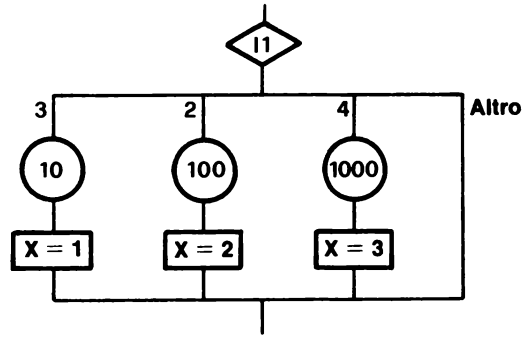
Vediamo ora come è possibile invece codificare il test indicato da questo blocco.



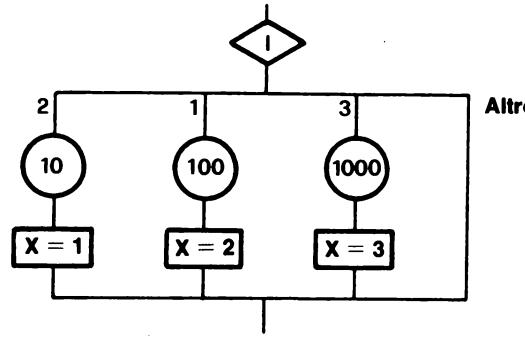
Come si può osservare, le condizioni d'uscita si possono immaginare scritte:




per cui, pensando ad una variabile I1 = A/3.5, si potrebbe considerare il test equivalente:




Non siamo però ancora nelle condizioni utili per l'impiego del GO TO calcolato, in quanto, se è pur vero che le condizioni d'uscita sono legate a valori interi consecutivi, questi però non partono da 1. Questo ovviamente non è un problema: basta considerare il test sulla variabile I = I1—1 per ottenere:



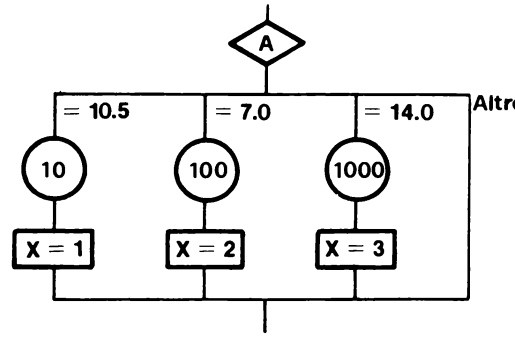

tornando quindi alle condizioni previste per il GO TO calcolato.

Perciò, partendo dal nostro caso iniziale, il test potrebbe essere scritto

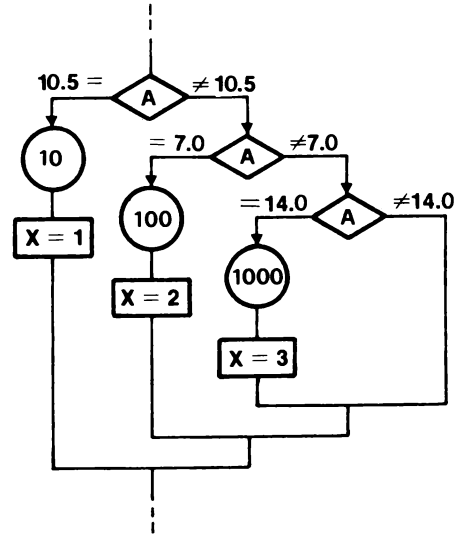
```
. . .
I = A/3.5—1
GO TO (100,10,1000),I
. . .
```

A questo punto, comunque, desideriamo aggiungere alcune considerazioni relative a come l'alternativa a più uscite possa essere ricondotta a più strutture alternative semplici, come vuole il Teorema di Jacopini-Böhm (1) e come ci eravamo preoccupati di precisare parlando dei flow-chart.

Così nell'esempio ora visto, il blocco:




può essere ridisegnato, volendo rispettare il teorema:




(1) Vedi Paragrafo 1.1 - Parte I.

Ovviamente il GO TO calcolato non è adatto ad un rispetto formale stretto del teorema, che vorrebbe una codifica come quella del secondo disegno. Tuttavia molti problemi di carattere scientifico risultano sicuramente assai semplificati dall'uso dei test a più uscite. Ciò ci porta a non sconsigliarne l'uso, nonostante le indicazioni, che si erano inizialmente date, di rispettare quanto più possibile il teorema utilizzando solo le tre strutture.

### 2.4 GO TO assegnato

E' questo un ulteriore tipo di comando GO TO che si presenta nella seguente forma:

GO TO i,($n_1$ ,$n_2$ ,...,$n_k$)

dove i è il nome di una variabile intera
$n_1$ ,...,$n_k$ sono k label numeriche associate ad altrettante linee di codifica.

Il modo di funzionare di questo comando è abbastanza diverso rispetto a quello del GO TO calcolato. Infatti perchè un GO TO calcolato esplichi la sua azione, è sufficiente che all'atto della sua esecuzione vi sia, nella variabile i, un valore intero.

Per quanto riguarda invece il GO TO assegnato, occorre che preventivamente alla sua esecuzione si *assegni* alla variabile di controllo i un valore di una label.

Come è possibile realizzare tale fatto? Non certo tramite uno degli enunciati di assegnazione che abbiamo visto in precedenza, bensì tramite un enunciato particolare e cioè l'enunciato ASSIGN.

Tale enunciato, utilizzabile solo in associazione ad un GO TO assegnato, ha la funzione di assegnare ad una variabile intera il valore di una label che, ricordiamolo, non è altro che un indirizzo richiamabile a programma.

Il formato dell'enunciato ASSIGN è il seguente:

ASSIGN n TO i

dove n è una generica label numerica associata ad un'istruzione
i è la variabile intera usata in un GO TO assegnato.

L'enunciato ASSIGN deve precedere, in fase esecutiva, il GO TO assegnato che lo utilizza.

Solitamente le label $n_1$, $n_2$,...,$n_k$ che compaiono nel campo dell'enunciato GO TO i,($n_1$,$n_2$,...$n_k$) costituiscono l'insieme di tutti i punti di programma a cui deve essere possibile saltare con tale GO TO. Però non viene eseguito alcun controllo per assicurarsi che non vengano fatte anche assegnazioni diverse. In pratica l'elenco delle label che vengono scritte nell'istruzione, serve semplicemente al programmatore per tenere presenti le diverse destinazioni accessibili da parte di quel GO TO.

Precisato tutto questo, siamo ora in grado di specificare dettagliatamente il funzionamento del GO TO assegnato: una volta incontrato l'enunciato ASSIGN $n_1$ TO i, alla variabile intera i resterà associato il valore dell'indirizzo corrispondente alla label $n_1$; incontrato quindi l'enunciato GO TO i,($n_1$ ,...,$n_k$) si salterà all'istruzione associata alla label $n_1$.

Vanno ricordate due cose importanti: non è possibile utilizzare un normale enunciato di assegnazione, come I = 3 se si vuole eseguire GO TO I,(5,3,7,91) saltando all'istruzione associata alla label 3; e viceversa, una volta utilizzato ASSIGN 3 TO I, non è possibile scrivere N = M*I, volendo moltiplicare M per 3, in quanto il contenuto di I non è il valore intero 3, ma l'indirizzo di memoria dell'istruzione associata alla label 3. In ogni caso le variabili utilizzate con degli enunciati ASSIGN possono essere sempre riusate a patto che si trovino una prima volta come primo termine in un usuale enunciato di assegnazione.

Vediamo ora di chiarire ulteriormente con un esempio: si voglia eseguire una stessa operazione su dati diversi che vengono elaborati man mano nel corso del programma. Una possibile codifica che risolva tale problema potrebbe essere:

```
      . . .
      . . .
      I=2
      A=8.5
      B=7.3
      ASSIGN 10 TO K
      . . .
      . . .
      . . .
1     ALFA=A*I+B**I—7
      GO TO K,(10,20,30)
      . . .
      . . .
      . . .
```

```
10    A=A+.05
      I=I+1
      B=B*I—5.
      ASSIGN 20 TO K
      . . .
      . . .
      . . .
      GO TO 1
      . . .
      . . .
      . . .

20    A=A+I—B
      I=I—2
      B=B+I—A
      ASSIGN 30 TO K
      . . .
      . . .
      . . .
      GO TO 1
      . . .
      . . .
      . . .

30    .....
      . . .
      . . .
      . . .
```

## 2.5 IF logico

L'IF è un comando che serve per analizzare delle condizioni e, in base al risultato dell'analisi, prendere opportunamente delle decisioni. A seconda del modo con cui vengono analizzate le condizioni, è possibile esprimere tale comando utilizzando due diverse modalità, che vengono dette IF logico e IF aritmetico.

L'**IF logico** è un enunciato che ha la seguente struttura:

IF(*espressione*) *istruzione*

dove *espressione* è un'espressione che fornisce un risultato logico: può quindi essere o un'espressione logica, od una di relazione;

*istruzione* è una qualunque istruzione eseguibile ad eccezione di un altro IF logico od un DO (che vedremo tra poco).

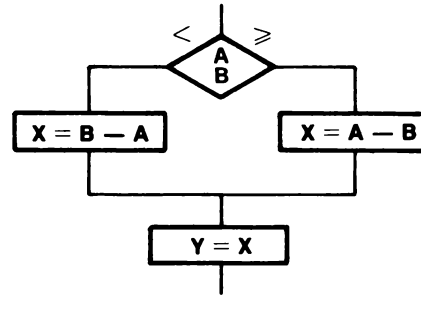
Le modalità di funzionamento di tale comando sono molto semplici: in fase esecutiva viene analizzata l'espressione entro parentesi: se essa fornisce il valore logico .TRUE., viene eseguita l'istruzione posta a seguito dell'IF, mentre se l'espressione assume il valore logico .FALSE., si passa all'istruzione che si trova nell'enunciato che segue l'IF.

Così se ad esempio volessimo incrementare un contatore IP qualora in un'area A si trovi un valore positivo, potremmo scrivere:

```
...
...
...
IF(A.GT.0) IP=IP+1
X=A*B+IP
...
...
...
```

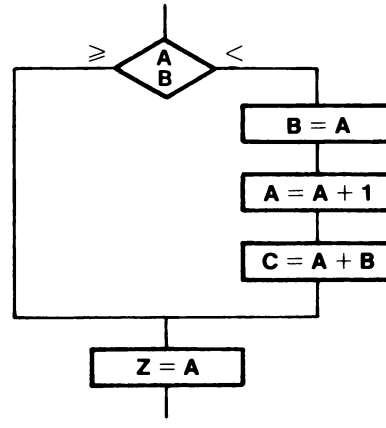
in tal modo, se il contenuto di A è minore o uguale a zero, il valore di IP resta inalterato, mentre viene incrementato di 1 se il contenuto di A è positivo; X viene calcolato comunque dopo il test.

Un altro esempio può essere il seguente, esso ci permette di vedere come ci si comporta nel caso di una struttura alternativa come questa:




```
...
...
...
    IF(A.GE.B) GO TO 1
    X = B—A
    GO TO 2
1   X=A—B
2   Y=X
...
...
...
```

Ovviamente nel caso di un'alternativa tra l'eseguire o meno un gruppo di istruzioni, il problema risulta semplificato; così:




```
      . . .
      . . .
      . . .
      IF(A.GE.B) GO TO 5
      B=A
      A=A+1
      C=A+B
5     Z=A
      . . .
      . . .
      . . .
```

## 2.6 IF aritmetico

Anche l'enunciato **IF aritmetico** ci permette di prendere diverse decisioni a seconda del risultato della valutazione di un'espressione. Il suo formato è infatti:

IF (*espressione*) $n_1$ , $n_2$ , $n_3$

dove: *espressione* è una espressione aritmetica non complessa;

$n_1$ ,$n_2$ ,$n_3$ sono tre label numeriche (non necessariamente tutte distinte) associate a degli enunciati eseguibili (vedi Appendice B).

Anche in questo caso la modalità di funzionamento è molto semplice: l'espressione aritmetica entro parentesi viene confrontata con lo zero. Se il risultato dell'espressione è minore di zero, si salterà ad eseguire l'istruzione

associata alla label $n_1$, mentre se è nullo si salterà all'istruzione associata alla label $n_2$, ed infine se il risultato dell'espressione è positivo, si salterà all'istruzione associata alla label $n_3$.

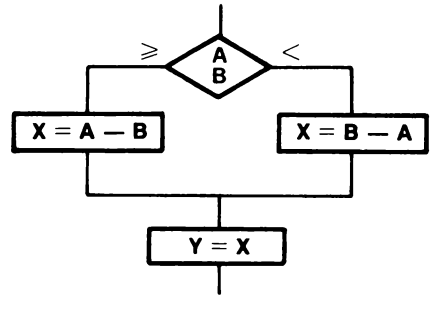
Per come è costituito il comando, che prevede una destinazione per ogni condizione si possa verificare, nel caso in cui $n_1 = n_2 = n_3 = n$, l'istruzione svolge *quasi* la stessa funzione di un GO TO incondizionato. Va ribadita l'importanza di quel *quasi*, perchè se è pur vero che, qualunque sia il risultato sul test dell'espressione aritmetica, si salta all'istruzione associata alla label n, purtuttavia occorre sempre analizzare un'espressione e questa deve essere di tipo aritmetico (non complesso).

Un ultimo avvertimento: è possibile sottintendere uno o più degli $n_1$, $n_2$, $n_3$ intendendo con ciò rimandare all'istruzione successiva; ad esempio:

```
IF(A—B)3,,3
```

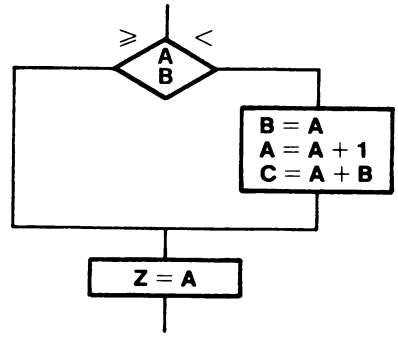
significa che se $A - B \gtrless 0$ il rimando è alla label 3, se invece $A - B = 0$ viene eseguita l'istruzione immediatamente successiva.

L'IF aritmetico può spesso agevolmente sostituire quello logico, come possiamo vedere relativamente agli esempi già analizzati:




```
      . . .
      . . .
      . . .
      IF(A—B) ,1,1

      X=B—A
      GO TO 2
1     X=A—B
2     Y=X
      . . .
      . . .
      . . .
```

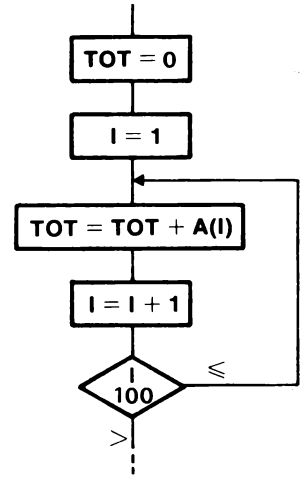



```
      . . .
      . . .
      IF(A—B) ,5,5
      B=A
      A=A+1
      C=A+B
5     Z=A
      . . .
      . . .
      . . .
```

## 2.7 Le istruzioni DO e CONTINUE

In molti casi, lo abbiamo già detto nel capitolo della programmazione, può presentarsi la necessità di ripetere ciclicamente una o più istruzioni.

Si ricorderà la struttura **ciclo**: se ad esempio, per ottenere la totalizzazione degli elementi di un certo vettore A(100) avessimo disegnato un flow-chart come il seguente:



potremmo agevolmente codificarlo con le istruzioni che già conosciamo (in pratica IF, GO TO e qualche operazione aritmetica):

```
      . . .
      . . .
      . . .
      TOT=0
      I=1
5     TOT=TOT+A(I)
      I=I+1
      IF(I.LE.100) GO TO 5
      . . .
      . . .
      . . .
```

tuttavia lo stesso flow potrebbe venir codificato più agevolmente utilizzando un'istruzione che il FORTRAN ci mette a disposizione proprio per una più semplice gestione di tutte le situazioni di **ciclo**. Si tratta dell'istruzione DO che ha il formato:

DO m n=h,k,i

dove: m è la label dell'ultima istruzione del gruppo che va eseguito ciclicamente e che chiameremo **rango** del DO;

n è il nome di un indice (intero) che permette il controllo delle *volte* che il ciclo viene eseguito e prende il nome di **variabile di controllo** del DO, esso non deve comunque essere ridefinito all'interno del rango del DO;

h è il valore iniziale dell'indice n (può essere un'espressione) e deve essere intero;

k è l'ultimo valore di n per il quale viene eseguito il gruppo di istruzioni di DO (può essere un'espressione) e deve essere un intero positivo (nel caso h > k le istruzioni del rango del DO vengono eseguite una ed una sola volta);

i è l'incremento che subisce l'indice alla fine dell'esecuzione di ogni ciclo di DO (tale esecuzione termina quando si verifica la condizione h > k) e deve essere intero positivo (può essere un'espressione), esso può essere omesso, nel qual caso è come sottintendere un incremento di 1.

L'esempio precedente può allora essere facilmente codificato così:

```
      . . .
      . . .
      . . .
      TOT = 0.
      DO 5 I = 1,100
    5 TOT = TOT + A(I)
      . . .
      . . .
      . . .
```

intendendo con ciò che il gruppo di istruzioni che vanno da quella seguente il DO a quella di label 5 inclusa (in questo caso in pratica solo quella di label 5) devono venir ripetute 100 volte: la prima volta I varrà 1, la seconda 2, ... fino a 100; quando I varrà 101 non verrà più ripetuta l'istruzione di label 5 e verrà eseguita quella immediatamente successiva. Notiamo subito che un eventuale riutilizzo della variabile I è sempre possibile, pur di effettuarne una riassegnazione. Una assegnazione iniziale, invece, non è necessaria, nel senso che è proprio quando si entra nel DO che I assume il valore iniziale (1 nel nostro caso): così se nell'esempio sopra vi fosse poi un altro DO che utilizza lo stesso indice I (ad es. su altri due vettori B e C di dimensione 20):

```
      . . .
      . . .
      . . .
      TLT = 0.
      TOL = 0.
      TOT = 0.
      DO 5 I = 1,100
    5 TOT = TOT + A(I)
      TOT = TOT/100
      DO 6 I = 1,20
      TOL = TOL + B(I)
    6 TLT = TLT + C(I)
      . . .
      . . .
      . . .
```

esso verrebbe normalmente eseguito, totalizzando B in TOL e C in TLT.

Quello dell'inizializzazione dell'indice del DO e del suo incremento è un punto assai importante: si tenga sempre presente che l'indice viene inizializzato (al valore di h) all'entrata nel DO, alla fine delle istruzioni del rango del DO esso viene incrementato (del valore di i) ed a questo punto viene eseguito il confronto col valore massimo (k).

Assai spesso il rango del DO è chiuso da un'istruzione particolare:

m CONTINUE

dove: m è la label di chiusura del rango del DO e
CONTINUE sta ad indicare che non va intrapresa alcuna operazione (che non sia quella di proseguimento del ciclo o della sua conclusione per aver raggiunto il valore massimo dell'indice).

Si può giustificare l'esistenza di una tale istruzione ricordando che il ciclo di DO non può terminare con un'istruzione di controllo (IF aritmetico, GO TO, altro DO), per cui, in caso di una tale necessità, il ciclo dovrà essere fatto terminare esplicitamente su una successiva istruzione CONTINUE.

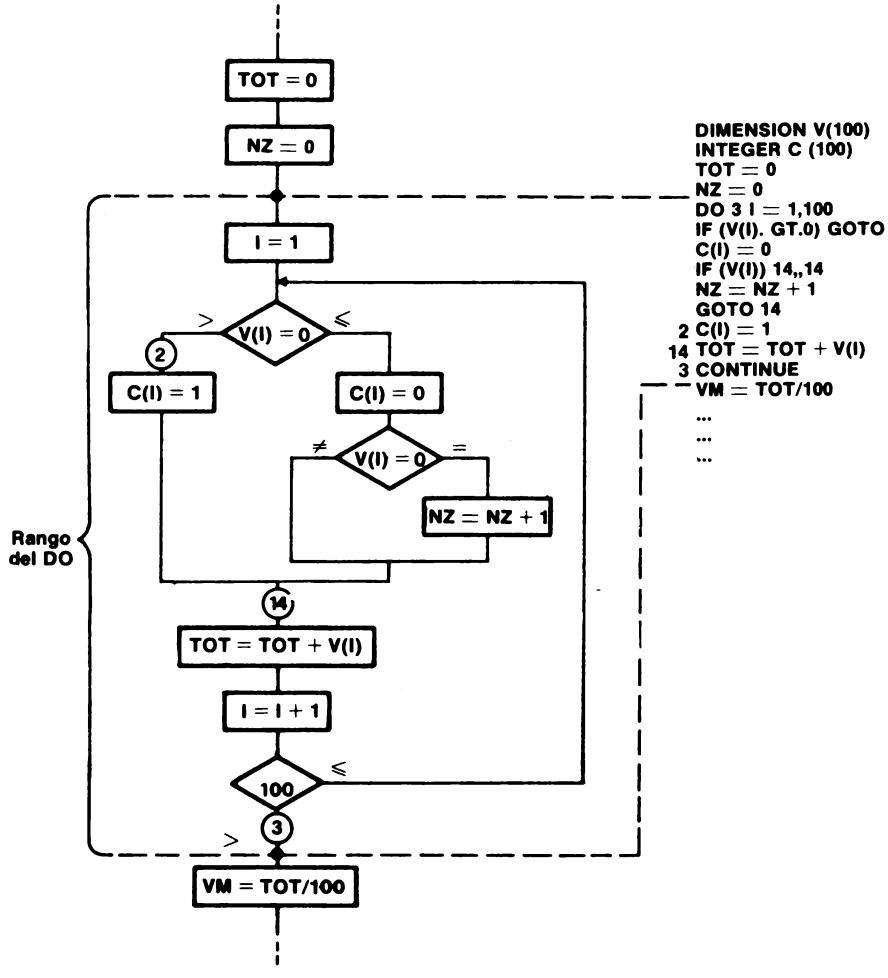
Comunque essa ha, spesso, un valore documentativo e facilita la lettura del programma chiarendo la struttura del DO: in questo senso la sua utilità è tale che, negli esempi che seguono, preferiremo servircene in ogni caso.

## 2.8 Esempi

*Esempio 1* - Dato un vettore V(100) confrontarne ogni componente con 0 ponendo in un vettore C(100) ogni corrispondente elemento a 0 per valori negativi o nulli di V e ad 1 per valori positivi; totalizzare, inoltre, gli elementi di V in TOT e contare, utilizzando un contatore NZ, gli elementi nulli di V; calcolare, infine, il valor medio della serie di valori di V.

Di questo esempio riportiamo nella pagina seguente sia i flow che la codifica allo scopo non solo di vedere in dettaglio come si passi da questa a quella, ma anche di verificare la utilità e brevità del DO.

E' bene fare almeno due osservazioni: nel flow, ovviamente, non compaiono le istruzioni non eseguibili quali il dimensionamento e la definizione di tipo che sono, invece, obbligatorie nel programma; inoltre è importante notare che si deve provvedere all'azzeramento iniziale dei contatori e totalizzatori (è bene sottolineare che ciò vale in generale) in quanto nulla ci assicura che al momento della prima somma essi non siano, come si dice abitualmente, *sporchi*, cioè contenenti dei valori qualunque.




*Esempio 2* - Dato un vettore P(50) calcolare il rapporto tra ogni elemento di posto pari e di posto dispari memorizzando il risultato in un vettore R(25); totalizzare gli elementi di quest'ultimo.

```
      . . .
      . . .
      . . .
      DIMENSION P(50), R(25)
      TOT=0.
      DO 6 K=2,50,2
      R(K/2)=P(K)/P(K—1)
      TOT=TOT+R(K/2)
    6 CONTINUE
      . . .
      . . .
      . . .
```

*Esempio 3* - Dato un vettore Q(50) calcolare la somma SP degli elementi positivi e quella SN degli elementi negativi.

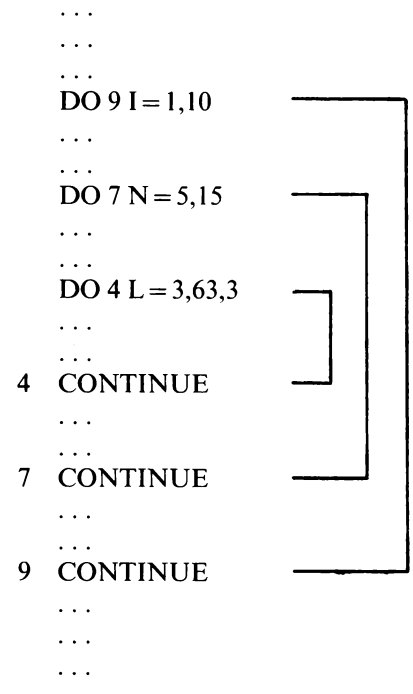
```
      . . .
      . . .
      . . .
      DIMENSION Q(50)
      SP=0.
      SN=0.
      DO 4 I=1,50
      IF(Q(I).LE.0) SN=SN+Q(I)
    4 IF(Q(I).GT.0) SP=SP+Q(I)
      . . .
      . . .
      . . .
```

Se avessimo preferito l'uso dell'IF aritmetico, non si sarebbe potuto terminare il ciclo sull'istruzione di controllo ed avremmo dovuto utilizzare la CONTINUE.

```
      . . .
      . . .
      DIMENSION Q(50)
      SP=0.
      SN=0.
      DO 4 I=1,50
      IF(Q(I))4,3
      SN=SN+Q(I)
      GO TO 4
    3 SP=SP+Q(I)
    4 CONTINUE
      . . .
      . . .
      . . .
```

## 2.9 DO nidificati

In molte occasioni può rendersi necessario eseguire più cicli uno dentro l'altro: si ricorre, in tal caso, alla codifica di più DO **nidificati**.




Nell'esempio di cui sopra, abbiamo previsto tre cicli di DO uno dentro l'altro (ma se ne possono fare ben di più, senza alcuna limitazione): l'esecuzione segue le normali regole del DO e, nel nostro caso si ha che I viene posto ad 1, N a 5, L viene fatto variare da 3 a 63 (con incrementi di 3) e solo a questo punto N viene incrementato a 6, viene ripetuto il ciclo su L, nuovamente incrementato N e così via fino a quando quest'ultimo supera il valore 15, finalmente, viene ora incrementato I, i vari cicli su N ed L vengono ripetuti... etc. etc.

Come si vede, la cosa cui va fatta più attenzione è l'uso degli indici che, ovviamente, non possono essere contemporaneamente utilizzati in un DO ed in un altro interno a questo. Tuttavia, ad un indice di DO di livello gerarchico

superiore si può sempre fare riferimento; ad esempio nulla vieta casi come questo:

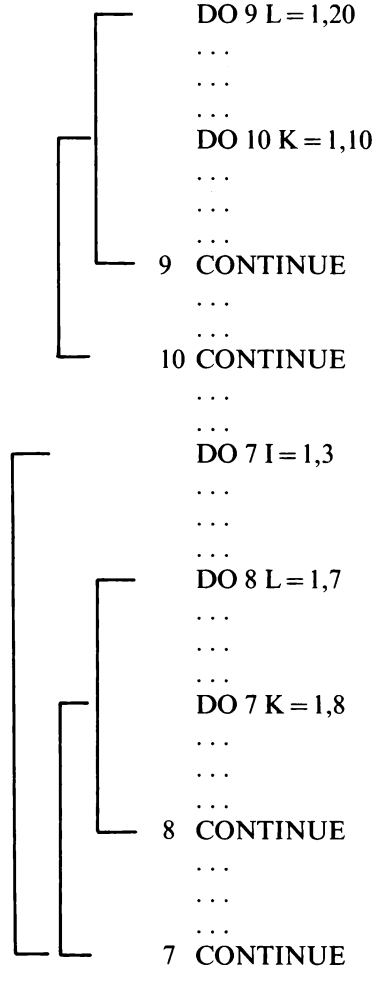
```
      . . .
      . . .
      . . .
      DO 4 IND = 1,100
      . . .
      . . .
      DO 3 KIX = IND,IND + 10
      . . .
      . . .
      X = IND + IND/2*(...)
      . . .
      . . .
3     CONTINUE
4     CONTINUE
      . . .
      . . .
```

Spesso i vari DO terminano sulla stessa istruzione: nell'esempio sopra tra le due CONTINUE di label 3 e 4, infatti, non vi sono istruzioni, ciò significa che avremmo tranquillamente potuto chiudere con un'unica CONTINUE senza che cambiasse assolutamente nulla:

```
      . . .
      . . .
      DO 5 IND = 1,100
      . . .
      . . .
      DO 5 KIX = IND,IND + 10
      . . .
      . . .
      X = IND + IND/2*(...)
      . . .
      . . .
5     CONTINUE
      . . .
      . . .
```

Nel caso in cui i cicli di DO vengano chiusi su istruzioni diverse, occorre prestare attenzione che non si generino delle nidificazioni errate.

E' infatti insito nel concetto di nidificazione quello di completa inclusione delle istruzioni del DO più interno. Così saranno vietate strutture tipo le seguenti:




Terminiamo, anche questa volta, con un breve esempio, ricordando che sempre, quando si vogliano scorrere delle matrici con 2 o più livelli di indice, sarà necessario ricorrere a dei DO nidificati.

*Esempio* - Data la matrice A(10,10) totalizzare ogni suo elemento, costruire inoltre la matrice B(10,10) che abbia ad 1 gli elementi della diagonale principale, a 0 quelli della metà inferiore, uguali a quelli di A nella metà superiore.

$$A = \begin{bmatrix} a_{1,1} & a_{1,2} & ..a_{1,10} \\ a_{2,1} & a_{2,2} & ..a_{2,10} \\ \cdots & \cdots & \cdots \\ a_{10,1} & a_{10,2} & a_{10,10} \end{bmatrix}$$

$$B = \begin{bmatrix} 1 & a_{1,2} & a_{1,10} \\ 0 & 1 & a_{2,10} \\ \cdots & \cdots & \cdots \\ 0 & 0 & ..1 \end{bmatrix}$$

```
      DIMENSION A(10,10),B(10,10)
      TOT=0
      . . .
      . . .
      . . .
      DO 11 I=1,10
      B(I,I)=1
      DO 11 J=1,10
      TOT=TOT+A(I,J)
      IF(I.GT.J) B(I,J)=0
      IF(I.LT.J)B(I,J)=A(I,J)
   11 CONTINUE
      . . .
      . . .
      . . .
```

## 2.10 Trasferimenti permessi nel DO - DO esteso

Prima di concludere i discorsi sul DO vanno ancora fatte alcune osservazioni sui trasferimenti di controllo (GO TO) dall'interno all'esterno di un DO e viceversa. In generale va ricordato che:

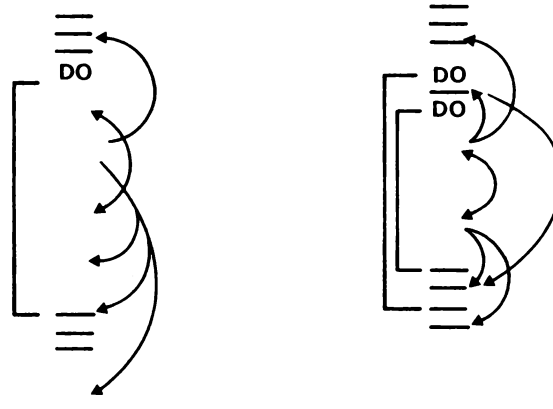
1) Si può sempre eseguire un trasferimento:
   a) dall'interno all'interno del ciclo di DO,
   b) dall'interno all'esterno del ciclo di DO,

2) non si può mai trasferire il controllo dall'esterno verso l'interno.

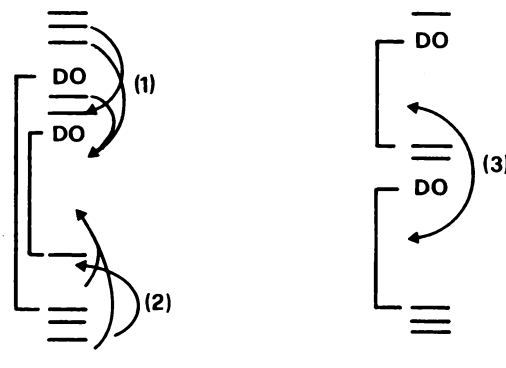
Vedremo tra breve una eccezione a questo secondo punto, per ora tuttavia teniamo valida la regola generale che ha un suo preciso motivo: entrare in un punto intermedio di un ciclo di DO significherebbe saltare l'inizializzazione del DO stesso (in particolare l'assegnazione del valore iniziale dell'indice di ciclo, ma anche la memorizzazione dell'incremento, del valore massimo dell'indice e della istruzione finale del ciclo).

Riassumiamo in uno schema quanto più semplice possibile i trasferimenti permessi e quelli non permessi.

*TRASFERIMENTI PERMESSI*

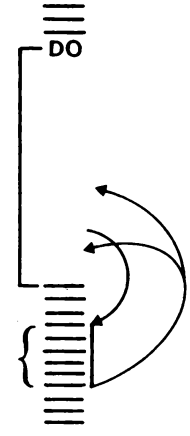



*TRASFERIMENTI NON PERMESSI*




I trasferimenti del gruppo (1) saltano l'inizializzazione del ciclo di DO, quelli del gruppo (2) sono effettuati con un indice o di valore espressamente riassegnato, o non definito, ma il ciclo non è stato, comunque, inizializzato; quelli del gruppo (3) si riconducono rispettivamente al caso (1) se dall'alto in basso, al caso (2) dal basso all'alto.

Abbiamo parlato poco fa di un'eccezione: esaminando il caso (2) ci rendiamo conto che l'unico impedimento a quei trasferimenti è la mancata inizializzazione del ciclo. Qualora si fosse, in qualche modo, provveduto a tale inizializzazione, nulla avrebbe impedito simili rientri. Su questo principio si basa il **DO esteso**: una volta eseguita l'inizializzazione del DO, è possibile uscire dal ciclo per rientrarvi successivamente.




Ad esempio:

```
      .....
      .....
      DO 6 K = 1,NOB
5     .....
      IF(A.EQ.B) GO TO 7
      .....
6     CONTINUE
      .....
7     A = A + ....
      .....
      GO TO 5
      .....
      .....
```

## 2.11 Le istruzioni PAUSE STOP END

Un'istruzione poco usata, tuttavia utile negli elaboratori più piccoli, è l'istruzione PAUSE. Essa ha il compito di arrestare l'esecuzione del programma; questa verrà ripresa solo ad un esplicito comando dato da console. Richiedendo dunque l'intervento dell'operatore essa è usata, come si diceva, solo nei sistemi di piccole dimensioni, dove la figura dell'operatore e quella del programmatore solitamente coincidono.

Poichè può risultare utile riconoscere il punto del programma in cui si è richiesta l'interruzione, questa istruzione prevede il formato

PAUSE n

dove n è un numero intero senza segno: esso comparirà alla console al momento dell'esecuzione dell'istruzione stessa.

Assai più importante è l'istruzione STOP: essa causa l'arresto finale del programma e va posta dunque alla sua fine logica. In uno stesso programma vi possono essere più STOP, nel qual caso verrà eseguito uno solo di questi, (ovviamente il primo incontrato seguendo il flusso di esecuzione).

E' possibile il formato

STOP n

dove n è un intero senza segno che, al momento dell'arresto, verrà inviato in stampa (o, a seconda dei sistemi, a console) per riconoscere quale sia lo STOP eseguito fra i vari possibili.

Infine l'istruzione END: essa non è una istruzione eseguibile: va vista piuttosto come una direttiva al compilatore per segnare la fine fisica del programma. Essa è dunque obbligatoria, deve essere unica e deve comparire alla fine di ogni programma.

```
      .....
      .....
      IF (A.GT.B) GO TO 10
      .....
      IF (A.GT.C) GO TO 10
      .....
      IF (X.LE.Y) STOP
      .....
      GO TO 11
      .....
10    STOP
      .....
11    IF (Z) 12,13,12
      .....
12    STOP
13    .....
      .....
      .....
      STOP
      END
```

CAPITOLO 3

# L'INPUT - OUTPUT

## 3.1 I comandi di input-output

Nelle considerazioni svolte sin'ora, abbiamo sempre ritenuto le informazioni da trattare come già presenti in memoria centrale.

In effetti, però, tale situazione non rispecchia che raramente quanto accade nei casi più usuali: molto spesso infatti dovremo trattare informazioni che ci provengono dall'esterno (input all'elaboratore) e, senz'altro, dovremo riportare i risultati delle nostre elaborazioni su qualche supporto esterno (output).

Per poter ottenere queste funzioni il FORTRAN utilizza alcuni comandi (i cosiddetti comandi di input-output) che esamineremo accuratamente in questo capitolo.

I comandi che considereremo ora sono quelli di input-output più tradizionali, cioè quelli che prevedono come input il lettore di schede e come output la stampante, o comunque quella che abbiamo chiamato "riga messaggio". Per quanto riguarda gli altri dispositivi di I/O frequentemente usati (e cioè i nastri ed i dischi) torneremo su di essi nel prossimo capitolo.

Riprendiamo brevemente alcune nozioni generali sull'I/O: sappiamo che le informazioni presenti su un supporto esterno possono essere trasportate in memoria centrale ed ivi elaborate secondo le nostre richieste: perchè l'elaborazione sia realizzabile, occorre che ogni campo da trattare sia individuabile tramite un nome univoco. In modo del tutto corrispondente le informazioni da riportare su un supporto esterno saranno precedentemente presenti in memoria centrale e da qui trasferite su un ben definito dispositivo di output. Sarà quindi necessario che i comandi di I/O permettano di svolgere un intero complesso di operazioni, e precisamente: indicare il tipo di funzione richiesta, individuare l'unità esterna a cui si vuole accedere, stabilire i campi che si vorranno utilizzare e specificare il nome con cui sono trattati nel programma, infine precisare secondo quale codice dovranno essere immagazzinati od interpretati i dati in memoria centrale.

Vediamo allora come effettivamente i comandi di I/O soddisfino quanto indicato.

### 3.2 Il verbo READ

Il verbo READ serve per *leggere*, cioè rendere disponibile in memoria centrale i dati presenti su un supporto esterno.

Il suo formato è

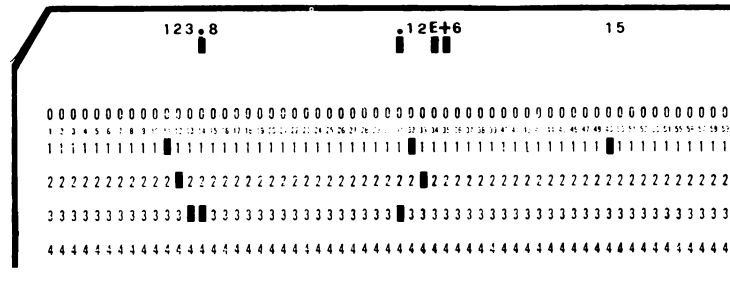
READ (n,m) *lista*

dove: n è un numero intero senza segno che rappresenta il cosiddetto **numero logico** dell'unità periferica da cui si vogliono leggere i dati. Per quanto riguarda il lettore di schede a cui facciamo riferimento in questa prima fase, il numero logico in questione è il 5. Esisterà poi qualche metodo, dipendente dal sistema operativo usato, per associare a tale numero logico l'effettivo file su schede che deve essere presente nel lettore.

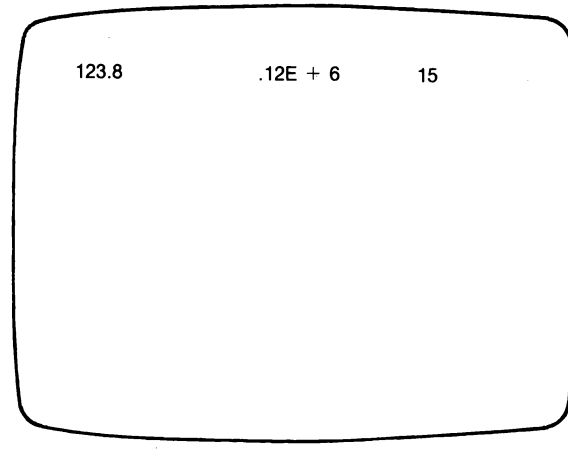
m è un numero intero senza segno che individua il contenuto del campo label del cosiddetto enunciato FORMAT o, più semplicemente FORMAT, in cui, come vedremo in seguito, si metteranno le informazioni che permettono di precisare, tra l'altro, i codici di conversione secondo i quali dovranno essere memorizzati i dati letti.

*lista* è la successione dei nomi dei campi in cui dovranno essere ordinatamente memorizzati i dati letti, secondo le modalità precisate dall'associato enunciato FORMAT. La lista qui indicata, viene anche detta **lista interna**, in contrapposizione alla cosiddetta **lista esterna** presente nell'enunciato FORMAT e che riguarda la lista dei codici di conversione ivi contenuta, come vedremo in dettaglio più appresso. I nomi dei campi dovranno essere scritti consecutivamente e separando l'uno dall'altro tramite una virgola. Dopo l'ultimo nome di campo non dovrà essere indicato alcun simbolo di punteggiatura.

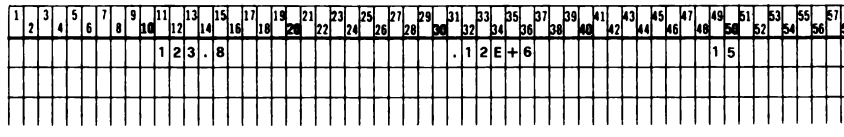
Si voglia, ad esempio, leggere da una scheda l'insieme dei campi indicati in figura, immagazzinando le informazioni nelle aree di memoria centrale associate rispettivamente ai nomi A, B, N.



Ovviamente la "riga messaggio" di cui si è parlato nel paragrafo 1.8 della parte I, a proposito del terminale, è costituita da quanto compare in alto sulla scheda: su di esso verrebbe visualizzata come segue:




Poichè sul video non sono facilmente conteggiabili gli spazi vuoti, d'ora in avanti noi preferiremo presentare queste "righe messaggio" nel modo seguente:




Si ponga attenzione al fatto che quanto detto può valere anche per l'output. Sia che tale output sia su video di terminale o su carta (di stampante), l'importante sarà sempre che vengano esaminate le "posizioni" dei "caratteri di output". Così nelle figure che seguono, indipendentemente dal fatto che vengano usati i simboli di "riga messaggio" su video o su carta della stampante, ciò che sarà importante, sarà notare le "posizioni" dei caratteri.

Il comando di lettura assumerà la seguente forma:

```
READ (5,100) A,B,N
100 FORMAT (.....)
```

e nell'enunciato FORMAT saranno contenute le informazioni necessarie a far sì che il primo numero venga immagazzinato nella cella di memoria A come numero reale, il secondo numero (sempre reale) sia immagazzinato nella cella B, mentre il terzo (questa volta come numero intero) sia immagazzinato nella cella N.

E se volessimo immagazzinare i dati in un array, come dovremmo scrivere? Una risposta banale potrebbe essere quella di scrivere nella lista interna tutti gli elementi dell'array, ma chiaramente non è un metodo molto pratico. Effettivamente esistono dei metodi più utili per ottenere tale risultato.

Supponiamo per esempio di avere un vettore A di 100 elementi ed una matrice B di 10 righe e 20 colonne. All'inizio della codifica avremo la dichiarazione di dimensionamento degli array:

```
DIMENSION A(100), B(10,20)
```

Pensiamo che i dati necessari per memorizzare l'array A e l'array B si trovino su un pacco di schede, che dovrà quindi essere letto.

Con un ordine di lettura intendiamo memorizzare in A i suoi dati e con un altro ordine quelli di B.

Per ottenere quanto ci interessa possiamo usare il cosiddetto **DO implicito**: si scrive cioè nella lista interna il nome dell'elemento dell'array, indicando il campo di variazione dell'indice (o degli indici). Per quanto riguarda il vettore A, potremo scrivere

```
      READ (5,100) (A(I),I = 1,100)
100 FORMAT (....)
```

dove con tale scrittura si intende che nel caricamento si partirà dall'elemento di indice 1, poi si passerà a quello di indice 2 e così via sino a quello di indice 100. E' bene notare che, per indicare il DO implicito, occorre racchiudere i riferimenti inerenti a tale DO entro parentesi. Come si vede, nel caso di vettori, cioè di array monodimensionali, non c'è alcun problema.

Un po' più complessa si presenta la situazione per quanto riguarda la matrice B. Infatti, procedendo in modo analogo a quanto visto prima (a parte una semplice estrapolazione alle due dimensioni), quale dei due modi di scrivere sarà giusto?

```
      READ (5,200) ((B(I,K),K = 1,20),I = 1,10)
200 FORMAT (....)
```

oppure:

```
      READ (5,200) ((B(I,K),I = 1,10),K = 1,20)
200 FORMAT (....)
```

In effetti sono giusti tutti e due, solo che i risultati del caricamento sono piuttosto diversi. Infatti nei DO impliciti ora indicati, il ciclo che viene

eseguito per primo è quello relativo alle parentesi più interne, così nel primo ordine di READ verrebbero caricati gli elementi:

```
B(1,1) B(1,2)  ..... B(1,20)
B(2,1) B(2,2)  ..... B(2,20)
.....
B(10,1) B(10,2) ..... B(10,20)
```

Gli elementi verrebbero cioè caricati *per righe* (immaginando la matrice come un'usuale tabella bidimensionale), mentre nel secondo caso verrebbero caricati nell'ordine gli elementi:

```
B(1,1) B(2,1)  ..... B(10,1)
B(1,2) B(2,2)  ..... B(10,2)
.....
B(1,20) B(2,20) ..... B(10,20)
```

Gli elementi sarebbero cioè caricati *per colonne*.

Quale sarà allora l'ordine da usare? Chiaramente dipende dall'ordine in cui sono memorizzati i dati da caricare in input, quindi se si hanno in input i dati consecutivi che costituiscono le colonne, occorrerà usare il secondo metodo, in caso contrario si userà il primo.

A proposito della lettura e del caricamento degli array vi è ancora qualcosa da dire: in FORTRAN è possibile fare compiere queste operazioni in modo, per così dire, automatico, cioè precisando nella lista interna il solo nome dell'array, senza altre specificazioni. Questa possibilità è facilmente comprensibile nel caso degli array monodimensionali, dato che la modalità di caricamento è unica, ma come si può pensare che sia possibile nel caso di array a due o più dimensioni che, come abbiamo visto, possono essere caricati in diversi modi? La risposta è, tutto sommato, piuttosto semplice: viene seguito un criterio fisso e quindi univoco. Questo consiste, immaginando il DO implicito, nel far variare a partire dal ciclo più interno a quello più esterno rispettivamente il primo, il secondo, ... l'n-esimo indice (dove, ricordiamolo ancora, si può arrivare sino a 3 o 7 indici come massimo a seconda dei compilatori).

Così, avendo:

```
DIMENSION C(10,20,30)
```

la scrittura

```
READ (5,1) C
1 FORMAT (...)
```

mettendo in chiaro il DO implicito equivale a:

```
      READ(5,1) (((C(I,K,L),I = 1,10),K = 1,20),L = 1,30)
    1 FORMAT (....)
```

In tal modo l'array viene letto e memorizzato per *pagine* e, all'interno di ogni pagina, per *colonne*.

Se volessimo leggerlo e memorizzarlo per pagine, ma all'interno di ogni pagina, memorizzarlo per righe, dovremmo esplicitamente usare il DO implicito:

```
      READ(5,1) (((C(I,K,L,),K = 1,20),I = 1,10),L = 1,30)
    1 FORMAT (....)
```

Quanto abbiamo visto sinora ci permette di leggere dei dati dall'input qualora si sappia esattamente quanti questi siano.

Può accadere però che in un'elaborazione si debba leggere un numero di dati non noto a priori, cioè leggere ed elaborare i dati di un file. Durante l'elaborazione, in generale, si tornerà a ciclare sul comando di lettura sino al termine dei dati di input.
Come sarà allora possibile accorgersi di questo termine, senza che vada in errore il programma che cerca di leggere un file già terminato?

In tempi passati si cercava di aggirare l'ostacolo inserendo come dato finale un valore non esistente tra i dati da elaborare, facendo un controllo subito dopo la lettura, per assicurarsi di non aver letto quel valore: nell'un caso si procedeva nella normale esecuzione di quanto previsto per l'input, mentre nell'altro si terminava l'elaborazione.

Così, dovendo elaborare un insieme di dati reali presenti in input, la situazione poteva essere:

```
    1 READ(5,10) DATO
   10 FORMAT (.....)
      IF (DATO.EQ.999999) GO TO 1000
      . . .
      . . .
      . . .
      . . .
      . . .
      GO TO 1
 1000 STOP
      END
```

Attualmente, invece, praticamente tutti i compilatori permettono di gestire, durante la lettura, la cosiddetta **fine del file**. Questa non consiste in altro che in un record con dei caratteri particolari che vengono riconosciuti dal sistema come indicanti la fine del file in cui si trova tale record. Per poter sfruttare questa possibilità, il comando di lettura è stato leggermente modificato e precisamente esso ha la forma:

READ(n,m,END = 1) lista

dove n,m, lista hanno lo stesso significato visto prima, mentre 1 è il numero di una label di un'istruzione a cui si dovrà saltare, una volta riconosciuta la fine del file.

Può accadere però che per qualche ragione il comando di lettura non possa essere portato a buon fine. In tal caso si ha di solito un errore di programma che provoca la sua interruzione. E' attualmente possibile, con la generalità dei compilatori, gestire anche questa eventualità. E' infatti disponibile un'istruzione di lettura che prevede

READ(n,m,END = 1,ERR = k) lista

dove n,m,1, lista hanno il significato visti prima, mentre k è il numero di una label di un'istruzione a cui si dovrà saltare nel caso venga riscontrato un qualsiasi errore in fase di lettura.

Le clausole END ed ERR non devono necessariamente coesistere.

Così l'esempio visto prima potrà essere scritto più semplicemente:

```
    1  READ(5,10,END = 1000,ERR = 2000) DATO
   10  FORMAT(....)
       . . .
       . . .
       . . .
       . . .
       . . .
       GO TO 1
 2000  . . .
       . . .
       . . .
       . . .
```

```
1000 ...
     ...
     ...
     ...
     STOP
     END
```

senza dover più introdurre dei dati fittizi con cui, a volte, si può correre qualche rischio, specie nel caso in cui, a nostra insaputa, quel dato sia invece tra quelli da elaborare.

### 3.3 Il verbo WRITE

Questo verbo serve a svolgere la funzione *duale* rispetto al precedente: esso ci permette infatti di *scrivere* su un supporto esterno i dati presenti in memoria centrale. Il suo formato è:

WRITE(n,m) *lista*

dove n, m, *lista* hanno lo stesso significato visto per il verbo READ, solo che n, questa volta, assume generalmente il valore di 6 per la stampante e *lista* può anche mancare nel caso in cui tutte le informazioni da portare in output siano contenute direttamente nell'associato enunciato FORMAT.

Se noi volessimo quindi scrivere il contenuto delle celle di memoria A, B, N, non dovremmo fare altro che usare il comando:

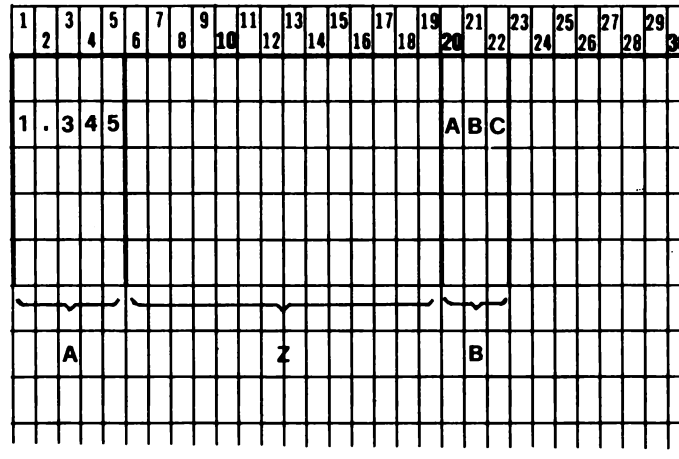
```
     WRITE(6,200) A,B,N
200 FORMAT(....)
```

dove l'enunciato FORMAT conterrà i codici appropriati che permettono di convertire in modo esatto le informazioni presenti nelle varie celle di memoria centrale, così che il risultato sia stampatibile in modo corretto.

### 3.4 FORMAT

Come debba avvenire l'input o l'output viene definito dall'enunciato FORMAT. Esso serve a descrivere il **record logico** che sarà, in generale, composto da dati (informazioni) di tipo diverso: questa istruzione ci permette di definire il **tipo** di dato e la sua lunghezza all'interno del record logico stesso.

Per quanto riguarda la lunghezza è sufficiente tenere presente che essa è determinata, in generale, dal numero di caratteri che intendiamo prendere in considerazione: per esempio si abbia un input come figura:




Vi abbiamo indicato i due campi A e B corrispondenti a due valori (1.345 e ABC) che, poniamo, vogliamo comunicare in input al sistema.

L'informazione è contenuta tra colonna 1 e colonna 22 e dunque questa è la dimensione totale che dobbiamo specificare, ma allo stesso tempo dobbiamo anche precisare che A è il dato che si trova tra colonna 1 e 5, B tra la 20 e la 22 e che dobbiamo saltare il campo Z (colonne tra 6 e 19): per fare ciò nell'istruzione FORMAT specificheremo le lunghezze dei tre campi. Per quanto riguarda il tipo di informazione teniamo presente che, nell'esempio proposto, il primo dato è un numero razionale, il campo Z corrisponde a caratteri da non prendere in considerazione ed infine il campo B corrisponde a caratteri alfabetici.

L'istruzione che descrive quanto vogliamo è:

```
100  FORMAT(F5.3,14X,A3)
```

dove:

100 è la label;
F5.3 specifica che il primo campo è un numero reale di 5 caratteri di cui 3 decimali;
14X specifica che vi sono quattordici posizioni da saltare
A3 specifica che il terzo campo è alfabetico di tre caratteri

Naturalmente prima di comprendere appieno come abbiamo composto tale istruzione è necessario ricordare quali sono i tipi di dati che possono venire trattati dal FORTRAN: cominciamo dunque con l'esame di questi e del simbolo che in un'istruzione di FORMAT li caratterizza.

| Tipo di dato o funzione | Simbolo del FORMAT |
|---|---|
| Per ignorare delle posizioni | X |
| Numeri interi | I |
| Numeri reali | F |
| Numeri reali in notazione esponenziale (Floating point) | E |
| Numeri reali in doppia precisione in notazione esponenziale (floating point double precision) | D |
| Caratteri e stringhe in output | H |
| Variabili di caratteri | A |
| Variabili logiche | L |
| Fattore di scala | P |
| Stampa in funzione del risultato | G |
| Tabulazione | T |

Il FORMAT è composto da una specifica di **tipo** e da una specifica di **lunghezza** alle quali, entrambe, si deve porre attenzione a seconda che si tratti di un *input* o di un *output*.

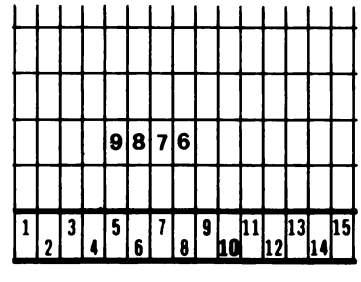
Per capire bene l'utilizzo di tale istruzione analizzeremo i vari casi della tabella ora vista riferendoci, sempre, ad esempi specifici.

### 3.5 Il FORMAT X

**nX** specifica un salto di n spazi.

In input vengono saltati (non letti) n caratteri, in output vengono lasciate n posizioni a blank.

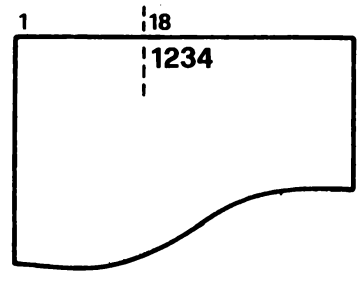
Così, se il dato da leggere in input fosse:



nel programma, per indicare di saltare le prime 4 colonne scriveremmo:

```
      READ(5,100)
  100 FORMAT(4X,..)
```

Se la stampa da ottenere fosse del tipo:




per indicare di iniziare la stampa dalla 18-esima colonna scriveremmo:

```
      WRITE(5,200)
  200 FORMAT(18X,..)
```

Da notare che (come si vedrà meglio più avanti) quando il primo carattere di una riga di stampa viene posto a blank, si intende che la riga stessa deve venire scritta ad interlinea semplice dalla precedente (cioè a capo).

### 3.6 Il FORMAT I

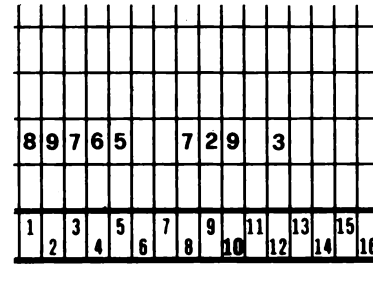
**In** specifica un numero intero di n cifre tanto in input che in output.

Così:

**Il programma**

```
1       7
|       |
|       |
|       |
|       |  READ(5,100)N
|       |
|       |
| 100   |  FORMAT (I3)
|       |
|       |
```

**Dato da leggere**



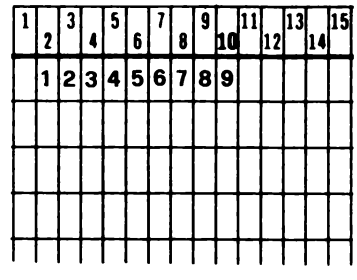
specifica che va letta la variabile N di tre cifre alla quale verrà assegnato il valore 897.

| **Se in memoria** | **il programma** |
|---|---|
| K = 12 | 1 7<br>WRITE(6,1)K<br>1 FORMAT (1X,I2) |

specifica che venga posto in output il numero 12.

Ci sono però alcune particolarità da tenere presenti e per le quali è bene porre attenzione ai seguenti esempi:

se in input avessimo:



e l'istruzione di lettura fosse:

READ (5,90) I

| con i FORMAT | otterremmo in memoria I |
|---|---|
| 90 FORMAT (I1) | 0 |
| 12 | 1 |
| 13 | 12 |
| ... | ... |
| I10 | 123456789 |

se l'istruzione di lettura fosse:

READ (5,99) I,M,K

con i FORMAT

| | otterremmo in memoria | | |
|---|---|---|---|
| | I | M | K |
| 99 FORMAT (I1,I2,I1) | 0 | 12 | 3 |
| (I5,I1,I2) | 1234 | 5 | 67 |
| (I1,I7,I1) | 0 | 1234567 | 8 |
| (I8,I1,I3) | 1234567 | 8 | 900 |
| (I10,I2,I2) | 123456789 | 0 | 0 |

poniamo attenzione al fatto che i blank vengono trattati come zeri significativi quando si trovano a destra delle cifre; a sinistra, invece, sono ritenuti zeri non significativi.

Se in memoria avessimo i valori (da porre in output):

I = 1 J = 20 K = 3150 L = 5 M = —789 N = —9800

e l'istruzione di scrittura fosse:

WRITE(6,100) I

```
con i FORMAT                                   otterremmo in stampa
col. 1        7                                      col. 1
              .                                        .
              .                                        .
              .                                        .
(1)      100  FORMAT (1X,I1)                          1
              .                                        .
              .                                        .
              .          (1X,I3)                             1
```

dunque con accostamento a destra del valore ed eventuale riempimento di blank a sinistra.

Se l'istruzione di scrittura fosse:

```
col.          7
              .
              .
              .
              WRITE (6,8) I,J,K
```

```
con i FORMAT                          otterremmo la stampa

FORMAT (1X,I1,2X,I2,2X,I4)        1     20   3150
FORMAT (1X,I3,I3,I5)                1 20   3150
FORMAT (8X,2I5,I6)                                1     20    3150
FORMAT (1X,3I5)                     1     20   3150
```

cioè si può premettere il numero di volte che si intende ripetere quel codice di conversione nel FORMAT, anzi si può anche scrivere:

```
FORMAT (2X,3(I4,1X))              1     20 3150
FORMAT (3(1X,I6))                  1       20    3150
```

Da notare, ancora, che negli esempi visti non è mai comparso il segno + (si trattava di numeri positivi): quando il numero è negativo il segno viene stampato e di esso va tenuto conto come occupante una posizione del nostro FORMAT; per i numeri positivi il segno viene sempre omesso.

---

(1) Il FORMAT X iniziale sta ad indicare, come si è già accennato, un controllo del carrello di stampa (vedere il relativo paragrafo) cioè una interlinea normale.

Una specifica di formato non bastante a contenere il numero, provoca la stampa di tanti asterischi quante sono le posizioni insufficientemente dichiarate.

Così se, sempre per i valori dell'esempio precedente, l'istruzione di scrittura fosse:

```
WRITE (6,500) K,L,M,N
```

| con i FORMAT | | otterremmo in stampa |
|---|---|---|
| 500 | FORMAT (1X,I4,1X,3I6) | 3150 -5 -789 -9800 |
| | FORMAT (1X,2I2,I6,I5) | **-5 -789-9800 |
| | FORMAT (6X,I4,I2,I3,I5) | 3150-5***-9800 |
| | FORMAT (1X,I2,3(3X,I5)) | ** -5 -789 -9800 |

Infine attenzione che se fosse K = 0 e si avesse WRITE (6,10) K

| con i FORMAT | | si otterrebbe la stampa: |
|---|---|---|
| 10 | FORMAT (1X,I1) | 0 |
| | FORMAT (1X,I6) | 0 |

Facciamo ora un'osservazione che, teniamolo ben presente, riguarda tutte le specifiche di FORMAT che vedremo d'ora innanzi; essa va ritenuta regola generale di una certa importanza: le specifiche di FORMAT possono non esaurire la lista delle variabili dichiarate sul verbo READ o WRITE, ciò nel senso che può non risultare apparente una corrispondenza uno a uno tra specifica e variabile; tuttavia, poichè il sistema prende in considerazione le indicazioni del FORMAT leggendo le parentesi da sinistra a destra e poi ricominciando fino ad esaurimento della lista di variabili, basterà che la corrispondenza esista complessivamente, tenuto conto, cioè, delle possibili ripetizioni delle specifiche. Ciò accade in molti casi, il più frequente riguarda l'utilizzo del DO implicito:

```
DIMENSION M(10)
READ(5,100)  (M(I),I = 1,10)
```

con il FORMAT 100 FORMAT(I3)

vengono letti 10 numeri di tre cifre, uno per scheda. Si era detto infatti che ogni FORMAT riguarda un singolo record; con

```
100 FORMAT(2I3)
```

vengono letti 10 numeri di tre cifre, due per scheda e ivi accostati tra loro; con

```
100 FORMAT(10(I3,1X))
```

vengono letti 10 numeri di tre cifre, tutti su di una sola scheda separati l'uno dall'altro da uno spazio.

Ma anche senza DO implicito:

```
    READ(5,50) I,J,K
50 FORMAT(I5)
```

vengono letti tre numeri (I,J,K nell'ordine) di 5 cifre, uno per scheda.

A questo punto precisiamo una particolarità sull'uso dei fattori di ripetizione nel FORMAT: abbiamo visto che essi sono utilizzabili per scrivere in forma compatta sequenze uguali di codici di conversione. Così FORMAT (1X,4I5,2X,4I5,2X,I3) si può scrivere meglio come FORMAT (1X,2(4I5,2X),I3), portando quindi alla presenza di due blocchi di parentesi innestati l'uno nell'altro. Tale fatto è importante, poiché i fattori di ripetizione permettono di ottenere, in caso di necessità, ulteriori blocchi di parentesi innestate. Però il FORTRAN permette di gestire sino ad un massimo di tre livelli di parentesi, così potremo avere scritture, in forma compatta come:

```
FORMAT(1X,2(2X,2(I3,2X),I4),1X,I5)
```

che equivale, in forma estesa, a:

```
FORMAT(1X,2X,I3,2X,I3,2X,I4,2X,I3,2X,I3,2X,I4,1X,I5).
```

Come si vede, la scrittura compatta è molto più agevole. Il livello delle parentesi viene contato dalla più esterna alla più interna, così nell'esempio precedente avremo:

```
                FORMAT(1X,2(2X,2(I3,2X),I4),1X,I5)
livello                 1    2    3
```

La questione dei livelli di parentesi nel FORMAT è piuttosto delicata: infatti quando le richieste della lista interna non sono esaurite dai codici forniti dalla lista esterna, il FORMAT viene riesaminato. Se però in quest'ultimo sono presenti diversi livelli di parentesi, esso viene riesaminato a partire dalla parentesi di livello (od "ordine")2: occorrerà allora prestare attenzione anche a "come" si scrive il FORMAT ed in particolare a come si utilizzano i fattori di ripetizione.

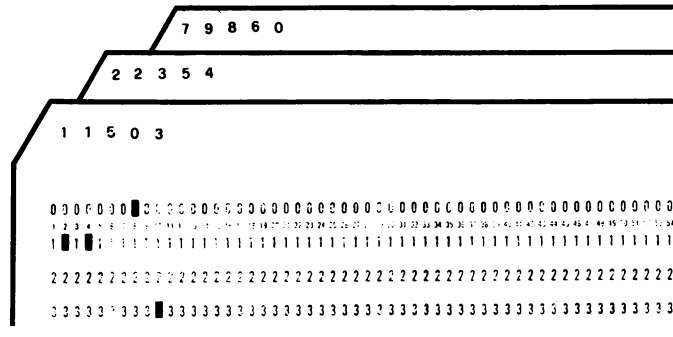
Completiamo ora un esempio visto a proposito del DO implicito: si era detto (vedi pag. 103) che:

READ(5,50) ((M(I,J),I = 1,5),J = 1,3)

era del tutto analogo a:

READ(5,50) M

Se i dati sono scritti su schede nel modo seguente:




il FORMAT dovrà essere:

50 FORMAT(5(1X,I1))

Abbiamo ora in memoria la matrice M di 5 righe e 3 colonne, per cui volendone una stampa che corrisponda proprio a questa disposizione, cioè ad esempio:

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | | | | | | |
| | | | | 1 | | | 2 | | | 7 | | | |
| | | | | 1 | | | 2 | | | 9 | | | |
| | | | | 5 | | | 3 | | | 8 | | | |
| | | | | 0 | | | 5 | | | 6 | | | |
| | | | | 3 | | | 4 | | | 0 | | | |
| | | | | | | | | | | | | | |

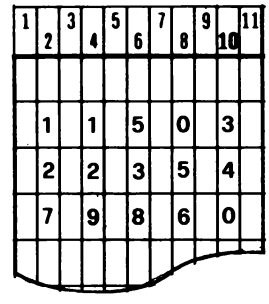
come ben sappiamo dovremo scrivere:

WRITE(6,20)((M(I,J),J = 1,3),I = 1,5)

ed il FORMAT dovrà allora essere:

20 FORMAT(3X,3I3)

volendo invece una stampa del tipo:




dovremo scrivere:

WRITE(6,30)M

oppure:

WRITE(6,30)((M(I,J),I = 1,5),J = 1,3)

con il FORMAT:

30 FORMAT(5(1X,I1))

oppure:

30 FORMAT(5I2)

Per concludere notiamo, ancora, che ogni FORMAT è sempre stato riferito, negli esempi visti, ad una singola scheda o ad una singola riga di stampa. Vedremo come sarà possibile produrre un passaggio al record successivo in una sola istruzione di FORMAT; è bene, tuttavia, almeno per il momento, tenere presente che la descrizione data in questa istruzione è sempre relativa al singolo record logico: ciò eviterà confusioni ed errori banali.

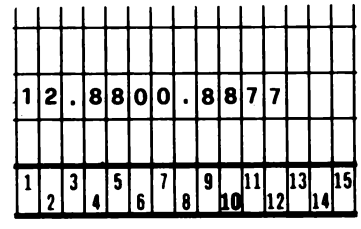
### 3.7 Il FORMAT F

Per i numeri reali in forma razionale (ed arrotondati) si utilizza il FORMAT F.

**Fm.n** specifica trattarsi di un numero di m caratteri di cui n sono cifre decimali (1).

Nello scrivere l'istruzione di FORMAT ci si dovrà comportare secondo le regole già viste per il FORMAT I e che dunque non ripeteremo.

Avendo in input i dati:




e l'istruzione di lettura:

READ(5,100) A,B,C,D

considerando che la presenza del punto è decisiva e determina la posizione effettiva della parte decimale (anche se il FORMAT sembra richiedere un numero diverso di cifre decimali) e che di quelli richiesti dal FORMAT viene tenuto conto solo se il punto non compare, si ha che:

| con i FORMAT | si ottiene in memoria | | | |
|---|---|---|---|---|
| | A | B | C | D |
| 100 FORMAT(F3.2,F3.1,F3.1,F3.2) | 12. | 88.0 | 0.8 | 8.77 |
| 4F3.2 | 12. | 8.80 | 0.8 | 8.77 |
| F5.2,F4.2,F3.1,F6.4 | 12.88 | 0.8 | 87.7 | 0.0 |
| 2F5.2,2F3.1 | 12.88 | 0.88 | 77.0 | 0.0 |
| 4F6.2 | 12.880 | 0.8877 | 0.0 | 0.0 |
| F4.1,F2.1,F3.1,F6.1 | 12.8 | 8.0 | 0.8 | 87700.0 |

(1) La notazione normalmente usata dagli elaboratori è quella anglosassone che utilizza il punto al posto della virgola come separatore della parte decimale di un numero dalla sua parte intera.

Naturalmente il FORMAT può essere composto da specifiche diverse, così, sempre nell'esempio di cui sopra, ma nel caso si sia dichiarato INTEGER C,D

| con i FORMAT | si ottiene in memoria | | | |
|---|---|---|---|---|
| | A | B | C | D |
| 100 FORMAT(1X,F3.1,2X,F3.1,I1,I2) | 2.8 | 0.8 | 8 | 77 |
| (F5.2,F6.2,2I1) | 12.88 | 0.887 | 7 | 0 |
| (8X,F3.1,F3.1,2I2) | 88.7 | 70.0 | 0 | 0 |

In output va tenuto presente che, come per il FORMAT I:

il segno + non viene stampato;

gli zeri non significativi che precedono il numero non vengono stampati, ma vengono aggiunti zeri a destra fino a raggiungere, qualora ve ne fosse bisogno, il numero di cifre decimali richieste;

una dichiarazione di un numero di cifre decimali minore di quelle effettive provoca in stampa un arrotondamento;

un FORMAT insufficiente a contenere il numero provoca la stampa di asterischi.

Così:

| **se in memoria si ha** | **con il FORMAT** | **viene stampato** |
|---|---|---|
| +76.985 | F6.3 | 76.985 |
| +76.985 | F7.3 | 76.985 |
| +76.985 | F7.4 | 76.9850 |
| +76.985 | F5.2 | 76.99 |
| +8.72 | F3.1 | 8.7 |
| +8.77 | F3.1 | 8.8 |
| -0.1234 | F7.4 | -0.1234 |
| -0.1234 | F6.4 | -.1234 |
| +0.1234 | F5.4 | .1234 |
| -0.1234 | F5.4 | ***** |
| +12.988 | F7.4 | 12.9880 |
| +12.988 | F7.2 | 12.99 |
| +12.988 | F3.2 | *** |
| +12.988 | F8.4 | 12.9880 |

## 3.8 Il FORMAT E

Per i numeri reali in notazione esponenziale viene utilizzato il FORMAT E.
Porre un numero in notazione esponenziale significa esprimere quel numero in forma di prodotto tra le sue cifre ed una opportuna potenza di 10. Così il numero:

$$123.4567$$

può essere posto in notazione esponenziale scrivendo ad esempio:

$1.234567 \cdot 10^{+2}$ oppure $12345.67 \cdot 10^{-2}$ oppure $0.0001234567 \cdot 10^{6}$

Diremo che è in forma **normalizzata** nel caso si ponga il punto decimale prima della prima cifra significativa:

$$0.1234567 \cdot 10^{3}$$

Aggiungiamo a tali considerazioni generali il fatto che indicheremo appunto con E l'esponente da dare a 10.

Così in input **Em.n** indica un numero reale di m caratteri (compresi il punto decimale, la lettera E per **esponente** e l'eventuale segno dell'esponente), di cui n sono cifre decimali.

Se n è zero il numero viene letto come è perforato sulla scheda (si vedano i primi degli esempi seguenti), altrimenti definisce, come si è detto, la posizione del punto decimale, ma questo a meno che esso non sia espresso esplicitamente sulla scheda stessa: tale presenza è comunque determinante ed in caso di contrasto prevale su quanto scritto nel FORMAT (si veda il quinto esempio).

| Se il numero da leggere e da portare in memoria è | sulla scheda potrà essere scritto | si potrà, allora, utilizzare il FORMAT |
|---|---|---|
| 8728.0 | 8728E 03 | E10.0 |
| 12.54 | 12.54E 00 | E9.0 |
| $5432.1898 = 54.321898 \cdot 10^{2}$ | 54321898E 02 | E12.6 |
| $6532.1 = 6.5321 \cdot 10^{3}$ | 65321E3 | E7.4 |
| $12.13 = 1.213 \cdot 10$ | 1.213E+01 | E9.2 |
| $1213.14 = 1.21314 \cdot 10^{3}$ | 121314E+03 | E10.5 |
| $0.000000789 = 7.89 \cdot 10^{-7}$ | 7.89E-7 | E7.2 |
| 0.00012345 = 0.12345E-3 | .12345E-3 | E10.0 |
| $0.00000000001 = 1 \cdot 10^{-11}$ | 1E-10 | E5.1 |
| $8265.4 = 8.2654 \cdot 10^{3}$ | 82654E03 | E10.4 |
| " | 82654E+3 | E10.4 |
| " | 82654E3 | E10.4 |
| " | 8265.4 | E10.4 |
| " | 8265.4E00 | E10.4 |
| " | 8.2654E03 | E10.4 |

In output **Em.n** indica che deve essere stampato un numero di m caratteri totali (inclusi eventuali blank, punto decimale, E, segno del numero e segno dell'esponente) in forma normalizzata e con n cifre decimali.

| Così, se in memoria si trova il numero | utilizzando il FORMAT | si ottiene in stampa |
|---|---|---|
| 278.567632 | E16.9 | 0.278567632E 03 |
| " | E10.4 | 0.2786E 03 |
| " | E16.7 | 0.2785676E 03 |
| " | E16.10 | 0.27856763 20E 03 |
| " | E11.5 | 0.27857E 03 |
| -6.543 | E12.5 | -0.65430E 01 |
| " | E8.1 | -0.7E 01 |
| " | E14.5 | -0.65430E 01 |
| 0.21 | E8.2 | 0.21E 00 |
| 0.00017 | E10.3 | 0.170E-03 |
| " | E8.2 | 0.17E-03 |
| $0.0000000000001 = 10^{-13}$ | E7.1 | 0.1E-12 |
| " | E9.3 | 0.100E-12 |
| -0.5678 | E10.3 | -0.568E 00 |
| -0.0000004 | E8.1 | -0.4E-06 |
| " | E12.2 | -0.40E-06 |
| -6.781 | E14.8 | -.67810000E 01 |

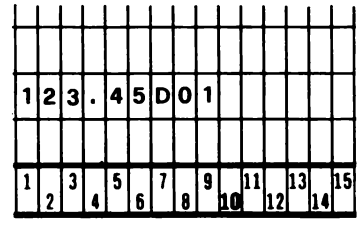
Si tenga inoltre presente che qualora il numero in memoria fosse costituito da più di n decimali ne verrebbero stampati solamente n, ma con l'opportuno arrotondamento.

Infine, per quanto riguarda l'esponente osserveremo che questo viene stampato sempre in due cifre precedute dal simbolo E da un blank per gli esponenti positivi o dal segno meno per i negativi.

### 3.9 Il FORMAT D

Per i numeri reali in doppia precisione espressi in forma esponenziale, viene utilizzato il FORMAT D. Sia in input che in output per tale FORMAT valgono in tutto e per tutto le stesse norme del FORMAT E: l'unica differenza sta nel formato di memorizzazione.

Se in input si ha:




con il FORMAT D9.2 viene posto in memoria, in doppia precisione, il numero 1234.5.

Se, d'altro canto, si ha in memoria il numero 278.567632 in doppia precisione, con un FORMAT D17.10 viene stampato

0.2785676320D 03

### 3.10 Il FORMAT H

Quando si voglia emettere in stampa una stringa di caratteri si utilizza il FORMAT **nH**, dove n è il numero di caratteri della stringa che viene posta subito dopo il simbolo H stesso, senza alcun carattere intermedio. Tale FORMAT verrà utilizzato per intestazioni o frasi di commento da alternare, ad esempio, ai risultati numerici. Così, se un certo risultato fosse il numero 3.21 e volessimo ottenere la scrittura:

RISULTATO DI X = 3.21

dovremmo introdurre la stringa RISULTATO DI X = nel modo seguente:

15HRISULTATO DI X =

accostando poi il FORMAT del numero 3.21 nel modo usuale (si vedano gli esempi più avanti).

Si ottiene lo stesso effetto inserendo la stringa tra apici: così è equivalente scrivere:

'RISULTATO DI X = '

in tal caso gli apici vengono interpretati come delimitatori e non sono stampati. Qualora si renda necessaria la stampa di tali apici, è sufficiente utilizzare il FORMAT H:

24HL'ULTIMO RISULTATO E' Y =

Alcuni compilatori permettono l'uso dei due delimitatori accostati che vengono interpretati come un unico apice appartenente alla stringa; l'ultimo esempio potrebbe allora essere scritto così:

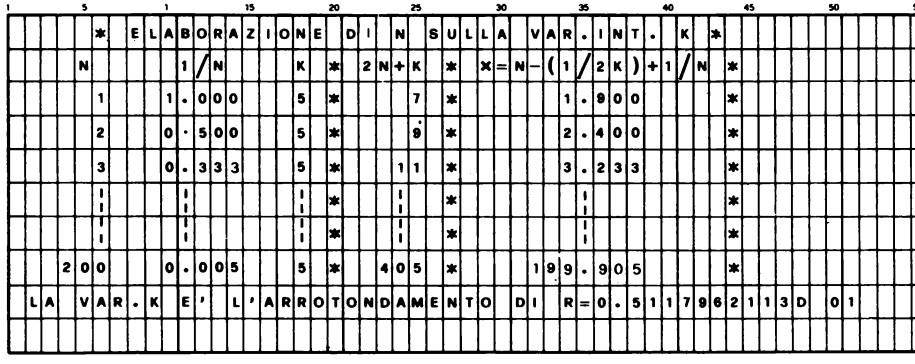
'L''ULTIMO RISULTATO E'' Y = '

### 3.11 Un esempio d'uso dei FORMAT in output

Prima di procedere nelle restanti specifiche di FORMAT, ci pare opportuno analizzare un esempio completo così da comprendere meglio come utilizzare quanto appreso finora.

Si abbia in memoria il vettore N(200) contenente i primi 200 numeri interi positivi, una variabile K = 5 ottenuta come arrotondamento della variabile in doppia precisione R = 0.5117962113·10 e si vogliano calcolare, per ogni componente di N, i valori di Y = 1/N, IZ = 2N + K ed X = N — (1/2K) + 1/N.

Si decida di emettere in stampa i risultati secondo una formattazione della pagina cosiffatta:




la parte di programma riguardante quei calcoli e la stampa potrà, allora, essere:

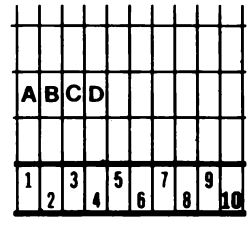
```
      ...
      WRITE(6,100)
100   FORMAT(6X, '* ELABORAZIONE DI N SULLA VAR.INT. K *')
      ...
      ...
      ...
      WRITE(6,150)
150   FORMAT(5X,1HN,5X,3H1/N,4X,'K * 2N+K * X=N—(1/2K)+1/N *')
      DO 6 I = 1,200
      Y = 1/N(I)
      IZ = 2*N(I)+K
      X = N(I)—(1/(2*K))+1/N(I)
      WRITE(6,200) N(I),Y,K,IZ,X
200   FORMAT(4X,I3,3X,F5.3,3X,I1,4H *      ,I3,2H*  ,4X,F7.3,5X,1H*)
  6   CONTINUE
      WRITE(6,300) R
300   FORMAT(36H LA VAR.K E' L'ARROTONDAMENTO DI R=,D16.10)
```

### 3.12 Il FORMAT A

Per leggere o stampare variabili di caratteri si utilizza il FORMAT A.

Poichè, come si è visto trattando della *memoria* esiste una unità minima di indirizzamento che di solito è la parola, occorrerà fare riferimento a questa nel definire il campo da trattare. A seconda del codice di rappresentazione dei caratteri (ASCII, EBCDIC, BCD, FIELDATA) la **parola** di memoria va vista come composta di 4 o 6 caratteri. Per quanto ci riguarda, assumeremo di

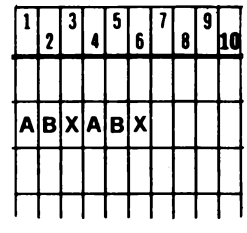
aver a che fare con parole di 4 caratteri ciascuna. Allora, se volessimo memorizzare i quattro caratteri:




con il FORMAT A4, otterremmo nella parola di memoria:


Se l'area di memoria prevista fosse troppo piccola per la stringa da memorizzare, o se, in altre parole, n fosse maggiore del numero di caratteri contenibili nella parola, si avrebbe un troncamento della stringa in input:

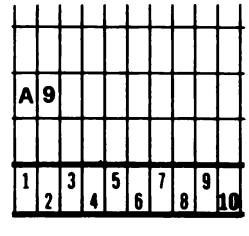
se infatti si avesse:




la lettura con il FORMAT A6 produrrebbe:


Se invece fosse la stringa in input ad essere più breve, si avrebbe un riempimento di blank a destra;

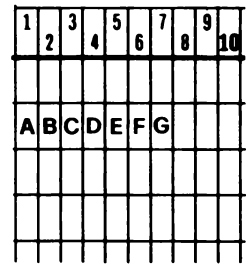
così se si avesse:



la lettura con il FORMAT A2 produrrebbe:

| A | 9 | | |
|---|---|---|---|

A questo punto può sorgere il problema di voler memorizzare stringhe di caratteri più lunghe di una **parola**. E' allora necessario utilizzare un fattore di ripetizione scrivendo un FORMAT del tipo **mAn**; così, nel caso si volesse memorizzare tutta la stringa:




si potrebbe utilizzare il FORMAT 2A4, ottenendo:

| A | B | C | D | | E | F | G | |
|---|---|---|---|---|---|---|---|---|

in due **parole** consecutive di memoria.

In output il format **An** produce la scrittura di n caratteri prelevandoli dalla parola di memoria a partire da sinistra, ma accostandoli a destra nel campo di stampa; così se il contenuto della parola fosse

| A | 1 | B | 2 |
|---|---|---|---|

con una specifica A4 si otterrebbe:

| A | 1 | B | 2 |
|---|---|---|---|

con A7:

| | | | A | 1 | B | 2 |
|---|---|---|---|---|---|---|

usando invece la specifica A3, il risultato sarebbe:

| A | 1 | B |
|---|---|---|

Se poi volessimo mettere in stampa il contenuto di più parole consecutive di memoria, come ad esempio

| A | 1 | B | 2 | | C | 3 | D | |
|---|---|---|---|---|---|---|---|---|

ancora dovremmo usare un fattore ripetitivo;
la specifica 2A4 permetterebbe di ottenere:

| A | 1 | B | 2 | C | 3 | D | |
|---|---|---|---|---|---|---|---|

mentre con 2A3 si sarebbe ottenuto

| A | 1 | B | C | 3 | D |
|---|---|---|---|---|---|

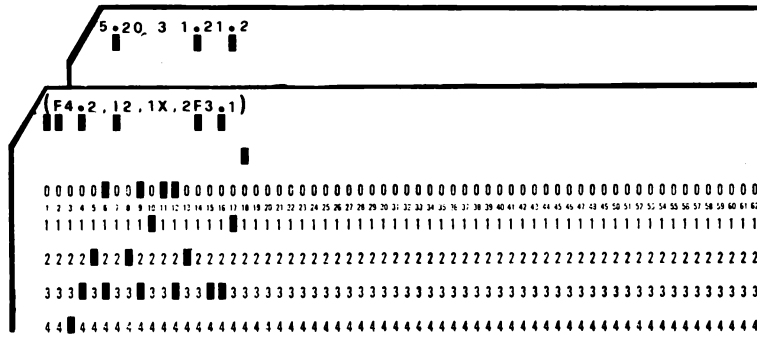
### 3.13 Il FORMAT variabile

Una delle più comode funzionalità ottenibili con l'uso del FORMAT A è sicuramente quella relativa al FORMAT variabile. Si tratta, in pratica, di scrivere la specifica come contenuto di una variabile di caratteri e ad esso faranno riferimento i verbi di input-output.

Vediamo subito un esempio che metta in grado di comprendere l'utilità di questo modo di procedere.

L'input sia affidato ad un file su schede; si vuole preparare un programma che vada bene in tutti i diversi casi di possibile elaborazione, comunque vengano scritte le schede stesse.

Il problema può essere facilmente risolto ponendo in testa al pacco di schede dati, una scheda descrittiva dei dati che seguono, cioè dando su quella prima scheda le stesse specifiche di formato che si scriverebbero nell'istruzione FORMAT.




È ovvio che se si trattasse di altrettante linee di scrittura su video si procederebbe allo stesso modo.

Nel programma si scriverebbe

```
      DIMENSION FRMT(5)
      .....
      .....
      READ(5,100) FRMT
  100 FORMAT(5A4)
      READ(5,FRMT) X,I,R1,R2
      .....
      .....
```

Come si vede, la prima READ ha il formato normale che già conosciamo e serve a memorizzare in FRMT la parte della prima scheda contenente la descrizione del FORMAT che dovrà essere usato successivamente; la secon-

da lettura, invece di avere il riferimento ad un FORMAT direttamente codificato nel programma attraverso una label, fa riferimento a quello contenuto nella variabile FRMT.

Come si vede la struttura dei dati di input rimane della massima generalità: sarà sufficiente ricordare di porre ogni volta come prima scheda quella con il FORMAT appropriato.

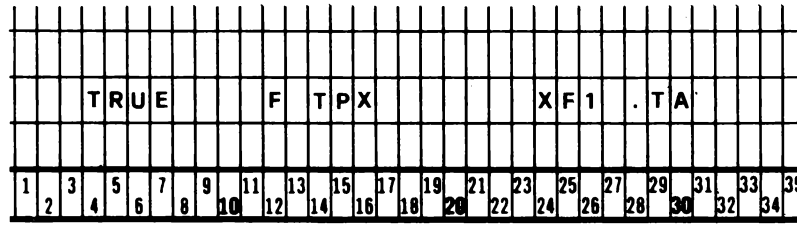
Il programma è divenuto così assai flessibile ed utilizzabile senza modifiche in situazioni diverse.

### 3.14 Il FORMAT L

Per le variabili logiche viene utilizzato il FORMAT L.

**Ln** indica in input un campo di lettura di n caratteri: se il primo di questi, diverso da blank, è una T o se si ha la successione .T, la variabile assume il valore .TRUE. (vero), in caso contrario o se il campo è tutto a blank, la variabile assume il valore di .FALSE. (falso).

Si abbia in input:




il programma

```
      LOGICAL A,B,C,D,E,F
      . . .
      . . .
      . . .
      READ(5,12) A,B,C,D,E,F
12    FORMAT(L8,2L4,L5,L5,L4)
      . . .
      . . .
      . . .
```

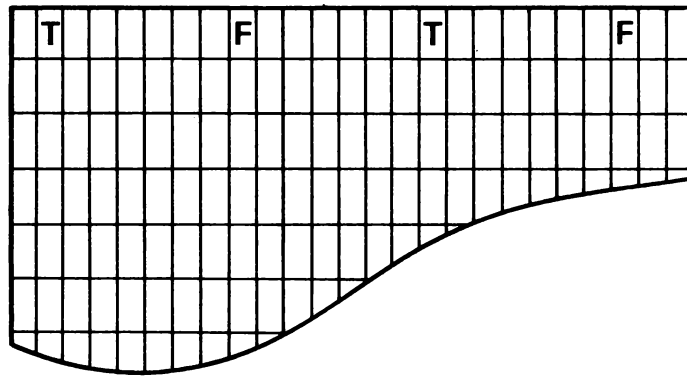
fa le seguenti assegnazioni:

```
A = .TRUE.
B = .FALSE.
C = .TRUE.
D = .FALSE.
E. = .FALSE.
F. = .TRUE.
```

In output **Ln** produce la scrittura di n caratteri di cui l'ultimo è T per le variabili uguali a .TRUE., F per quelle uguali a .FALSE., ed i precedenti sono posti a blank:

```
      LOGICAL A,B,C,D
      A = .TRUE.
      B = .FALSE.
      C = .TRUE.
      D = B
      WRITE(5,100) A,B,C,D
  100 FORMAT(1X,L2,2X,L5,2L7)
```

produce la stampa:




### 3.15 Il FORMAT G

Il FORMAT G può essere utilizzato per tutti i tipi di variabili con la sola eccezione di quelle di caratteri.

In input può essere utilizzato al posto dei FORMAT che già conosciamo e ciò nel senso che il FORMAT **Gn** per variabili intere ha lo stesso effetto di In, per variabili logiche ha lo stesso effetto di Ln ed inoltre **Gn.m** per variabili reali ha lo stesso effetto di Fn.m.

Ovviamente, dunque, tale specifica di formato si rende utile quando si desideri una certa uniformità di scrittura o, ancor più, quando si voglia

utilizzare un unico statement FORMAT in riferimento a diverse istruzioni di input-output su variabili, magari, di tipo diverso. Tuttavia i vantaggi maggiori di questa specifica stanno nell'uso che se ne può fare in output: in questo caso ci è possibile scrivere un numero reale con una dimensione in caratteri dipendente dalla grandezza del numero stesso. Ancora, per le variabili intere o logiche, **Gn** ha lo stesso effetto dei FORMAT In o Ln, ma per quelle reali **Gn.m** fa sì che il numero venga scritto in n-4 caratteri (incluso il punto decimale), seguito da quattro blank e con un numero di cifre decimali tanto più piccolo quanto più grande è il valore della variabile.

Le modalità precise con cui avviene tale scrittura sono le seguenti:

con il FORMAT Gn.m si voglia scrivere la variabile X:

| **se il valore di X è tale che** | **Il numero viene scritto come se fosse utilizzato il FORMAT** | **Così, ad esempio, con G10.3 se il valore di X è** | **viene scritto** |
|---|---|---|---|
| $0.1 \leqslant x \leqslant 1$ | F(n-4).m, 4X | 0.12345 | 0.123 |
| $1 < x \leqslant 10$ | F(n-4).(m-1), 4X | 1.2345 | 1.23 |
| $10 < x \leqslant 10^2$ | F(n-4).(m-2), 4X | 12.345 | 12.3 |
| ... | ... | | |
| $10^{m-2} < x \leqslant 10^{m-1}$ | F(n-4).(m-(m-1)),4X | | |
| $10^{m-1} < x \leqslant 10^m$ | F(n-4).0,4X | | |
| negli altri casi | notazione esponenziale En.m | 12345.6789 | 0.123E 05 |

### 3.16 Il fattore di scala P

Qualora si desideri uno spostamento del punto decimale presente nei numeri che intendiamo mettere in input o in output, si utilizza il **fattore di scala nP**.

In input esso può essere utilizzato con i FORMAT F,E,D, ma sui FORMAT E e D non ha alcun effetto; invece sul numero cui è attribuito il FORMAT F, nP sposta la posizione del punto decimale come se si moltiplicasse per $10^{-n}$ :

| **Se il numero scritto sulla scheda è** | **il FORMAT** | **produce la memorizzazione di** |
|---|---|---|
| 21.098 | 2PF7.3 | 0.21098 |
| 22.157 | —2PF6.3 | 2215.7 |
| 378.568E0 | 5PE9.0 | 378.568 |

In output il fattore di scala nP ha invece effetto su tutti i FORMAT numerici. Sul FORMAT F esso provoca uno spostamento del punto decimale come se si moltiplicasse per $10^n$, mentre nei FORMAT E e D avviene la moltiplicazione del numero per $10^n$ ed il contemporaneo aggiustamento dell'esponente così da lasciare inalterato il valore del numero stesso.

| **Se in memoria si trova il numero** | **con un FORMAT** | **si ha in stampa** |
|---|---|---|
| 0.21098 | 2PF8.3 | 21.098 |
| 2109.800 | —3PF8.3 | 2.110 |
| 2109.800 | 3PF6.3 | ****** |
| 2109.800 | 1PF8.2 | 21098.00 |
| 6665.43 | 0PE18.6 | 0.666543 E 04 |
| 6665.45 | 3PE18.6 | 666.543000 E 01 |

Va tenuto conto del fatto che il fattore di scala, una volta dichiarato, vale per tutte le specifiche che seguono e che, dunque, per toglierne l'effetto è necessario porre 0P.

Così, se in memoria si hanno le grandezze:

AB = 0.0072 C = 1.0 I = 261 K = 2 DZ = 3.00000151

con le istruzioni di output

```
      WRITE (6,100) AB,C,I,K,DZ
100   FORMAT(3X,2P2F7.2,2X,I8,0P,I5,2X,D14.8)
```

si ottiene in stampa:

```
   0.72  100.00        261     2   0.300000015D 01
```

Per concludere, una nota sull'utilità del fattore di scala: esso verrà utilizzato ogniqualvolta l'input sia costituito da numeri preceduti o seguiti da un ugual numero di zeri.

Se ad esempio dovessimo eseguire dei calcoli su una certa sequenza di numeri che in input si presentano così: 15.000.000, 20.000.000, 251.000.000,

7.500.000..., sarebbe del tutto inutile lavorare sui milioni: in questo ed in molti altri casi simili potrebbe essere più agevole prendere in input i numeri 15, 20, 251, 7.5..., utilizzando l'opportuno fattore di scala 6P, lavorare su tali grandezze ed infine emettere un output ancora con il fattore di scala 6P per ottenere nuovamente in milioni il risultato corretto.

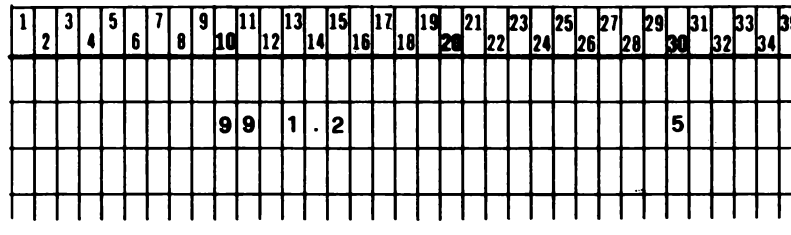
Forse tutto ciò può sembrare poco significativo poichè, dopotutto, l'elaboratore è certo in grado di agire su numeri di un simile ordine di grandezza. Tuttavia si tenga presente che per numeri ancora più grandi o per input coinvolgenti una mole di dati considerevole, l'economia di lavoro può concretizzarsi in un certo risparmio di tempo e di spazio in memoria e soprattutto in un più facile controllo del programma per quanto riguarda ad esempio l'esame dei risultati intermedi.

### 3.17 Il FORMAT T

Un'ultima specifica di FORMAT tra quelle che intendiamo trattare è di uso estremamente semplice: si tratta del FORMAT T.

E' sufficiente tenere presente che esso indica un posizionamento assoluto nel record e cioè **Tn** sta ad indicare che la prossima istruzione di lettura o scrittura deve iniziare dall'n-esimo carattere (o posizione) del record.

Se, ad esempio, si avesse un input del tipo:

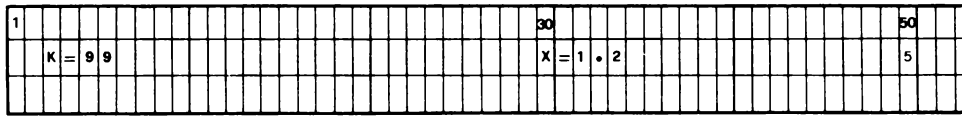



potremmo scrivere:

```
      .....
      READ(5,100) K,X,Y
100   FORMAT(T10,I2,1X,F3.1,T30,I1)
      .....
      WRITE(6,110) K,X,Y
110   FORMAT(3X,'K = ',I2,T30,'X = ',F3.1,T50,I1)

      .....
```

ottenendo in stampa:




La differenza sostanziale rispetto al FORMAT X che spesso viene usato al posto di tale specifica, (e che comunque, come ci si può facilmente rendere conto, risolve agevolmente ogni problema di posizionamento al pari del FORMAT T), consiste nel fatto che con quest'ultimo siamo in grado di assegnare direttamente la posizione da trattare: *tabuliamo* la prossima lettura o scrittura designando la posizione in modo assoluto. Con il FORMAT X, invece, indichiamo sempre quanti spazi intendiamo saltare, ma nel senso che, almeno in output, intendiamo non solo enumerarli, ma anche *scrivere* dei blank.

### 3.18 Controllo del carrello di stampa

Per quanto riguarda i FORMAT di output vanno fatte ancora alcune osservazioni riguardo la gestione dell'interlinea, il posizionamento a pagina nuova e così via, ovvero ciò che viene chiamato **controllo del carrello**.

Come si ottenga tale controllo è presto detto: normalmente il primo carattere di una riga non viene stampato, ma ha, appunto, il compito di gestirlo secondo i criteri della seguente tabella:

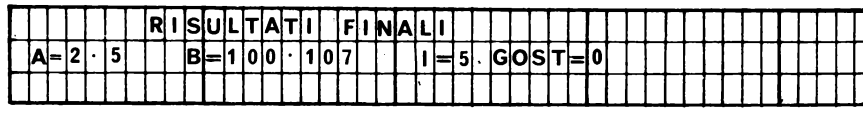
| se il primo carattere è | si ottiene |
|---|---|
| blank | normale interlinea rispetto alla riga precedente |
| 0 | doppia interlinea rispetto alla riga precedente |
| + | nessuna interlinea (la riga viene stampata sovrapposta alla precedente) |
| 1 | posizionamento a pagina nuova |

All'interno di un FORMAT è possibile andare a capo inserendo il carattere /.

Così, se A = 2.5 B = 100.107 I = 5, l'istruzione di scrittura con il relativo FORMAT:

```
    WRITE(6,100)
100 FORMAT(1H1,7X,17HRISULTATI FINALI/1H ,'A = ',6X,'B = ',10X,'I = ')
    WRITE(6,150) A,B,I
150 FORMAT(1H+,3X,F3.1,5X,F7.3,5X,I1,1X,'COST = 0')
```

produce la stampa:




Un ultimo avvertimento: abbiamo detto che il controllo vale per le specifiche di output: va posta attenzione al fatto che in un eventuale FORMAT di output che non abbia per destinazione la stampa (ma ad esempio una registrazione su nastro, disco, ecc...) il primo carattere perde la sua funzione di controllo e viene trattato come un carattere normale.

### 3.19 Istruzione NAMELIST

Un altro modo di rendere i programmi sufficientemente generali rispetto al formato dei dati è quello di evitare del tutto il ricorso alla specifica di FORMAT utilizzando lo statement NAMELIST. Come vedremo tra breve questo modo di procedere ci permetterà di rendere la codifica indipendente dai formati di input, ma potrà essere utilmente impiegato anche in output, per stampe semplici non strutturate in particolari impaginazioni.

Lo statement NAMELIST permette l'assegnazione di un nome di gruppo ad una lista di variabili ed il suo formato è:

NAMELIST/$nome_1$ /$a_1$ ,$a_2$ ,...,$a_n$ /$nome_2$ /$b_1$ ,$b_2$ ,...,$b_m$ .../$nome_k$ /$x_1$ ,$x_2$ ,...,$x_h$

dove $nome_1$, $nome_2$ ,... $nome_k$ sono i nomi di gruppo rispettivamente delle liste di variabili:

$a_1$ ,$a_2$ ,..,$a_n$ $b_1$ ,$b_2$ ,..,$b_m$ ... $x_1$ ,$x_2$ ,..,$x_h$

I vari nomi di gruppo potranno essere utilizzati nei verbi di input-output al posto del riferimento al FORMAT.

Così, nell'esempio che segue, si hanno tre gruppi di variabili di input-output: per quanto riguarda l'input va tenuto presente che i valori saranno passati al programma dall'esterno come al solito, anche se con modalità particolari; per quanto riguarda l'output invece, si otterrà una stampa di tutto il gruppo di variabili e secondo formati particolari che vedremo.

```
...                                                    } schede
...                                                    } di controllo
...                                                    } (richiesta
                                                       } di compilazione)
NAMELIST/LISTX/V,I,K/STAM/A,B,K
NAMELIST/LISTY/N,LOGIC,CHAR
DIMENSION I(5)
...
...
READ(5,LISTX)
READ(5,LISTY)                                          } programma
...
...
WRITE(6,STAM)
...
...
STOP
END
...                                                    } schede
...                                                    } di controllo
                                                       } (richiesta
                                                       } di esecuzione)
$LISTX V=2,5,I(1)=1,I(2)=2,I(3)=3,                     } dati di input
I(4)=10,I(5)=11,K=200 $                                } dati di input
$LISTY N=2931,LOGIC=.T.,CHAR="MILANO"$                 } dati di input
```

Come si può osservare i dati di input sono scritti secondo le usuali assegnazioni del FORTRAN su delle schede che portano il nome del gruppo di NAMELIST. Tale gruppo è racchiuso da un simbolo particolare (nel nostro caso è stato scelto il $, ma tale carattere dipende dal sistema di cui si fa uso) posto all'inizio ed alla fine del gruppo stesso.

Per quanto riguarda l'output si sarebbe ottenuta una stampa del tipo:

```
---
NAMELIST STAM
A = --- B = --- K = ---
END NAMELIST STAM
```

ed i FORMAT implicitamente utilizzati per tale scrittura sarebbero stati quelli definiti come FORMAT standard del sistema; questi dipendono, ap-

punto, dal sistema, ma generalmente sono i seguenti:

| | |
|---|---|
| per grandezze REALI | si ha il FORMAT E16.8 |
| per grandezze INTERE | si ha il FORMAT I16 |
| per grandezze LOGICHE | si ha il FORMAT L2 |
| per grandezze DOPPIA PREC. | si ha il FORMAT D26.18 |
| per grandezze COMPLESSE | si ha il FORMAT 2E16.8 |

Infine, a precisare quanto visto nell'esempio, vanno tenute presenti alcune regole ed osservazioni di una certa importanza:

1) lo statement NAMELIST deve precedere qualunque riferimento alle variabili contenute nella lista (è di tipo dichiarativo e va, comunque, premesso agli altri statement dichiarativi relativi a quelle variabili);
2) sono possibili più liste di variabili, come si è visto, in un unico statement di NAMELIST, ma anche la presenza di più NAMELIST ed una stessa variabile può appartenere a più di una lista;
3) le variabili dichiarabili in NAMELIST possono essere di qualunque tipo; è permesso dichiarare, nella lista, degli array; essi però non possono avere più di tre indici.

CAPITOLO 4

# FILE SU DISCO E NASTRO

## 4.1 Periferiche a supporto magnetico e tipo di accesso

I comandi di I/O che sono stati presentati sino ad ora, fanno riferimento spesso a quei tipi di supporti che sono stati maggiormente usati sin dai primi tempi di esistenza del FORTRAN. Così in fase di lettura si è sempre pensato alle schede perforate, od alla linea di input su video, ed in fase di scrittura ai tabulati di stampa: questi supporti implicano sempre la necessità di una transcodifica dal codice Hollerith al codice di rappresentazione interna dell'elaboratore e, viceversa, dalla rappresentazione interna ai codici adatti per la stampa.

D'altra parte sappiamo che, per memorizzare le informazioni, esistono anche altri tipi di supporti e precisamente quelli magnetici: dischi, nastri e diskette. E' immediato capire come, relativamente a tali supporti, i problemi di input-output siano diversi rispetto a quelli tipo la scheda, la carta o il terminale. Quindi anche i comandi che ci permettono di agire sui supporti magnetici, presentano qualche nuova particolarità.

Una prima caratteristica essenziale dei supporti magnetici è che, a differenza di quanto si è fin'ora presupposto, cioè di poter accedere alle informazioni solo in modo sequenziale, essi permettono oltre che un accesso appunto **sequenziale** (dischi, nastri, diskette), anche un accesso **diretto** (dischi, diskette).

D'ora in avanti chiameremo **file** ogni insieme di informazioni da trattare in input-output, e diremo **record** ogni unità informativa del file. Così, ad esempio, nel caso di un file su schede, ogni singola scheda sarà un record, per un file di stampa ogni riga potrà essere vista come un record.

Diciamo che un file è trattato in **accesso sequenziale** quando i suoi record vengono elaborati in modo consecutivo, secondo la loro sequenza, secondo cioè l'ordine col quale sono presenti nel file stesso.

Un file è trattato invece in **accesso diretto** o **random** quando le sue informazioni vengono elaborate indipendentemente dal loro ordine nel file, quando, cioè, si accede direttamente ad un record voluto.

Può accadere ad esempio che l'elaborazione delle informazioni avvenga trattando per primo il centesimo record, poi il cinquantesimo, poi il centounesimo e così via a seconda delle esigenze del trattamento. Per poter fare ciò dovremo poter riconoscere in qualche modo il record desiderato, così da informarne il sistema; ma ciò che più conta è il fatto che, quando si desidera procedere in questa maniera, la struttura del file deve essere adatta. Non potremo accedere direttamente ad una certa scheda né ad un certo record di un file su nastro, come non potremo prima scrivere la 100-esima riga di stampa e poi la ventesima: il programma deve leggere tutti i record di quel tipo in sequenza ed emettere sequenzialmente tutte le varie righe di stampa. Ecco perchè, come si era accennato tra parentesi, l'accesso diretto è possibile solo per file su disco o diskette.

Oltre alla grossa suddivisione tra i due tipi di accesso, va tenuta presente un'altra importante questione: o, come abbiamo già visto, esiste il problema di decodificare i dati nel passaggio tra memoria ed unità periferica, oppure, per un qualche motivo, si desidera che ciò non avvenga come, ad esempio, quando si vogliono copiare i dati così come stanno in memoria.

Nel momento in cui si accede ad un'unità a supporto magnetico, non esiste il problema di avere i dati in formato a noi comprensibile e l'unica cosa che conta è quella di sapere *come* i dati sono memorizzati. Così, per quanto riguarda sia la scrittura che la lettura delle informazioni, i comandi di I/O potranno fare uso di un enunciato FORMAT oppure no.

Vediamo qui di seguito quali caratteristiche presentino i comandi di I/O sia per quanto riguarda l'accesso ai file (sequenziale o diretto), sia per quanto riguarda la necessità di decodifica (comandi di I/O con associato FORMAT oppure no).

### 4.2 File ad accesso sequenziale e comandi di I/O relativi

Oltre ai noti file, di cui abbiamo già parlato, anche le periferiche a supporto magnetico permettono la gestione di file ad accesso sequenziale.

Nel caso si vogliano memorizzare i dati nello stesso modo in cui poi verrebbero stampati, si farà uso dei comandi di I/O già visti e con l'enunciato FORMAT contenente i codici di decodifica.

La situazione è diversa nel caso si vogliano copiare le informazioni così come sono presenti in memoria: in tal caso non dovremo far uso dell'enunciato FORMAT.

*COMANDI DI I/O SENZA FORMAT PER L'ACCESSO SEQUENZIALE* - Questi enunciati ci permettono di trasferire da memoria centrale a memoria di massa, e viceversa, delle variabili, degli array o degli elementi di array. Tale trasferimento avviene senza l'interpretazione e le trasformazioni imposte dal FORMAT stesso: si è soliti dire che sul file su disco o nastro si trova l'immagine di memoria del dato (ne è infatti l'esatta copia). Tratteremo prima di tutto il caso di un accesso sequenziale.

Potremo usare i comandi:

WRITE(n,ERR = $l_2$) lista

e

READ(n,END = $l_1$ ,ERR = $l_2$) lista (1)

dove:

n è una costante o variabile intera e rappresenta il numero logico associato alla particolare unità di I/O (che dipende da caso a caso);

$l_1$ è la label di un punto di programma a cui si vuole saltare nel caso si incontri la fine del file di input;

$l_2$ è la label di un punto di programma a cui si vuol saltare nel caso si abbia un errore durante l'esecuzione del comando;

lista è l'insieme delle variabili, array od elementi di array che si vogliono scrivere o leggere con un unico comando di I/O.

Così, scrivendo:

WRITE(26) ((A(I,K),K = 1,100,2),I = 1,20,2)

si scrivono sull'unità associata logicamente al numero 26, tutti gli elementi dell'array A aventi indice di riga e di colonna dispari, e tutto questo senza decodificare i dati, ma copiandoli così come sono in memoria.

Nel caso si voglia saltare un record, si darà un ordine di I/O senza lista: per esempio:

READ(18)

---

(1) Le specifiche END ed ERR sono comunque facoltative.

significa saltare il record attualmente a disposizione e posizionarsi sul prossimo.

Agendo su file sequenziali può a volte essere necessario operare dei posizionamenti sul file e questi possono essere di due tipi: o riposizionamento all'inizio del file oppure all'inizio del record precedente a quello a cui si è attualmente posizionati. Inoltre, nel caso si stia elaborando un file sequenziale in output, ad un certo punto occorrerà precisare che tale file è terminato.

Per ottenere tutte queste funzioni esistono degli appositi comandi: REWIND, BACKSPACE, ENDFILE.

*COMANDO REWIND* - Il comando REWIND ha la funzione di far riposizionare all'inizio di un file sequenziale su nastro o disco. Il suo formato è:

REWIND n

dove:

n è una costante o variabile intera che identifica il numero logico del file sul quale si vuole eseguire il riposizionamento.

*COMANDO BACKSPACE* - Il comando BACKSPACE permette di posizionarsi sul record precedente a quello su cui ci si trova attualmente. Il suo formato è:

BACKSPACE n

dove:

n è una costante o variabile intera indicante il file su cui si vuole agire.

Il comando BACKSPACE può essere dato solo dopo un comando di READ o WRITE.

*COMANDO ENDFILE* - Il comando ENDFILE serve a terminare un file sequenziale di output, che può essere sia su schede, che nastro o disco. In ognuno dei rispettivi casi farà perforare una scheda con dei caratteri particolari oppure registrerà dei record particolari di fine file su nastro o disco. Il suo formato è:

ENDFILE n

dove:

n è una costante o variabile intera indicante il file di output in cui si vuole la segnalazione di fine file.

Dopo il comando ENDFILE, ed ovviamente per il file sottoposto all'azione del comando, è ammesso il solo uso del comando REWIND.

### 4.3 File ad accesso diretto e comandi di I/O relativi

Come si è già osservato i file ad accesso diretto sono essenzialmente file su disco e la denominazione **accesso diretto** fa riferimento alla possibilità di accedere ad un record all'interno del file, senza dover leggere preventivamente tutti quelli che lo precedono.

Per poter usare file ad accesso diretto occorre fornire preventivamente alcune informazioni tramite il comando DEFINE FILE: solo dopo si potranno usare i veri e propri verbi di I/O che sono: FIND, READ, WRITE.

Anche nel caso di file ad accesso diretto è possibile scambiare con le periferiche informazioni decodificate oppure no, ossia far uso di verbi di I/O con FORMAT oppure senza.

*COMANDO DEFINE FILE* - Il comando DEFINE FILE serve per precisare quali file, durante l'esecuzione del programma, possono essere utilizzati ad accesso diretto. Tale comando deve essere eseguito prima di ogni altro comando di I/O relativo a tali file. Essi sono specificati nella lista del comando il cui formato è:

DEFINE FILE $n_1$ ($r_1$, $m_1$, $x_1$, $v_1$),$n_2$ ($r_2$, $m_2$, $x_2$, $v_2$)...

dove:

$n_i$ è una costante o variabile intera associata ad un file su disco ad accesso diretto;

$r_i$ è una costante intera che specifica il numero di record nel file;

$m_i$ è una costante intera che specifica l'ampiezza massima di un record del file o in termini di caratteri (o byte) o di parole a seconda di quanto espresso dal successivo parametro $x_1$.

$x_i$ è una lettera: **L** specifica che il trasferimento riguarda informazioni che possono essere decodificate (presenza dell'enunciato FORMAT) oppure no; in ogni caso la dimensione data dal parametro m è in caratteri o byte;

**E** specifica che i dati da trasferire
sono sottoposti all'azione di un FORMAT;
la dimensione fornita dal parametro m è
in caratteri o byte;

**U** specifica che i dati da trasferire
non sono sottoposti all'azione di un enunciato
FORMAT; la dimensione fornita dal parametro
m è in parole;

$v_i$ è una variabile intera associata al file: durante l'esecuzione di una READ o una WRITE conterrà un valore corrispondente alla posizione del prossimo record del file.

*Esempio:*

DEFINE FILE 4(1000,140,L,F4),8(100,80,U,F8)

precisa che il file 4 contiene 1000 record, la cui lunghezza in byte (parametro L) è di 140 e la variabile di controllo è F4; inoltre il file 8 contiene 100 record lunghi 80 parole (parametro U) e la variabile di controllo è F8.

*COMANDO FIND* - Serve per aumentare la rapidità di accesso ai dati, facendo posizionare le testine di un disco sul record desiderato. Il suo formato è:

FIND(n'e)

dove:

n è una costante o variabile intera identificante il file cui si vuole accedere. Notare che n è immediatamente seguita da un apostrofo e non da una virgola scritta in alto per errore di stampa.

e è un'espressione intera (che utilizza eventualmente la variabile $v_i$ vista nel comando DEFINE FILE) che indica la posizione del record, all'interno del file, su cui ci si vuole posizionare.

Dopo il comando FIND si darà il comando READ relativo al record che si trova nella stessa posizione. Il comando FIND non è indispensabile in un'elaborazione, ma può essere utile quando vi siano molte operazioni di I/O, per diminuire i tempi di accesso ai record.

*COMANDO READ* - Anche per i file ad accesso diretto il comando READ svolge la funzione di prelevare le informazioni da un supporto esterno e

portarle in memoria centrale. Il suo formato è:

$$\text{READ(n'e,l,END}=l_1\text{,ERR}=l_2\text{) lista} \qquad (1)$$

dove:

n (seguita da un apice) è una costante o variabile intera indicante il file su cui si vuole leggere.

e è un'espressione intera (che utilizza eventualmente la variabile $v_i$ vista nel comando DEFINE FILE) che indica la posizione del record, all'interno del file, su cui ci si vuole posizionare.

l è una label (opzionale) di un eventuale enunciato FORMAT nel caso fosse necessario decodificare i dati durante il caricamento in memoria centrale

$l_1$ è la label di un punto di programma a cui si intende saltare nel caso si arrivi alla fine del file

$l_2$ è la label di un punto di programma a cui si vuole saltare nel caso si riscontri un errore durante l'esecuzione della READ

lista è l'insieme delle variabili, array od elementi di array che si vogliono leggere.

*COMANDO WRITE* - Anche per i file ad accesso diretto il comando WRITE svolge la funzione di trasferire sul supporto esterno il contenuto della memoria centrale. Il suo formato è:

$$\text{WRITE(n'e,l,ERR}=l_2\text{) lista}$$

dove:

n è una costante o variabile intera (seguita da un apice) che identifica il file su cui si vuole scrivere;

e è un'espressione intera che indica la posizione del record all'interno del file;

l è la label (opzionale) di un eventuale enunciato FORMAT in caso di necessità di decodifica dei dati durante il trasferimento delle informazioni da memoria centrale ad unità periferica;

---

(1) END ed ERR sono ancora specifiche facoltative.

$l_2$ è la label di un punto del programma a cui si vuole saltare nel caso si riscontri un errore durante l'esecuzione della WRITE;

lista è l'insieme delle variabili, array od elementi di array che si vogliono scrivere sul file.

Negli statement READ e WRITE di accesso diretto si è detto che il riferimento al FORMAT (label 1) è opzionale: si ponga attenzione al fatto che sulla maggior parte dei sistemi un trattamento in accesso diretto è permesso solo senza FORMAT. Ciò è dovuto al fatto che in essi la struttura fisica del file è studiata in modo tale da permettere la massima rapidità di I/O e dunque il tutto deve avvenire senza interpretazioni e relative trasformazioni dei contenuti.

CAPITOLO 5

# SOTTOPROGRAMMI

## 5.1 Funzioni intrinseche

Nei problemi di tipo scientifico o tecnico si presenta, ovviamente assai spesso, la necessità di eseguire operazioni particolari come l'estrazione di radice o calcoli di funzioni matematiche tipiche quali il seno o il logaritmo che, se dovessimo codificare estesamente in un programma, richiederebbero non poche istruzioni e procedimenti abbastanza laboriosi.

Il FORTRAN ci soccorre in questi casi con una serie di **Funzioni intrinseche** che permettono di ottenere il risultato per mezzo di una unica istruzione: se, ad esempio, volessimo calcolare la formula $a = \sqrt{b}$, dando nome B alla variabile contenente il numero b ed A alla variabile in cui desiderassimo memorizzare il risultato, si scriverebbe:

A = SQRT(B)

allo stesso modo, volendo calcolare a = Log b:

A = ALOG10(B)

Il formato generale in cui deve essere scritta la funzione è il seguente:

**Risult = Funz(Arg)**

dove **Funz** è la particolare funzione che intendiamo applicare, o meglio il suo nome FORTRAN, **Risult** viene detto valore della funzione (noi lo chiameremo anche risultato della funzione), **Arg** argomento.

A scopo esemplificativo vediamo una prima serie di funzioni normalmente utilizzabili e presenti su tutti i compilatori FORTRAN, cioè su qualunque tipo di elaboratore: si tratta di una prima elencazione molto incompleta ma generale.

Prima TABELLA Funzioni Intrinseche

| **Funzione ed operazione matematica** | **Funzione intrinseca FORTRAN** | **Argomento x** | **Risultato y** |
|---|---|---|---|
| $y = \sqrt{x}$ | Y = SQRT (X) | reale | reale |
| $y = e^x$ | Y = EXP (X) | reale | reale |
| $y = lg_e\ x$ | Y = ALOG (X) | reale | reale |
| $y = \|x\|$ | Y = ABS (X) | reale | reale |
| y = sen x | Y = SIN (X) | reale in radianti | reale |
| y = cos x | Y = COS (X) | reale in radianti | reale |
| y = tan x | Y = TAN (X) | reale in radianti | reale |

*Tab. 21*

Più avanti è elencata in tabella una serie più completa di funzioni intrinseche, per ora cerchiamo di fissare le idee su quanto visto con alcuni esempi:

| **Argomento** | **Operazione** | **Funzione (nome della)** | **Risultato (valore della funzione)** |
|---|---|---|---|
| A = 4.0 | $b = \sqrt{a}$ | B = SQRT (A) | B = 2.0 |
| X = 3.14 | y = cos x | Y = COS (X) | Y = —1 |
| X = 2. | y = sen x | Y = SIN (X) | Y = 0.9093 |
| Z = —100.2 | y = \|z\| | Y = ABS (Z) | Y = 100.2 |
| A = 1 | $y = e^a$ | Y = EXP (A) | Y = e = 2.718... |
| X = 3 | $y = \text{sen } e^x$ | Y = SIN (EXP (X)) | $Y = \text{sen } e^3 = 0.9445$ |

Come si vede dall'ultimo esempio è possibile porre come argomento di una funzione un'altra funzione; è anche possibile porre come argomento una espressione valida il cui risultato fungerà da argomento vero e proprio (e dovrà dunque essere del tipo appropriato): così, ad esempio, la formula:

$$y = \sqrt{\frac{\text{sen } (a + b)}{e^a - \log b}}$$

potrebbe venire scritta nell'unica istruzione:

```
Y=SQRT(SIN(A+B)/(EXP(A)—ALOG(B)))
```

D'altra parte un'espressione può contenere più funzioni utilizzate alla maniera di normali operatori: così la formula:

$$y = k + \text{sen } x + \sqrt{a - b} + c^5 - \log \frac{\text{sen } x}{e^x}$$

potrebbe venire scritta nell'istruzione:

```
Y=K+SIN(X)+SQRT(A-B)+C**5—ALOG(SIN(X)/EXP(X))
```

Va tenuto presente che l'argomento di ogni funzione deve essere, comunque e sempre, del tipo appropriato (lo si veda in tabella) e che il risultato sarà del tipo designato (a seconda del tipo di funzione): per quasi tutte le funzioni il premettere la lettera D significa fare riferimento a variabili in doppia precisione (risultato e/o argomento); premettere la lettera C significa fare riferimento a variabili complesse; in tutti gli altri casi il nome della funzione è tale da definire implicitamente il tipo del risultato (si veda a tale proposito la tabella che segue). Come si è già detto essa reca l'elenco più completo delle funzioni intrinseche di uso più comune e diffuso e che si trovano sulla maggior parte dei compilatori FORTRAN oggi esistenti. Aggiungiamo solo che neppure questa tabella è esaustiva e che vi possono essere alcune differenze da elaboratore ad elaboratore: rimandiamo per maggiori dettagli alla documentazione specifica delle case costruttrici.

| **Funzione od operazione matematica** | **Funzione intrinseca FORTRAN** | **Argomento $X, X_1, X_2, \ldots$** | **Risultato Y (valore della funzione)** |
|---|---|---|---|
| $y = \sqrt{x}$ | Y = SQRT (X) | reale | reale |
| $y = \sqrt{x}$ | Y = DSQRT (X) | doppia precis. | doppia precis. |
| $y = \sqrt{x}$ | Y = CSQRT (X) | complesso | complesso |
| $y = e^x$ | Y = EXP (X) | reale | reale |
| $y = 10^x$ | Y = EXP10 (X) | reale | reale |
| $y = e^x$ | Y = DEXP (X) | doppia precis. | doppia precis. |
| $y = 10^x$ | Y = DEXP10 (X) | doppia precis. | doppia precis. |
| $y = e^x$ | Y = CEXP (X) | complesso | complesso |
| $y = \log x$ | Y = ALOG (X) | reale | reale |
| $y = \text{Log } x$ | Y = ALOG10 (X) | reale | reale |
| $y = \log x$ | Y = DLOG (X) | doppia precis. | doppia precis. |

| | | | |
|---|---|---|---|
| y = Log x | Y = DLOG10 (X) | doppia precis. | doppia precis. |
| y = log x | Y = CLOG (X) | complesso | complesso |
| y = \|x\| | Y = ABS (X) | reale | reale |
| y = \|x\| | Y = IABS (X) | intero | intero |
| y = \|x\| | Y = DABS (X) | doppia precis. | doppia precis. |
| y = \|x\| | Y = CABS (X) | complesso | reale |
| y = parte intera di x | Y = INT (X) | reale | intero |
| y = parte intera di x | Y = AINT (X) | reale | reale |
| y = parte intera di x | Y = IDINT (X) | doppia precis. | intero |
| resto della divisione di $x_1$ per $x_2$ | Y = MOD ($X_1$, $X_2$) | interi | intero |
| | Y = AMOD ($X_1$, $X_2$) | reali | reale |
| | Y = DMOD ($X_1$, $X_2$) | doppia precis. | doppia precis. |
| conversione da: | | | |
| x intero a y reale | Y = FLOAT (X) | intero | reale |
| x reale a y intero | Y = IFIX (X) | reale | intero |
| y = sen x | Y = SIN (X) | reale (in radianti) | reale |
| y = sen x | Y = DSIN (X) | doppia precis. (rad) | doppia precis. |
| y = sen x | Y = CSIN (X) | complesso | complesso |
| y = cos x | Y = COS (X) | reale (rad) | reale |
| y = cos x | Y = DCOS(X) | doppia precis. (rad) | doppia precis. |
| y = cos x | Y = CCOS (X) | complesso | complesso |
| y = tang x | Y = TAN (X) | reale (rad) | reale |
| y = tang x | Y = DTAN (X) | doppia precis. (rad) | doppia precis. |
| y = tang x | Y = CTAN (X) | complesso | complesso |
| y = arsen x | Y = ARSIN (X) | reale | reale (rad) |
| y = arcos x | Y = ARCOS (X) | reale | reale (rad) |
| y = artang x | Y = ATAN (X) | reale | reale (rad) |
| y = Sinh x | Y = SINH (X) | reale | reale |
| y = Cosh x | Y = COSH (X) | reale | reale |
| y = Tangh (x) | Y = TANH (X) | reale | reale |
| generazione randomizzata di un numero y tale che $0 < y < x$ | Y = RAND (X) | reale | reale |

*Tab. 22*

Come ultimo, ulteriore, esempio analizziamo il seguente problema: scrivere un programma che legga un numero reale a, calcoli il resto della divisione per 2 π, calcoli seno e coseno di tale resto e quindi il valore della formula:

$$x = a + \text{sen } a + \frac{10^a}{e^a}$$

Potremmo scrivere, ad esempio:

```
      ...
      READ(5,100) A
  100 FORMAT(F5.2)
      B = AMOD(A,6.28)
      S=SIN(B)
      C=COS(B)
      X=A+SIN(A)+EXP10(A)/EXP(A)
      ...
```

### 5.2 Statement Function

Come si sarà certamente notato le funzioni intrinseche soddisfano ad una già vasta parte delle possibili esigenze di programmazione FORTRAN. Tuttavia può accadere che a volte si debba utilizzare ripetutamente il calcolo di una formula o di una particolare funzione non appartenente al gruppo di quelle intrinseche: in questi casi il FORTRAN dà la possibilità di definire la funzione voluta, **Statement Function** appunto, e di utilizzarla poi nel programma allo stesso modo delle funzioni già viste.

Tale definizione ha la forma

**Funz($arg_1$ ,$arg_2$ ,...,$arg_n$)=espressione**

dove: **Funz** è il nome che prenderà la nostra funzione,

**$arg_1$ ,$arg_2$ ,...,$arg_n$** l'elenco degli argomenti (i nomi che verranno utilizzati in **espressione**).

**espressione** è appunto l'espressione che costituisce la nostra funzione.

Prima di analizzare le regole cui deve sottostare lo **Statement Function**, vediamo di chiarirne subito l'utilizzo con un esempio: si debba calcolare, in un certo programma, il valore della funzione polinomiale di terzo grado

$$y = x^3 + 3x^2 + 3x + 1$$

per tutta una serie di punti.

In testa al programma scriveremo la definizione:

**POL3**(X)=X**3+3*X**2+3*X+1

se, poi, il vettore V(100) contiene il valore dei punti per ognuno dei quali si vuole calcolare il valore della funzione, potremo scrivere:

```
      DO 10 I=1,100
      Y=POL3(V(I))
      .....
      .....
10    CONTINUE
```

Come si è prima accennato, lo **Statement Function** deve soddisfare alcune regole:

1) deve occupare un unico statement;

2) deve trovarsi nella stessa unità di programma in cui viene fatto l'utilizzo della funzione (cioè non può, ad esempio, venir fatta una definizione nel programma principale e l'utilizzo della funzione in un sottoprogramma);

3) il nome della funzione non può essere messo in COMMON, EXTERNAL, EQUIVALENCE e deve precedere tutte le altre istruzioni eseguibili;

4) il nome della funzione deve definire implicitamente od esplicitamente il tipo di variabile costituente il risultato della funzione;

5) il risultato della funzione deve essere, ovviamente, unico.

Come si può osservare dal formato generale, il numero degli argomenti può essere anche maggiore di uno: al proposito va notato che il riconoscimento dell'argomento stesso avviene in base all'ordine e non in base al nome assegnato. Così ad esempio nel caso prima visto della funzione POL3, l'argomento al momento della definizione è stato chiamato X, mentre al momento dell'utilizzo della funzione è stato chiamato V(I), ma entrambi i nomi faranno riferimento alla stessa area di memoria e cioè, in definitiva, allo stesso valore.

Si osservino ora i seguenti esempi:

si debbano calcolare le formule $v = \frac{s}{t}$ ; $a = \frac{s}{t^2} = \frac{v}{t}$; $v = v_0 + at$ per vari

valori di $s$ e $t$; potremmo definire le funzioni:

```
VEL(S,T)=S/T
ACC(S,T)=S/T**2
VF(VI,A,T)=VI+A*T
```

e l'utilizzo:

```
...
V=VEL(S,T)
...
A=ACC(X,Y)
...
VFIN=VF(V,A,T)
```

oppure

```
VFIN=VF(V,ACC(X,Y),T)
```

Si voglia ora calcolare $w = sen\,(x + y) + \sqrt{gz}$ prendendone la parte intera: potremmo definire la funzione

```
PUL1(X,Y,Z)=INT(SIN(X+Y)+SQRT(9.8*Z))
```

per farne poi un utilizzo del tipo:

```
P=PUL1(S,F,H)
```

Come si noterà, per tornare al discorso precedente, il riconoscimento degli argomenti avviene in base all'ordine: nella funzione VF, VI corrisponde a V, A ad A, T a T; nella funzione PUL1, X ad S, Y ad F, Z ad H. E' ovvio che, mentre nella definizione della funzione, i nomi degli argomenti sono qualunque (usati o non più usati nel programma), al momento del richiamo della funzione gli argomenti devono essere già stati definiti (ed il calcolo avverrà sul valore che hanno in quel momento).

Per concludere, un'ultima nota riguardo ad un fatto che sicuramente si sarà osservato negli esempi: una funzione può essere definita utilizzando altre funzioni precedentemente definite o funzioni intrinseche.

Lo **Statement Function** permette la definizione di una funzione che diventa tale a tutti gli effetti per quella unità di programma.

Tanto per vedere un piccolo programma completo che faccia uso degli **Statement Function** consideriamo il seguente problema: su ogni scheda di un file di 100 schede sono scritti accostati a partire da colonna 1 e nel formato F5.2 i tre coefficienti a, b, c di un'equazione di secondo grado della forma $ax^2 + bx + c = 0$. Si vuole calcolare ogni coppia di radici e stamparle.

Potremmo codificare un programma del tipo:

```
    DISCR(A,B,C)=B*B—4*A*C
    X1(X,Y,Z)=(—X—SQRT(Y))/(2*Z)
    X2(X,Y,Z)=(—X+SQRT(Y))/(2*Z)
  1 READ(5,100,END=7)A,B,C
100 FORMAT(3F5.2)
    D=DISCR(A,B,C)
    IF(D) 4,5,5
  4 D=—D
    CR=—B/(2*A)
    CI=SQRT(D)/(2*A)
    CI=—CI
    WRITE(5,200)CR,CI,CR,CI
    GO TO 1
  5 R1=X1(B,D,A)
    R2=X2(B,D,A)
    WRITE (6,220)R1,R2
    GO TO 1
  7 STOP
200 FORMAT("RADICI COMPLESSE (",F8.3,",",F8.3,")",10X,"(",F8.3,",",F8,3,")")
220 FORMAT("RADICI REALI",F6.3,2X,F6.3)
    END
```

### 5.3 Sottoprogramma FUNCTION

Qualora non sia possibile scrivere la funzione tutta su di un unico statement, il FORTRAN permette ugualmente la sua definizione, però, questa volta, trattando la funzione come un sottoprogramma.

In questo modo la funzione dovrà essere codificata, per così dire, a parte e tale codifica dovrà avere la forma

```
FUNCTION nomef(arg1,arg2 ,....,argn)
. . .
. . .
. . .
RETURN
END
```

dove **nomef** è il nome assegnato alla funzione e $\mathbf{arg_1}$ ,$\mathbf{arg_2}$ ,....,$\mathbf{arg_n}$ i suoi argomenti. L'istruzione RETURN ha la funzione di restituire il controllo al programma chiamante.

Per richiamare una FUNCTION si procederà allo stesso modo delle funzioni già viste. Il riconoscimento degli argomenti avviene ancora in base al loro ordine, il sottoprogramma FUNCTION restituisce un solo valore che, nel programma principale, è appunto la variabile **nomef**. Va posta attenzione al fatto che **nomef** deve soddisfare le definizioni implicite di tipo; nel caso si renda necessaria una definizione di tipo esplicito essa va premessa alla parola FUNCTION:

```
INTEGER FUNCTION ALFA(...)
DOUBLE PRECISION FUNCTION BETA(...)
COMPLEX FUNCTION GAMMA(...)
. . .
. . .
```

Vediamo un primo esempio: si voglia calcolare il fattoriale di una certa serie di numeri (per numeri negativi il fattoriale del valore assoluto): potremmo codificare un sottoprogramma del tipo:

```
      INTEGER FUNCTION FATTOR (N)
      IF(N)1,5,1
5     FATTOR = 1
      GO TO 4
1     FATTOR = ABS(N)
      M = ABS(N)
2     M = M—1
      IF(M) 4,4,3
3     FATTOR = FATTOR*M
      GOTO 2
4     RETURN
      END
```

E' possibile porre nel sottoprogramma più di un'istruzione RETURN in modo da provocare il ritorno al programma principale da più punti; è possibile però una sola END.

Sempre nel caso di prima, supponiamo che di ogni numero si voglia anche la radice e nel caso sia negativo, la radice cambiata di segno del valore assoluto:

```
      FUNCTION RADX(N)
      IF(N)1,2,2
1     RADX = —SQRT(—N)
      RETURN
2     RADX = SQRT(N)
      RETURN
      END
```

Il programma principale e le funzioni che utilizza, nonchè i sottoprogrammi, presenteranno una struttura del tipo:

```
C   PROGRAMMA PRINCIPALE...
    . . .
    . . .
    READ(5,...) N
    . . .
    . . .
    I = FATTOR (N)
    . . .
    Y = RADX(N) + Z + SQRT(X)...
    . . .
    . . .
    STOP
    END

C   SOTTOPROGRAMMA FATTOR
    INTEGER FUNCTION FATTOR(N)
    . . .
    . . .
    RETURN
    END

C   SOTTOPROGRAMMA RADX
    FUNCTION RADX(N)
    . . .
    RETURN
    . . .
    RETURN
    END
```

E' possibile passare ad un sottoprogramma FUNCTION anche dei vettori o delle matrici; in tal caso essi dovranno comparire tra gli argomenti e non solo: sarà opportuno che tra gli argomenti compaiano anche le rispettive dimensioni.

Ad esempio supponiamo di voler totalizzare gli elementi di alcune matrici trattate nel programma principale: possiamo preparare un sottoprogramma FUNCTION del tipo:

```
    FUNCTION SOM(A,N,M)
    DIMENSION A(N,M)
    SOM = 0
    DO 1 J = 1,N
    DO 1 K = 1,M
1   SOM = SOM + A(J,K)
    RETURN
    END
```

dove A è la generica matrice di cui si vogliono totalizzare gli elementi, N ed M le sue dimensioni. Come si vede SOM ha validità per matrici di ordine qualunque essendo stato fatto un uso, per così dire, parametrizzato delle dimensioni. Proprio per questo motivo, dimensioni così dichiarate nei sottoprogrammi prendono il nome di **adjustable dimension** *(dimensioni aggiustabili)*: si tenga presente che un sottoprogramma come SOM potrà comunque venir compilato, ma l'esecuzione sarà possibile solo qualora sia presente nel programma principale una definizione esplicita da assegnare poi, all'atto della chiamata della funzione, alle dimensioni aggiustabili N ed M (come del resto a tutti gli altri argomenti). Il programma principale e il sottoprogramma saranno, almeno nelle linee generali:

```
DIMENSION MATR(10,10),X(100,20),Y(10,100)
. . .
. . .
TOT = SOM(MATR,10,10)
. . .
K = 20
TOT = TOT + SOM(X,100,K)
. . .
K = 10
L = 100
TOT = TOT + SOM(Y,K,L)
. . .
. . .
STOP
END
FUNCTION SOM(A,N,M)
. . .
. . .
. . .
RETURN
END
```

Si tenga presente che sarebbe stato possibile scrivere, ad esempio:

```
FUNCTION SOM(A)
DIMENSION A(10,5)
```

in tal caso non si parla di dimensioni aggiustabili, bensì fisse. Il sottoprogramma funzionerà benissimo, limitando la totalizzazione alle prime 10 righe e 5 colonne di ogni matrice; sarà ovviamente un errore, invece, utilizzarlo per matrici di dimensioni diverse da quel 10 per 5.

L'uso delle dimensioni fisse raramente si presenta vantaggioso: utilizzando le dimensioni aggiustabili eventuali modifiche di dimensione riguardano solo

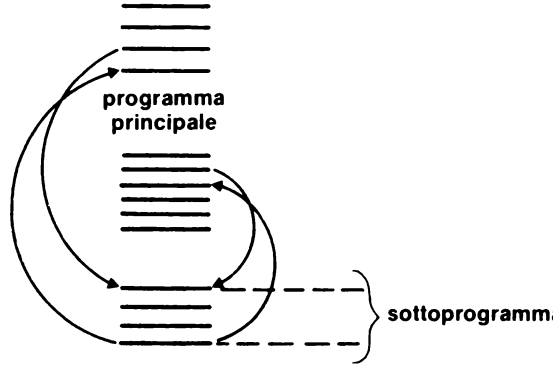
il programma principale e soprattutto il sottoprogramma può risultare utilizzabile in un gran numero di casi diversi.

### 5.4 Sottoprogramma SUBROUTINE

Tutte le funzioni viste finora sono costituite da veri e propri sottoprogrammi: le prime, quelle intrinseche, da sottoprogrammi di sistema, le altre da sottoprogrammi codificati dall'utente.

In ogni caso si tratta di una o più istruzioni che possiamo immaginare prendano il posto di quella di richiamo nel programma principale. Questo viene eseguito istruzione per istruzione e, al richiamo del sottoprogramma, il controllo passa a quest'ultimo, ne viene fatta l'esecuzione completa ed alla fine il controllo torna al programma principale all'istruzione immediatamente seguente quella di richiamo.

Ciò avviene in generale per tutti i sottoprogrammi.




E' possibile codificare dei sottoprogrammi che svolgono vari compiti ed operazioni e che, a differenza delle funzioni, possono restituire più di un valore al programma principale: si tratta delle **SUBROUTINE**. Queste hanno in comune con il programma principale tutte le variabili dichiarate come argomenti o tutto ciò che è stato dichiarato esplicitamente in comune per mezzo dell'istruzione dichiarativa di COMMON che vedremo tra breve.

Poichè il nome della SUBROUTINE non è legato al valore (o al risultato) di ritorno, come invece accade per le funzioni, non ha senso parlare di tipo della SUBROUTINE: il nome è un nome di sottoprogramma che viene utilizzato per la definizione e per il richiamo e non per assegnare un valore.

Il formato generale è:

codifica delle varie funzionalità che la subroutine deve svolgere

```
SUBROUTINE nomes(arg1 ,arg2 ,..,argn)
...
...
...
...
...
RETURN
END
```

La chiamata, da programma principale, deve avvenire per mezzo dell'istruzione CALL della forma:

CALL **nomes (argom$_1$ ,argom$_2$ ,...,argom$_n$)**

ancora gli argomenti verranno riconosciuti sulla base dell'ordine e non del loro nome, ancora sarà possibile mettere tra gli argomenti vettori o matrici, nel qual caso bisognerà porre attenzione alle dimensioni relative (aggiustabili o fisse).

Ad esempio:

```
DIMENSION A(10,100),B(20),C(100,100)
DOUBLE PRECISION D
X=...
Y=...
Z=...
I=20
CALL SUB1(A,10,100,X,Y,Z)
CALL SUB2(B,I)
CALL SUB3(C,D,100,100,X,Y,Z)
CALL SUB4(B)
STOP
END

SUBROUTINE SUB1(P,N,M,Q,R,S)
DIMENSION P(N,M)
...
...
RETURN
END

SUBROUTINE SUB2(P,K)
DIMENSION P(K)
...
...
RETURN
END
```

```
SUBROUTINE SUB3(C,D,J,L,Q,T,W)
DIMENSION C(J,L)
...
...
RETURN
END

SUBROUTINE SUB4(V)
DIMENSION V(20)
...
RETURN
END
```

| Subroutine | Dimensionamento | Nome Argomento nella Subroutine | Nome o Val. Argomento nel programma principale | Note |
|---|---|---|---|---|
| SUB 1 | aggiustabile per P | P<br>N<br>M<br>Q<br>R<br>S | A<br>10<br>100<br>X<br>Y<br>Z | dimensioni di A |
| SUB 2 | aggiustabile per P | P<br>K | B<br>I | = 20 che è la dimens. di B |
| SUB 3 | aggiustabile per C | C<br>D<br>J<br>L<br>Q<br>T<br>W | C<br>D<br>100<br>100<br>X<br>Y<br>Z | dimensioni di C |
| SUB 4 | fisso | V | B | |

*Tab. 23*

La dichiarazione dei nomi degli argomenti nella lista che segue il nome della SUBROUTINE, non provoca la generazione di un'apposita area di memoria: ad esempio la dichiarazione di P, nella SUB1 dell'esempio precedente, sta

solo a significare che questo nome verrà usato nella SUBROUTINE per fare però riferimento ad A che è il nome dichiarato nella posizione corrispondente nella lista della CALL. Per questo motivo, tali argomenti prendono solitamente il nome di *dummy arguments*. Va tenuto presente, relativamente ad essi, la limitazione che non possono apparire in una lista di NAMELIST.

### 5.5 Entry-point nelle SUBROUTINE

Come nei sottoprogrammi già visti, anche nelle SUBROUTINE è possibile avere più RETURN (ed un'unica END): in più ora è possibile che anche i punti d'entrata nella SUBROUTINE siano multipli: ciò è ottenuto per mezzo dell'istruzione ENTRY. La logica di utilizzo è assai semplice: si pensi di sostituire alla parola SUBROUTINE la parola ENTRY ed il nuovo **entry-point** è definito.

Il formato di tale istruzione di entrata nella SUBROUTINE è dunque:

ENTRY nome(**$arg_1$ ,$arg_2$ ,...,$arg_n$**)

Ogni SUBROUTINE deve iniziare con una definizione

SUBROUTINE **nome...**

ma potrà contenere quante ENTRY si desiderino.

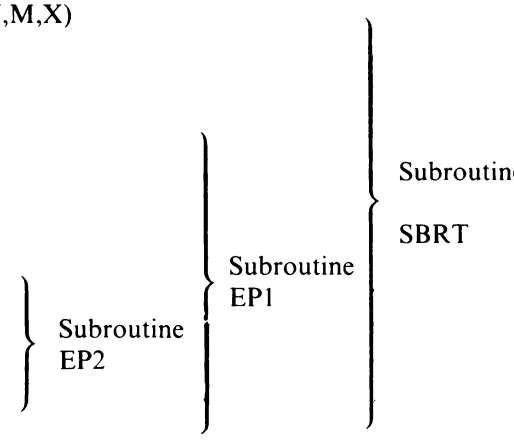
Il nome e il numero degli argomenti non devono essere necessariamente gli stessi nella definizione di SUBROUTINE e nelle successive definizioni di punti di ingresso: è importante, invece ed ovviamente, che tutti gli argomenti che vengono utilizzati da un certo punto in poi nella SUBROUTINE siano stati elencati nella ENTRY relativa.

Tenendo presente che ogni ENTRY può essere eseguita come un sottoprogramma a sé stante e che, allora, ognuna ha una sua lista di argomenti, si può avere una situazione del tipo:

```
DIMENSION A(10,100)
X=...
Y=...
Z=...
...
CALL SBRT(A,10,100,X)
...
CALL EP1(X,Y,Z)
...
CALL EP2(X,Y)
```

```
...
STOP
END

SUBROUTINE SBRT(A,N,M,X)
...
...
ENTRY EP1(X,W,T)
...
...
ENTRY EP2(A,B)
...
...
RETURN
END
```




### 5.6 L'istruzione COMMON

A volte può risultare assai utile assegnare la stessa area di memoria ad una o più delle variabili che compaiono nel programma principale e che devono essere utilizzate nei sottoprogrammi da questo richiamabili, senza farle apparire nella lista degli argomenti. Allo scopo si utilizza lo statement COMMON: esso ha la funzione di assegnare ad una zona di memoria particolare, che prende il nome di **common-area**, le variabili ad esso associate.

Se in un programma (ad esempio il programma principale) viene scritto

COMMON A1,A2,...,AN

le variabili A1,A2,...,AN vanno a collocarsi ordinatamente in quell'area; se poi in un altro sottoprogramma (ad esempio una SUBROUTINE richiamabile) viene scritto:

COMMON B1,B2,...,BN

tali variabili vengono ancora riferite ordinatamente alle prime N zone di *common area*, con la conseguenza che risultano essere le stesse variabili, a tutti gli effetti, che nel programma principale sono state chiamate A1, A2,...AN.

Quanto detto vale non solo per le SUBROUTINE, ma in generale per qualunque sottoprogramma.

Ad esempio:

**MAIN**

```
DIMENSION A(10,10)
...
...
COMMON B, C(3,4), V (20), I
X = ...
Y = ...
Z = ...
...
CALL SUBA (A, 10, 20, X)
...
CALL SUBB
...
X = FUNZ (Y)
...
STOP
END
```

**SOTTOPROGRAMMI**

```
SUBROUTINE SUBA (AM, I, J, R)
DIMENSION AM(I,I)
...
...
RETURN
END

SUBROUTINE SUBB
COMMON A, B (3,4) V (20), N
...
...
RETURN
END

FUNCTION FUNZ (W)
COMMON B1, B2 (3,4)
...
...
RETURN
END
```

per quanto riguarda SUBA il passaggio dei valori tra programma principale e sottoprogramma avviene solamente attraverso la lista degli argomenti; per quanto riguarda SUBB, invece, le variabili comuni sono solo quelle dichiarate in COMMON; nel caso, poi, di FUNZ vi sono sia variabili in comune (grazie alla lista degli argomenti), sia altre, (grazie alla loro enunciazione in COMMON). Dall'esempio si nota che lo statement COMMON è servito nel programma principale anche per dimensionare C e V: in generale la dichiarazione di COMMON può essere anche una dichiarazione di dimensioni. Non è possibile, invece, ed ovviamente, riferire aree di COMMON con dimensioni aggiustabili.

Vanno fatte alcune considerazioni:

a) perchè un enunciato COMMON abbia senso e sia efficace, dovrà essere presente in almeno due unità di programma;

b) l'ordine con cui le variabili sono indicate stabilisce il criterio di corrispondenza nelle diverse unità di programma;

c) le variabili presenti nelle diverse liste, essendo riferite a diverse unità di programma, possono anche avere nomi uguali, senza per questo generare alcuna possibilità di equivoco;

d) non è detto che le variabili presenti nei diversi COMMON debbano individuare complessivamente sempre zone di memoria della stessa estensione.

Così, se per esempio avessimo:

```
C   PROGRAMMA PRINCIPALE
    . . .
    . . .
    DIMENSION A(5)
    COMMON X,A
    . . .
    . . .

C   SOTTOPROGRAMMA
    . . .
    . . .
    DIMENSION S(3)
    COMMON R,R1,S,T
    . . .
    . . .
```

la corrispondenza sarà tale che col nome R si accede al contenuto della variabile X, con R1 ad A(1), con S(1) ad A(2), con S(2) ad A(3), con S(3) ad A(4), con T ad A(5).

Come abbiamo detto, l'enunciato COMMON può essere utilizzato in due o più unità di programma. E' quindi del tutto legittimo scrivere:

```
C   PROGRAMMA PRINCIPALE
    . . .
    DIMENSION A(5)
    COMMON X,A
    . . .

C   SOTTOPROGRAMMA1
    . . .
    DIMENSION S(3)
    COMMON R,R1,S,T
    . . .
    . . .

C   SOTTOPROGRAMMA2
    . . .
    COMMON S,T
    . . .
```

dove la corrispondenza per il primo sottoprogramma è quella preventivamente vista, mentre per quanto riguarda il secondo, avremo che con S si accede al contenuto di X, con T al contenuto di A(1).

Supponiamo ora che il nostro sottoprogramma debba accedere sì a delle aree in comune col programma principale, ma S debba essere in corrispondenza con A(2) e T con A(5). Dato che la corrispondenza parte sempre dall'inizio dell'area dichiarata COMMON, per ottenere quanto da noi desiderato occorrerà introdurre nell'area COMMON del secondo sottoprogramma delle aree fittizie che servono a stabilire esattamente la corrispondenza. Così dovremo scrivere:

```
C   SOTTOPROGRAMMA 2
    ...
    COMMON D1,D2,S,D3,D4,T
    . . .
```

E' chiaro che se l'area COMMON è molto estesa, un procedimento del genere può essere abbastanza noioso da gestire e può comportare facilmente introduzione di errori.

Per evitare problemi di questo tipo è stato introdotto quello che prende nome di **common labellato**.

Questo consiste in un enunciato COMMON nel quale ad ogni gruppo di aree COMMON si associa un nome. Quindi in diverse unità di programma sarà possibile utilizzare singoli blocchi COMMON, senza doversi preoccupare di introdurre variabili fittizie per tener conto delle aree non utilizzate. Il nome del blocco COMMON lo si scrive semplicemente ponendolo tra due barre, seguito dalle variabili che compongono il blocco e i cui nomi sono separati tramite virgole. L'unica restrizione consiste nel fatto che, all'interno di tutte le unità di programma che lo usano, l'estensione di un blocco COMMON di dato nome, dovrà essere la stessa.

Il formato completo dello statement COMMON è il seguente:

COMMON/**nomegruppo1/lista1/nomegruppo2/lista2...**

Si noti che è possibile assegnare a gruppi di variabili (**lista1, lista2,...**) nomi di gruppo (**nomegruppo1,nomegruppo2...**): essi vengono assegnati a tutta la zona di memoria destinata a contenere la lista corrispondente di variabili. Si dice che esse si trovano in una area **common con label**, cioè ancora in memoria comune, ma (per il fatto che questa è stata battezzata con un nome (**gruppo1...**)) con **label**. Non esiste una sostanziale differenza rispetto al normale utilizzo dell'istruzione di COMMON già visto, se non per il fatto che la **label** definita può essere utilizzata per scopi particolari (non del tutto inerenti

specificatamente il FORTRAN) in quanto conosciuta dal sistema anche al di fuori del programma (1).

Comunque, per tutto quanto detto, porre una variabile in COMMON (con o senza label) piuttosto che nella lista degli argomenti, non cambia sostanzialmente la logica di programmazione ed infatti, agli effetti pratici, non esistono differenze se si eccettua la disposizione in memoria delle variabili. Nel caso si utilizzi la dichiarazione nella lista degli argomenti, le variabili si trovano nella memoria del programma in cui sono definite ed ogni riferimento ad esse, anche da un sottoprogramma è un riferimento a quelle aree di memoria; nel caso invece esse vengano dichiarate in COMMON si trovano nella common area, o con label o senza (in questo caso si parla di *blank-common*), cioè in una zona di memoria indipendente dai vari programmi.

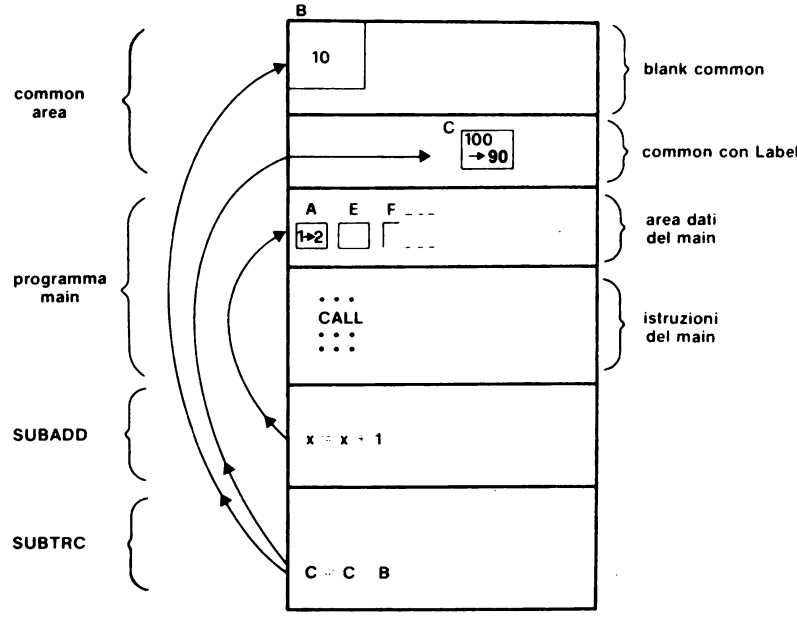
Così, ad esempio, dato il programma seguente con le sue SUBROUTINE, la disposizione in memoria sarà del tipo dello schema:

```
COMMON B/X1/C
.....
A=1
B=10
C=100
D=...
E=...
F=...
CALL SUBADD(A)
CALL SUBTRC
WRITE...
...
...
END

SUBROUTINE SUBADD(X)
X=X+1
RETURN
END

SUBROUTINE SUBTRC
COMMON B/X1/C
C=C—B
RETURN
...
...
END
```

(1) E' questo il caso in cui, ad esempio, si desidera mettere in comune una certa area di memoria tra programmi scritti in linguaggi diversi.




Ma le possibilità del common labellato sono ancora più ampie: è possibile infatti scrivere:

COMMON/UNO/X,Y,Z/DUE/A,B,C/UNO/W,R,S

ed è come se fosse stato scritto (nell'ordine):

COMMON/UNO/X,Y,Z,W,R,S/DUE/A,B,C

Ed infine in uno stesso enunciato COMMON è possibile inserire COMMON semplici e COMMON labellati a nostra discrezione, purchè, se si stanno scrivendo dei COMMON labellati e si intende poi elencare anche dei COMMON semplici, questi si separino tramite due barre consecutive:

COMMON A1,A2,A3/UNO/A,B,C/DUE/X,Y,Z//R,S,T/DUE/V,W

quest'unica scrittura è del tutto equivalente all'insieme dei due enunciati COMMON:

```
COMMON A1,A2,A3,R,S,T
COMMON/UNO/A,B,C/DUE/X,Y,Z,V,W
```

Abbiamo visto che le FUNCTION e le SUBROUTINE presentano un insieme di argomenti formali che verranno poi effettivamente specificati tramite l'istruzione che richiamerà i sottoprogrammi.

D'altra parte ogni compilatore FORTRAN accetta solo un certo numero di argomenti, così che, nel caso essi superino per qualche motivo tale valore massimo, ci troveremmo nell'impossibilità di scrivere il nostro sottoprogramma.

L'enunciato COMMON ci permette di superare questo ostacolo: infatti ogni variabile dichiarata in COMMON, viene eliminata dalla lista degli argomenti. Di conseguenza si possono dichiarare in COMMON anche tutti gli argomenti di una SUBROUTINE, mentre nel caso delle FUNCTION dovrà sempre restarne almeno uno esplicitamente dichiarato.

Così nel caso avessimo:

```
C PROGRAMMA PRINCIPALE          C SOTTOPROGRAMMA
  DIMENSION A (10), B (20)        SUBROUTINE SUB1 (R, S,
                                  T)
  ...                             DIMENSION T (10)
  CALL SUB1 (X, Y, A)             ...
  ...                             ...
  ...                             RETURN
  ...                             ...
  END                             END
```

potremmo equivalentemente scrivere:

```
C PROGRAMMA PRINCIPALE          C SOTTOPROGRAMMA
  DIMENSION B (20)                SUBROUTINE SUB1
  COMMON X, Y, A (10)             COMMON R, S, T (10)
  ...                             ...
  ...                             ...
  CALL SUB1                       RETURN
  ...                             ...
  ...                             ...
  END                             END
```

### 5.7 L'istruzione EXTERNAL

Resta da analizzare un'ultima istruzione relativa all'utilizzo di sottoprogrammi: la dichiarazione EXTERNAL.

Qualora si intendano richiamare dei sottoprogrammi contenenti, *tra gli argomenti*, nomi di altri sottoprogrammi è necessario dichiarare tali nomi nello statement EXTERNAL il cui formato è del tipo:

EXTERNAL **nome 1,nome2,...,nomen**

dove **nome1,nome2,...,nomen**, sono, appunto, i nomi di quei sottoprogrammi. Si tratta di uno statement non eseguibile e di tipo dichiarativo: dunque, va premesso a qualunque richiamo dei sottoprogrammi relativi.

Se, ad esempio, si dovesse calcolare la radice di un valore di un dato in input se esso è positivo, oppure il suo quadrato se è negativo, potremmo codificare un programma del tipo:

```
      EXTERNAL SQRT, QUADR
    1 READ(5,100,END=9) X
  100 FORMAT(...)
      IF(X)3,3,2
    3 Y=FUNZ(QUADR,X)
      GO TO 1
    2 Y=FUNZ(SQRT,X)
      GO TO 1
      ...
      ...
      ...
    9 STOP
      END


      FUNCTION FUNZ(A,B)
      FUNZ=A(B)
      RETURN
      END

      FUNCTION QUADR(C)
      QUADR=C**2
      RETURN
      END
```

Come si vede i sottoprogrammi possono essere tanto di sistema quanto dell'utente.

Se con l'esempio precedente, a causa della sua semplicità, può sfuggire l'utilità dell'uso di una funzione, va però tenuto presente che nel caso si renda necessario il calcolo ripetuto di una certa formula, l'uso di un'unica *function parametrizzata* conduce ad un risparmio notevole.

Nell'esempio che segue sia FTRG il nome di tale funzione parametrizzata: si supponga di dover calcolare, per un vettore X(100) il valore dato dalle formule seguenti:

$$p_1 = \begin{cases} \dfrac{a}{b + \text{sen } x} & \text{se } x > 0 \\ 0 & \text{se } x \leq 0 \end{cases}$$

ed in un altro caso

$$p_2 = \begin{cases} \dfrac{a}{b + \cos x} & \text{se } x > 0 \\ 0 & \text{se } x \leq 0 \end{cases}$$

ed in un altro caso ancora

$$p_3 = \begin{cases} \dfrac{a}{b + \text{tang } x} & \text{se } x > 0 \\ 0 & \text{se } x \leq 0 \end{cases}$$

ed infine in un ultimo caso

$$p_4 = \begin{cases} \dfrac{a}{b + \dfrac{1}{x} + \dfrac{1}{x^2} + \ldots + \dfrac{1}{x^{100}}} & \text{se } x > 0 \\ 0 & \text{se } x \leq 0 \end{cases}$$

dove a e b sono dati di input.

Si può raggiungere un certo grado di semplicità e soprattutto di generalità codificando un sottoprogramma ZETA che contenga una formula del tipo

$$P = Q/(R + FTRG(S))$$

dove la funzione FTRG sarà, volta per volta, a seconda della chiamata

SIN COS TAN SINVR

dove la funzione SINVR = $\dfrac{1}{x} + \dfrac{1}{x_2} + \ldots + \dfrac{1}{x^{100}}$

sarà un sottoprogramma FUNCTION dell'utente.

Il programma potrà essere del tipo:

```
      EXTERNAL SIN,COS,TAN,SINVR
      DIMENSION X(100)
      READ(5,100)A,B,(X(I),I = 1,100)
  100 FORMAT(...)
      ...
      DO 9 I = 1,100
      IF(X(I))7,7,1
    7 P1 = 0
      P2 = 0
      P3 = 0
      P4 = 0
      GO TO 9
    1 Z = X(I)
      CALL ZETA(P1,A,B,SIN,Z)
      CALL ZETA (P2,A,B,COS,Z)
      CALL ZETA(P3,A,B,TAN,Z)
      CALL ZETA(P4,A,B,SINVR,Z)
      WRITE (...)
      ...
      ...
    9 CONTINUE
      ...
      ...
      STOP
      END


      SUBROUTINE
      ZETA(P,A1,A2,FTRG,A3)   (1)
      P = A1/(A2 + FTRG(A3))
      RETURN
      END

      FUNCTION SINVR(H)
      SINVR = 0
      DO 1 K = 1,100
    1 SINVR = SINVR + 1/H**K
      RETURN
      END
```

(1) Anche ZETA poteva venir codificata come sottoprogramma FUNCTION.

CAPITOLO 6

# ALCUNE ISTRUZIONI AUSILIARIE

## 6.1 L'istruzione EQUIVALENCE

Da quanto presentato sinora, dovrebbe essere piuttosto chiaro che, ogni qualvolta si faccia riferimento ad un nome di variabile, esso sarà associato ad una corrispondente area di memoria, e nomi di variabili diversi corrisponderanno ad aree di memoria diverse.

Una possibilità di modificare questo stato di cose è fornito dall'enunciato EQUIVALENCE: con esso noi stabiliamo sostanzialmente l'identità fisica delle locazioni di memoria associate a diversi nomi di variabili. Da un punto di vista leggermente differente, possiamo dire che ridefiniamo alcune locazioni di memoria.

Prima di vedere più in particolare che cosa tutto questo significhi, osserviamo come si presenta l'enunciato EQUIVALENCE:

EQUIVALENCE **(lista-1),(lista-2),...,(lista-n)**

dove lista-i rappresenta un raggruppamento di 2 o più nomi di variabili o di elementi di array, comunque combinati.

Se allora scriviamo:

EQUIVALENCE(A,B,C)

il risultato sarà che la *stessa* parola di memoria sarà richiamabile con il nome A, oppure B, oppure C.

Occorrerà avere ben presente il fatto elementare che, in una locazione di memoria, può esistere un solo insieme di informazioni alla volta, così che se le aree A,B,C si riferiscono a dati letti man mano dall'esterno, nel momento in cui fosse stato letto il dato e memorizzato in A, *non* si potrebbero avere contemporaneamente a disposizione i dati B e C. E viceversa, leggendo B, le

precedenti informazioni di A verrebbero ricoperte. Quindi, usando tramite un enunciato EQUIVALENCE, nomi diversi relativamente ad un'area destinata a contenere informazioni diverse, questi non dovranno *mai* essere usati contemporaneamente.

E' il caso di spendere alcune parole per quanto riguarda l'enunciato EQUIVALENCE coinvolgente variabili ed elementi di array.

Se in un programma si facesse uso di tre aree A, B, C, queste sarebbero situate in tre locazioni di memoria non determinabili a priori né per quanto riguarda la loro posizione, né per quanto riguarda il loro ordine od il loro essere consecutive.

Però scrivendo ad esempio:

```
DIMENSION AR(3)
EQUIVALENCE(AR(1),A),(AR(2),B),(AR(3),C)
```

le aree A, B, C verrebbero riservate consecutivamente nell'ordine ora scritto: questo proprio per l'equivalenza stabilita con l'enunciato e per il fatto che, come abbiamo visto a suo tempo, un array è un insieme *consecutivo* di elementi.

Una cosa un pochino diversa è quella riguardante l'enunciato EQUIVALENCE tra elementi di array: infatti l'equivalenza dichiarata tra gli elementi di due o più array, si *propaga* a tutta la parte che in fase di memorizzazione si trova in comune.

Consideriamo infatti ad esempio l'enunciato:

```
DIMENSION A(3,5),B(2,10),C(12)
```

Se non si specifica alcun enunciato EQUIVALENCE, gli elementi sono presenti consecutivamente in memoria come:

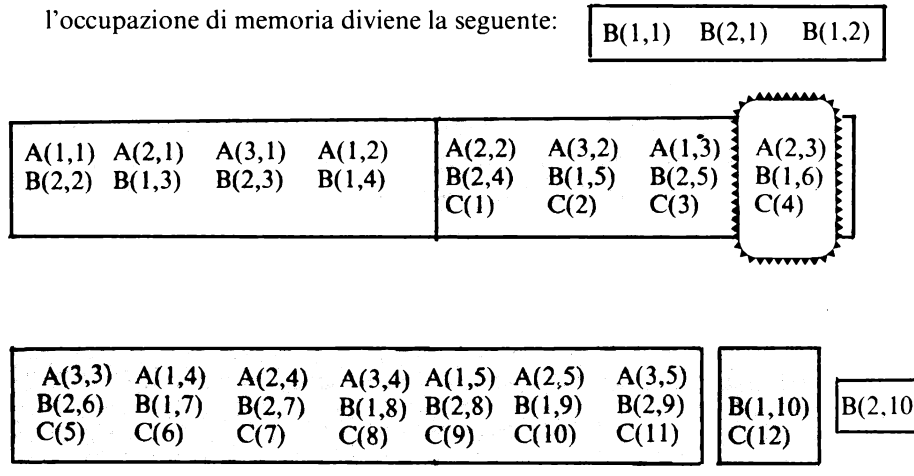
```
A(1,1) A(2,1) A(3,1) A(1,2) A(2,2) A(3,2) A(1,3) A(2,3) A(3,3)
A(1,4) A(2,4) A(3,4) A(1,5) A(2,5) A(3,5) B(1,1) B(2,1) B(1,2)
B(2,2) B(1,3) B(2,3) B(1,4) B(2,4) B(1,5) B(2,5) B(1,6) B(2,6)
B(1,7) B(2,7  B(1,8) B(2,8) B(1,9) B(2,9) B(1,10)B(2,10)C(1)
C(2)   C(3)   C(4)   C(5)   C(6)   C(7)   C(8)   C(9)   C(10)
C(11)  C(12)
```

Come si vede, vengono occupate 47 locazioni di memoria: 15 per l'array A, 20 per l'array B e 12 per l'array C.

Utilizzando l'enunciato EQUIVALENCE e ponendo, ad esempio,

EQUIVALENCE (A(2,3),B(1,6),C(4))

l'occupazione di memoria diviene la seguente:




Come si può notare, l'effetto più appariscente consiste in una notevole riduzione della memoria necessaria per contenere i 3 array (20 posizioni) ed inoltre che l'enunciato EQUIVALENCE fa sì che risultino equivalenti non solo gli elementi dichiarati nell'enunciato, ma anche tutti quelli presenti nell'area tratteggiata, per ciò che riguarda gli array A, B, C, nell'area punteggiata, per ciò che riguarda gli array A e B, nell'area grigia per ciò che riguarda gli array B e C.

Così sarebbe stato del tutto analogo dichiarare:

EQUIVALENCE (A(2,2),B(2,4),C(1))

oppure

EQUIVALENCE (A(3,2),B(1,5),C(2))

oppure

EQUIVALENCE (A(3,5),B(2,9),C(11)

Una cosa da tener presente nel caso si indichino delle quivalenze tra array, è che se l'elemento che si considera è il primo dell'array, si può scrivere anche il solo nome dell'array. Così avremmo anche potuto indicare:

EQUIVALENCE (A(2,2),B(2,4),C)

Usando questo enunciato occorre prestare abbastanza attenzione a non dichiarare equivalenze inconsistenti. Infatti se, con l'esempio precedente, avessimo scritto:

```
EQUIVALENCE (A(2,3),B(1,6),C(4)),(B(1,8),C(2))
```

ci saremmo trovati nella condizione assurda che B(1,8) avrebbe dovuto essere equivalente a C(2) per quanto dichiarato nella seconda parte dell'EQUIVALENCE e, contemporaneamente avrebbe dovuto concidere con C(8) come conseguenza della prima parte dell'enunciato stesso.

Uno degli scopi per cui si può usare questo enunciato di specificazione è quello di rendere equivalenti array di dimensioni diverse, ed in particolare di renderli equivalenti ad un array monodimensionale, ottenendo così quello che è noto come **linearizzazione** di un array.

Può infatti capitare di dover svolgere a volte delle operazioni che, su un array pluridimensionale possono presentare una certa difficoltà, mentre sarebbero molto più semplici su un array lineare.

Così, se dovessimo eseguire tali operazioni su un array definito da:

```
DIMENSION A(10,20,30)
```

ci basterebbe ridefinire tale array come monodimensionale. Questo si può ottenere introducendo un array AL che abbia complessivamente lo stesso numero di elementi e poi dichiarandolo equivalente al primo:

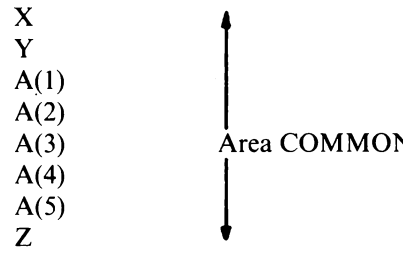
```
DIMENSION A(10,20,30),AL(6000)
EQUIVALENCE(A,AL)
```

Così, a seconda del tipo di operazioni da eseguire, si potrà rispettivamente usare come nome A, oppure AL.

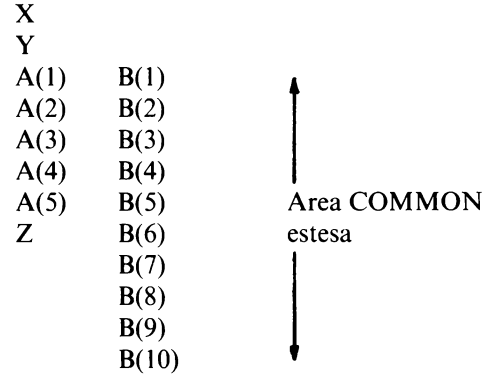
Inoltre è possibile la coesistenza dell'enunciato EQUIVALENCE con l'enunciato COMMON. In tal caso il risultato è che l'area COMMON può risultare ampliata. Consideriamo infatti un esempio:

```
DIMENSION A(5),B(10)
EQUIVALENCE(A,B)
COMMON X,Y,A,Z
```

Prescindendo dall'enunciato EQUIVALENCE, si avrebbe:



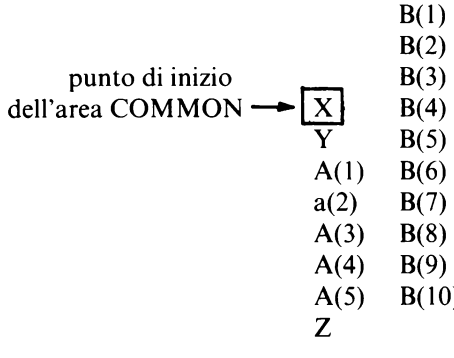

Grazie all'effetto dell'EQUIVALENCE la situazione diventa:




Ricordiamo che è possibile estendere l'area COMMON solo in avanti, rispetto al suo punto iniziale, quindi *non* è legittimo un insieme di enunciati come il seguente:

```
DIMENSION A(5),B(10)
EQUIVALENCE (A(5),B(10))
COMMON X, Y, A, Z
```

che comporterebbe:



## 6.2 L'istruzione DATA

Fra le varie possibilità offerte dal linguaggio FORTRAN, esiste anche quella di assegnare, durante la fase di compilazione, dei valori a nostra scelta a campi variabili.

Per ottenere questo risultato si utilizza un enunciato di specificazione: l'enunciato DATA, il cui formato è:

DATA **lista-1/cost-1/,lista-2/cost-2/,...,lista-n/cost-n/**

dove: **lista-i** rappresenta una lista di variabili, elementi di array od array che si desiderano inizializzare;
**cost-i** è una lista di costanti che rappresentano i valori che si vogliono assegnare; tale lista di costanti deve esssere racchiusa tra due barre.

I nomi presenti nella lista di variabili devono essere separati tra loro tramite virgole, e, quando si faccia riferimento ad elementi di array, è permessa la struttura del DO implicito; inoltre gli indici possono avere una qualunque delle forme:

c
v
c*v
c*v+k
c*v—k
v+k
v—k

dove c e k sono degli interi positivi e v è un nome di variabile.

Inoltre nel caso del DO implicito, i valori iniziale, finale e di incremento della variabile di controllo devono essere delle costanti intere.

Per quanto riguarda la lista delle costanti, va detto che esse vanno separate le une dalle altre tramite virgole, si riferiscono ordinatamente alle variabili espresse nella lista precedente e, nel caso di più costanti consecutive uguali, è permesso l'uso di un fattore moltiplicativo (di ripetizione) nella forma n*c dove n è un numero intero e c è il valore della costante.

Così, volendo inizializzare, in fase di compilazione, le variabli A,B rispettivamente ai valori 0 e 1 e l'array C(2,8) a zero, potremmo scrivere:

```
DATA A,B/0,1/,((C(I,J),I=1,2),J=1,8)/16*0./
```

o, equivalentemente:

```
DATA A/0/,B/1/,((C(I,J),I = 1,2),J = 1,8)/16*0./
```

oppure:

```
DATA A,B,((C(I,J),I = 1,2),J = 1,8)/0,1,16*0./
```

oppure:

```
DATA A,B,C/0,1,16*0./
```

Ovviamente con l'enunciato DATA è possibile eseguire qualunque tipo di inizializzazione, quindi anche con costanti logiche, Hollerith, complesse, etc.

Una osservazione va fatta in modo esplicito: poichè le assegnazioni con l'enunciato DATA vengono fatte in fase di compilazione, gli indici di un elemento di array devono essere costanti o variabili definite tramite un DO implicito e *non* possono essere variabili definite tramite un DATA.

Quindi *non* è possibile scrivere:

```
DATA I/2/,A(I)/7.5/
```

volendo attribuire ad A(2) il valore 7.5.

Riguardo l'uso del DO implicito, va ricordato che è anche possibile, in caso di necessità, assegnare dei valori a determinati elementi e non a tutti.

Ad esempio:

```
DATA (A(I),I = 1,30,2)/15*0./,(A(I),I = 2,30,2)/15*1./
```

permette di assegnare il valore 0. agli elementi di posto dispari dell'array ed il valore 1. a quelli di posto pari.

Va altresì ricordato che in ogni programma è possibile ripetere l'enunciato DATA più volte.

### 6.3 Sottoprogramma BLOCK DATA

Può accadere che, in diversi programmi o sottoprogrammi, sia necessario eseguire identiche inizializzazioni. Piuttosto che ripetere in ogni programma le inizializzazioni tramite l'enunciato DATA, è possibile riunirle in un unico sottoprogramma. Tale sottoprogramma è il BLOCK DATA: esso serve per

inizializzare aree COMMON labellate e può contenere solo gli enunciati non eseguibili (vedi Appendice B): DATA, EQUIVALENCE, COMMON, DIMENSION, IMPLICIT e dichiarazioni di tipo per variabili ed array. Esso va posto come prima parte della codifica, prima del programma principale, e ad esso faranno riferimento tutte le unità di programma: programma principale stesso, eventuali sottoprogrammi FUNCTION, eventuali SUBROUTINE.

Il formato del sottoprogramma BLOCK DATA è:

```
BLOCK DATA [nome]
.....
.....
END
```

dove: **nome** è un insieme da 1 a 6 caratteri identificante il sottoprogramma BLOCK DATA. Esso può venire omesso, in tal caso verrà assegnato un nome standard dipendente dal compilatore.

Nel nucleo del sottoprogramma, tra BLOCK DATA ed END, andranno poste le dichiarazioni, le inizializzazioni ecc.

Volendo, ad esempio, azzerare un array A di 10 elementi ed inizializzare a 1 le aree I e K, potremmo scrivere:

```
BLOCK DATA INIZ
DIMENSION A(10)
COMMON /X/A/Y/I,K
DATA A/10*0./,I,K/2*1/
END
```

Va esplicitamente osservato che non esiste l'istruzione RETURN, in quanto tale sottoprogramma non è richiamabile in fase esecutiva, ma può essere solo utilizzato durante la fase di costruzione del programma eseguibile.

# PARTE III
# FORTRAN 77

CAPITOLO 1

# INTRODUZIONE

## 1.1 Alcune generalità sul FORTRAN 77

Come si è già detto il FORTRAN nasce come linguaggio di tipo procedurale, esso infatti è orientato ai problemi tecnico-scientifici ed al calcolo numerico.

Tuttavia l'uso sempre più diffuso del linguaggio e le applicazioni di carattere sempre più vario, hanno condotto ad una evoluzione tale da far sentire la necessità di un nuovo standard.

Negli ultimi anni già erano stati previsti, dalle case costruttrici, certi ampliamenti particolari e di notevole utilità, tali da permettere, ad esempio, il trattamento di stringhe di caratteri, o una più agevole gestione dei file.

Recentemente è stato definito un nuovo standard del linguaggio, che prende nome FORTRAN 77, e che ha il preciso scopo di rendere standard, appunto, quelle migliorie.

Il fatto che si tratti di ampliamenti assicura la possibilità di codificare programmi che contengano gli statement del FORTRAN IV e del FORTRAN 77 contemporaneamente, e di far eseguire programmi scritti in FORTRAN IV con compilazioni FORTRAN 77.

E' proprio per tale motivo che in questa sede analizzeremo il nuovo standard procedendo per differenze, o meglio, per ampliamenti nei confronti di quanto già detto nella seconda parte.

Per avere una visione generale delle caratteristiche più importanti dell'evoluzione possiamo brevemente elencare alcuni concetti:

a) inserimento di un nuovo formato dei dati tale da permettere la gestione dei dati composti da caratteri;
b) inserimento di un nuovo modo di codificare i test, maggiori possibilità in alcune istruzioni di controllo, potenziamento del DO, il tutto con l'intento di rendere possibile una codifica di tipo strutturato;
c) nuovo modo di dimensionare gli array e maggiore dinamicità nei relativi trattamenti;
d) potenziamento dell'input-output.

Questi punti verranno trattati uno per uno in dettaglio, tuttavia, se per quanto riguarda i punti c) e d) la consistenza dei miglioramenti risulterà immediatamente evidente quando (tra poco) verranno trattati, per i punti a) e b) può valere la pena di esaminare qualche nota preliminare.

Per quanto riguarda la gestione di costanti non numeriche (né logiche), ovvero di dati composti da caratteri, va tenuto presente che il precedente standard ammetteva un trattamento molto limitato. La novità consiste nella possibilità di utilizzare memorizzazioni di dati, o meglio di variabili, di caratteri: sarà possibile definire variabili con contenuti, ad esempio, alfabetici. Così potrà venir definita una variabile con nome, tanto per fissare le idee, CITT contenente la parola "MILANO", (o qualunque altra stringa di caratteri), essa potrà poi venire utilizzata in riassegnazioni (cioè, si potrà cambiarne il contenuto), confronti, stampe ecc. E' ovvio che risulteranno facilitati tutti quei problemi di input-output, o comunque di generica elaborazione, in cui si renda necessaria la manipolazione di testi.

Per inciso va aggiunto che i nomi delle variabili, o di unità di programma, possono ora raggiungere gli otto caratteri.

Assai importante, ed in proposito va detto qualcosa di più, è il fatto che il FORTRAN 77 permetta una codifica di tipo strutturato.

Il metodo di disegno dei flow-chart che abbiamo adottato e descritto nella prima parte di questo testo è già indirizzato in questo senso: le sole differenze consistono nell'uso della simbologia tradizionale al posto di quella propria della strutturata (1), e nel fatto che la logica di disegno è comunque, per così dire, cronologica, nel senso che le varie operazioni vengono viste nella loro sequenza di elaborazione e non, come vorrebbe la programmazione strutturata vera e propria, in funzione della struttura dell'input-output. Del resto si è scelta la strada dei flow-chart proprio perchè nei problemi di ordine algoritmico e numerico la struttura dell'input-output non è determinante ai fini della effettiva elaborazione: la struttura del problema non è funzione della struttura dei dati messi in input, ovvero di come sono letti dalla procedura, bensì, semmai, di quali sono i loro valori. Lo stesso discorso vale per l'output. Piuttosto è la struttura del problema ad essere funzione della struttura dell'algoritmo solutivo: in definitiva, la codifica del programma dipende da come e quando vadano applicate certe formule di calcolo.

Come si diceva questi sono i motivi che ci hanno fatto preferire l'uso dei flow-chart; quanto sopra, tuttavia, non significa che il programma non vada sviluppato in modo top-down, cioè vedendo prima le operazioni nel loro

(1) Vedere Appendice A: La Programmazione Strutturata.

complessso per poi, via via, dettagliarle. Anzi, come si è già detto nella prima parte, questo modo di procedere, unitamente al rispetto del teorema di Jacopini-Böhm, sicuramente si dimostra essere il più vantaggioso. Ma questo non è che il punto di partenza di ogni discorso di programmazione strutturata!

Fatte, allora, queste considerazioni, è bene o no programmare in strutturata?

Sicuramente esiste tutta la casistica dei problemi di tipo gestionale per i quali la risposta non può che essere affermativa. Per quanto ci riguarda più da vicino, invece, ripetiamo che i problemi di tipo strettamente algoritmico a volte mal si adattano ad una vera e propria strutturazione. Tuttavia ripetiamo anche che il metodo top-down proposto è sempre sicuramente efficace e permette una facile codifica. Ciò specialmente quando si faccia uso di quelle istruzioni che della programmazione strutturata tengono conto, di quelle istruzioni che sono nate proprio in funzione di quest'ultima, o che, comunque, ne rispettano i canoni (quali il teorema di Jacopini-Böhm).

Tra queste va posta una nuova interessante struttura dell'istruzione IF nata col FORTRAN 77 e che analizzeremo più oltre.

L'evoluzione dovuta all'introduzione del nuovo standard, le numerose innovazioni non si esauriscono nei concetti sopra menzionati: avremo modo di vedere che ne esistono altre ed importanti. Queste elencate, tuttavia, ne costituiscono il nucleo ed in esse ci pare risiedano le principali ragioni di interesse del nuovo aspetto del linguaggio.

CAPITOLO 2

# TRATTAMENTO DI STRINGHE E VARIABILI DI CARATTERI

## 2.1 L'istruzione CHARACTER

L'istruzione CHARACTER serve a definire variabili di caratteri. Essa ha la forma:

CHARACTER*n $\mathbf{nome_1}$, $\mathbf{nome_2}$, ..., $\mathbf{nome_k}$

dove **n** indica la lunghezza in caratteri delle variabili il cui nome è presente nella lista che segue; deve trattarsi di una variabile intera non segnata o di una espressione (nel qual caso va posta in parentesi) con risultato intero; nel caso tale indicazione venga omessa viene assunta una lunghezza di un singolo carattere. In alcuni casi n può essere un asterisco tra parentesi (*); come vedremo tra breve ciò permetterà una definizione della lunghezza dei dummy-argument di eventuali sottoprogrammi.
Come si è già accennato $\mathbf{nome_1}$, $\mathbf{nome_2}$, ..., $\mathbf{nome_k}$ costituisce la lista dei nomi delle variabili che intendiamo definire come di caratteri nell'unità di programma che segue.

Aver definito variabili di caratteri significa aver predisposto aree di memoria atte a contenere tutti i possibili caratteri del set grafico ammesso dal FORTRAN 77(1), ovvero dei testi del tipo che viene normalmente chiamato alfanumerico. Così, se ad esempio si desidera definire un contenuto di memoria come il seguente:

| D | O | N | A | L | D |  | D | U | C | K |  | ( | - | 1 | . | 2 | ) | ; | Z | 3 | 7 | 5 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

si definisce una variabile di ventitrè caratteri, e quindi, si esegue un'assegnazione:

```
CHARACTER *23 ALPHA
ALPHA = 'DONALD DUCK  (—1.2); Z375'
```

(1) Vedere appendice D: set grafico del FORTRAN 77 e code-set ASCII, BCD, EBCDIC.

come si vede l'assegnazione del valore alfanumerico viene eseguita utilizzando gli apici a racchiudere il testo, ma le scritture seguenti sono equivalenti:

```
ALPHA = 23HDONALD DUCK (—1.2);Z375
ALPHA = "DONALD DUCK (—1.2);Z375"
```

Qualora nel programma si faccia uso di variabili di caratteri queste devono obbligatoriamente comparire nell'istruzione dichiarativa CHARACTER. Come si è visto nel formato generale, ognuna di queste può servire a definire più variabili (un'intera lista), ma c'è di più, e cioè il fatto che ognuna di queste può servire alla definizione di variabili di lunghezza diversa; così:

```
CHARACTER *10 ALPHA, BETA, GAMMA * 6, DELTA
```

significa che ALPHA, BETA e DELTA sono variabili di caratteri lunghi dieci e GAMMA sei: la specificazione di lunghezza a seguito del nome vale solo per quella particolare variabile.

Si possono avere array di caratteri, nel qual caso la specificazione di lunghezza (comunque posta) vale per tutti i singoli elementi dell'array:

```
CHARACTER * 2 A (2,2), B (100) * 1,C (10)
```

i quattro elementi di A ed i dieci di C sono composti, ognuno, di due caratteri, mentre ognuno dei cento di B di un singolo carattere.

Le variabili di caratteri possono far parte della lista di argomenti di un sottoprogramma, allora, anche in quest'ultimo, è necessaria la definizione di tipo:

```
CHARACTER A*3, B*10
....
....
CALL SUBCH (A)
....
CALL SUBCH (B)
....
STOP
END

SUBROUTINE SUBCH (X)
CHARACTER *(*) X
....
RETURN
END
```

L'uso di (*) significa che alla prima chiamata di SUBCH la variabile di

caratteri X è lunga tre (si tratta in realtà di A), alla seconda chiamata è lunga dieci (si tratta di B): la specificazione della lunghezza per i dummy-argument con l'asterisco tra parentesi permette di dimensionare la lunghezza stessa in maniera flessibile, sulla base delle condizioni di chiamata.

Si tenga presente che sono possibili CHARACTER FUNCTION e che proprio in questi casi risulta particolarmente utile l'uso di (*); a questo proposito si veda l'esempio che segue:

```
CHARACTER *4 FUNCTION EQUAL (A, B)
CHARACTER *(*) A, B
EQUAL = ".NE."
IF (A.EQ.B) EQUAL = ".EQ."
RETURN
END
```

come si vede EQUAL conterrà .NE. o .EQ. dopo l'esecuzione della funzione, fornendo, cioè, un particolare tipo di risposta ad un confronto di stringhe alfanumeriche: quel che più conta è il fatto che queste possono essere di lunghezza qualunque (però a due a due dello stesso numero di caratteri, in questo caso) e la FUNCTION può venir richiamata per coppie diverse ed in momenti diversi.

### 2.2 Espressioni alfanumeriche ed operatore di concatenazione

Ogniqualvolta si utilizzino costanti alfanumeriche o variabili di caratteri per produrre un contenuto alfanumerico, si parla di espressione alfanumerica.

Si è già visto negli esempi delle pagine precedenti come siano possibili le assegnazioni, come sia possibile eseguire i vari test trattando le variabili di caratteri allo stesso modo di tutte le altre; resta da vedere l'unico operatore esistente per questo tipo di dati: l'operatore di concatenazione //. Esso permette di accostare variabili di caratteri sì da ottenere un contenuto ancora di tipo alfanumerico: se sono state definite le variabili di caratteri A, B e C e si ha: A = "AAA", B= "BBB", scrivere:

```
C = A//B
```

significa ottenere C = "AAABBB".

Si tenga presente che la lunghezza del risultato deve essere stata prevista correttamente: se C fosse stata prevista lunga tre caratteri, C = A//B e C = A avrebbero avuto lo stesso effetto, cioè C = "AAA".

E' possibile scrivere un'espressione in cui siano contenute più concatenazioni, in tal caso esse vengono eseguite da sinistra verso destra secondo l'ordine

in cui sono dichiarate; è possibile l'uso delle parentesi, ma esse non hanno alcun effetto sulla concatenazione:

$$A//B//C = (A//B)//C = A//(B//C).$$

Nelle varie concatenazioni è, ovviamente, possibile l'uso di costanti alfanumeriche:

```
CHARACTER *6 X, Y, Z, W*24
X = 'XXXXXX'
Y = 'YYYYYY'
Z = 'ZZZZZZ'
W = X//'—'//Y//'—HHH—'//Z
```

produce W = 'XXXXXX—YYYYYY—HHH—ZZZZZZ'.

### 2.3 Alcune note sulle variabili di caratteri

Prima di concludere quanto detto sulle variabili di tipo CHARACTER vanno fatte alcune precisazioni. La prima di queste riguarda l'uso del FORMAT variabile che, come si è visto nella II parte, permette di generalizzare l'input di un programma attraverso la definizione del FORMAT nell'input stesso.

La differenza sostanziale consiste nel fatto che ora bisognerà definire di tipo CHARACTER la variabile contenente il FORMAT. Così, facendo riferimento all'esempio di pagina 124, ora dovremo scrivere:

```
      CHARACTER *80 FRMT
      DIMENSION ....
      ....
      READ (5,100)FRMT
100   FORMAT (A80)
      READ (5, FRMT) X,I,R1,R2
      ....
      ...
```

si è dovuto dichiarare qualcosa di più (la definizione CHARACTER per FRMT), ma questo ci ha permesso di generalizzare maggiormente il programma. Infatti ci è stato possibile sovradimensionare FRMT: tale variabile è stata definita di 80 caratteri (quanti può contenere una scheda) e ciò permette di utilizzare questa codifica qualunque sia la scrittura sulla prima scheda recante, appunto, la definizione di formato. Quando su tale scheda non venissero scritti parte degli 80 caratteri, in FRMT si avrebbe, come sappiamo per le caratteristiche del FORMAT A, un riempimento di blank.

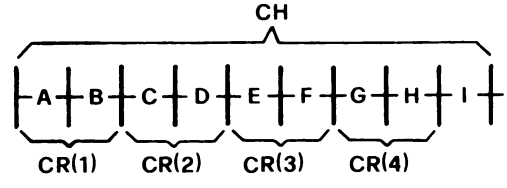
Questi non hanno alcun peso nell'interpretazione del FORMAT che verrebbe, dunque, trattato correttamente.

D'altro canto disporre di FRMT come di una vera e propria variabile permette anche di più: nessuno ci impedisce, infatti, di ridefinire in un punto qualunque del programma quella variabile e dunque di cambiare lo stesso FORMAT dopo un certo numero di letture o comunque al verificarsi di certe condizioni.

Una seconda avvertenza relativa alle variabili di caratteri riguarda l'uso dello statement EQUIVALENCE. Va tenuto presente che, come è ovvio, una variabile di caratteri può essere posta in EQUIVALENCE solo con altre variabili di caratteri. Ciò implica, altrettanto ovviamente, che vada posta una certa attenzione alle definizioni di lunghezza. Un esempio del tipo seguente è corretto:

```
CHARACTER CH*9, CR (4)*2
EQUIVALENCE (CH, CR (1))
CH = "ABCDEFGHI"
...
```

L'assegnazione significa che, poichè ogni elemento di CR è di soli due caratteri, CR(1) contiene "AB", CR(2), poichè come sappiamo l'EQUIVALENCE ha l'effetto di estendere il proprio campo d'azione agli elementi successivi di un array, contiene "CD" e così di seguito come risulta dallo schema seguente:




Una terza ed ultima nota va fatta sui test di variabili di caratteri: sono possibili i test di eguaglianza e disuguaglianza e, contrariamente a quanto si potrebbe pensare, anche se le variabili sono di lunghezza diversa. Così, se:

```
CHARACTER A*3,B*5
A = "AAA"
B = "AAABBB"
```

il test IF (A.EQ.B)... è possibile e dà risposta negativa (le due variabili vengono rese di uguale lunghezza, cioè quella della maggiore, per mezzo di un riempimento di blank: nel nostro caso A risulta minore di B).

I test .GE. , .GT. , .LE. , .LT. faranno riferimento al codice utilizzato dal

compilatore, così (come avviene in tutti i codici) il carattere A verrà ritenuto minore del carattere B, C e via dicendo con le altre lettere dell'alfabeto e, ad esempio, il test della seguente codifica:

```
CHARACTER *3 A,B
A = "0BB"
B = " CC"
IF (A.LT.B) ...
```

se il codice utilizzato è l'ASCII (1), darà risposta negativa in quanto il carattere blank precede, nella tabella del codice stesso, il carattere 0, o, come si dice, pesa di meno. Nel test ha importanza l'esame del carattere più a sinistra: solo nel caso di eguaglianza verrebbero confrontati, nell'ordine, gli altri.

Non sono, ovviamente, permesse IF aritmetiche in riferimento a variabili di caratteri, nonostante la loro codifica sia numerica, come si vede dalla tabella dei codici, e nonostante sia proprio sulla base di tale codice numerico che viene eseguito qualunque confronto.

(1) Vedere Appendice D: Set grafico del FORTRAN 77 e code-set ASCII, BCD, EBCDIC.

CAPITOLO 3

# ISTRUZIONI DI DICHIARAZIONE E DI DEFINIZIONE

## 3.1 La dichiarazione di tipo

Per quanto riguarda le dichiarazioni di tipo va detto che valgono ancora le regole di dichiarazione implicita ed esplicita viste nella II parte. Elenchiamo le definizioni esplicite evidenziando l'unica novità (già analizzata) : il tipo CHARACTER:

INTEGER
REAL
DOUBLE PRECISION
COMPLEX
LOGICAL
**CHARACTER**

Ancora è possibile l'uso dello statement IMPLICIT, va solo aggiunto che, ovviamente, ora potranno far parte delle sue specificazioni anche le iniziali di nomi di variabili di caratteri. Così, una dichiarazione del tipo:

IMPLICIT REAL (I,N), CHARACTER*3 (A-D), CHARACTER (E)

sta a significare che valgono le normali definizioni implicite di tipo per tutte le variabili meno che per quelle che hanno nomi che iniziano per I o per N, che saranno reali, per A, B, C, D che saranno di caratteri lunghe tre ed infine quelle che iniziano per E che saranno variabili di un solo carattere.

Vale la pena accennare ad un'altra novità relativa, questa, alle variabili, ma anche alle costanti complesse e precisamente al loro vero e proprio formato in memoria più che alla dichiarazione di tipo (pur sempre necessaria). I numeri complessi dovevano essere costituiti da una coppia di numeri reali, ora è possibile anche la presenza di interi. Così, il numero complesso $\alpha = 1 + 2i$ può essere rappresentato come (1.,2.) o (1,2) indifferentemente.

## 3.2 La dichiarazione PARAMETER

Qualora si desideri assegnare ad una costante un nome simbolico, si può

utilizzare la dichiarazione PARAMETER che ha la forma:

PARAMETER (**nome**$_1$ = **expr**$_1$, ..., **nome** $_n$ = **expr**$_n$)

dove **nome**$_i$ è il nome simbolico che si desidera assegnare alla costante risultato dell'espressione **expr**$_i$.

Abbiamo detto che il nome viene assegnato ad una costante nel senso che d'ora in avanti il valore non potrà venir modificato. Così potremo scrivere:

PARAMETER (PIGRC = 3.1415927, BASEE = 2.718282)

per poi utilizzare le costanti $\pi$ ed $e$ con i nomi PIGRC e BASEE rispettivamente. Si potrà obiettare che avremmo potuto fare, più semplicemente, le assegnazioni:

```
PIGRC = 3.1415927
BASEE = 2.718282
```

per ottenere lo stesso effetto. Tuttavia va tenuto presente che l'assegnazione attraverso la dichiarazione PARAMETER ci mette al sicuro da successive modifiche fatte per errore sul quel valore. Quanto poi riguardo al fatto che si sarebbero potuti usare direttamente i valori, va detto che sicuramente i nomi sono assai più comodi in codifica e che, ancora, fare errori è assai più improbabile.

Si è detto che l'assegnazione può essere fatta attraverso un'espressione: va semplicemente posta attenzione al fatto che eventuali variabili che entrano a far parte siano già state definite ed inizializzate e che il tipo di espressione e di risultato sia coerente al tipo dichiarato implicitamente od esplicitamente attraverso le usuali regole (ivi inclusa la definizione IMPLICIT).

Sono, allora, possibili situazioni come le seguenti:

```
COMPLEX IMM
REAL IX
I = 1
X = 2.5
....
PARAMETER (IMM = (0,1), C = 2.722,IX = X + 3.5)
CHARACTER CH * 4
IMPLICIT REAL (K)
....
PARAMETER (KLU = 2.517172, CH = "COST")
```

Nell'uso della dichiarazione PARAMETER è richiesta l'osservanza di alcune altre regole: una costante dichiarata in tal modo, si è detto, non può venir modificata, va precisato che non può neppure essere ridefinita in una ulterio-

re dichiarazione PARAMETER; una costante così definita non può far parte di una specifica di FORMAT; essa non può, infine, entrare a far parte di un'espressione che definisca un'altra costante di uno statement PARAMETER.

### 3.3 Dimensioni degli array

Lo statement DIMENSION permette di definire le dimensioni (numero degli indici) ed il range di ogni dimensione di un array come già in FORTRAN IV.
Ancora è 7 il massimo numero di dimensioni, ma vi sono sostanziali novità per quanto riguarda il range, ovvero il campo di variazione, degli indici. Il formato è:

$$\text{DIMENSION } a_1 (m_1: n_1, m_2: n_2, \ldots m_k: n_k), a_2 \ldots, a_h (\ldots)$$

dove $\mathbf{a_i}$ è il nome di un array di k dimensioni che variano tra $\mathbf{m_i}$ ed $\mathbf{n_i}$, analogamente si scrive per $\mathbf{a_2} \ldots \mathbf{a_h}$.
E dunque:

```
DIMENSION MATR (10:15, — 2:2), V (3,0:2)
```

sta ad indicare che MATR è una matrice intera di 6 righe e 5 colonne: l'indice di riga può variare tra 10 e 15 mentre l'indice di colonna può variare da —2 a 2 (assumendo anche il valore 0); V è una matrice reale di 3 righe e 3 colonne: l'indice di riga può variare da 1 a 3 (quando l'estremo inferiore non è espresso, viene assunto uguale ad 1), mentre l'indice di colonna può variare da 0 a 2.

Come si nota è ora possibile utilizzare valori negativi o nulli per gli indici specificando espressamente da quale valore a quale altro può variare l'indice: ciò può risultare utile in più di un caso (si veda l'esempio di pag. 221). Ovviamente andrà posta attenzione al fatto di dichiarare il limite inferiore minore di quello superiore, ma soprattutto si dovrà badare ad utilizzare per gli indici dei valori permessi: cioè per array così dichiarati:

```
DIMENSION A(0:1, 10,10:30),B(1:6,—1:1)
REAL I(0:1), L(—10:0)
```

potremo identificare correttamente alcune loro componenti scrivendo ad esempio:

```
A(0,2,15)  B(3,1)     I(0)  L(—2)
A(1,9,11)  B(6,—1)    I(1)  L(0)
```

ma sarebbe un errore scrivere:

A(1,2,2)  B(2,2)  I(2)  L(1)

nonostante A abbia la terza dimensione di 21 elementi, B tre colonne, I due elementi ed L undici.

I valori del limite inferiore e superiore degli indici possono essere espressi mediante costanti intere (come negli esempi e come è più frequente fare), nomi di costanti intere definite precedentemente in uno statement PARAMETER, o, infine, espressioni di costanti o variabili intere con risultato intero. Se il dimensionamento è fatto in quest'ultimo modo, le variabili coinvolte devono, ovviamente, essere state definite prima dello statement DIMENSION.

Il dimensionamento sopra descritto può essere fatto in uno qualunque degli statement dichiarativi del tipo già visto.

### 3.4 Dichiarazioni e definizioni relative a sottoprogrammi

Tra le istruzioni di tipo dichiarativo ne vanno elencate tre che si riferiscono specificatamente all'uso di sottoprogrammi: si tratta delle istruzioni EXTERNAL, INTRINSIC e SAVE. Si è già visto nella seconda parte che EXTERNAL serve a definire che un nome simbolico è relativo ad un sottoprogramma esterno. Più in generale:

EXTERNAL **nome**$_1$, **nome** $_2$, ... **nome**$_n$

significa che **nome**$_i$ è il nome di un sottoprogramma *utente* (o nome di una procedura esterna), il nome di un argomento fittizio di una CALL, oppure, infine, di un BLOCK DATA;

Gli usi più frequenti di un tale statement sono proprio relativi alla dichiarazione di un nome di sottoprogramma come argomento, così se nel sottoprogramma viene scritto:

```
EXTERNAL F1
...
CALL SUB1 (A,B,F1)
...
```

F1 non viene riconosciuta dal compilatore come una variabile, bensì come il nome di una funzione esterna.

Va notato che, in FORTRAN 77, se avessimo scritto

```
EXTERNAL SQRT
```

il compilatore avrebbe assunto SQRT come nome di una funzione esterna (dell'utente e non del sistema). Così il programma avrebbe dovuto presentarsi, ad esempio, come segue:

```
      EXTERNAL SQRT
      ...
      CALL SUB1 (X,SQRT)
      ...
      STOP
      END
      FUNCTION SQRT(Y)
      ...
      ...
      RETURN
      END
```

rendendo SQRT nome, appunto, di una funzione esterna codificata come più ci piace. Ciò comporta, per quel nome, di non essere più utilizzabile nel programma principale come nome di una funzione di sistema, o, come viene normalmente chiamata, di una funzione intrinseca: il normale calcolo della radice quadrata è stato sostituito dal calcolo codificato nel nostro programma SQRT.

Tutto ciò comporta l'ovvia limitazione che i nomi di funzione intrinseca non possano venire più utilizzati come dummy argument. In FORTRAN 77, per soddisfare tale esigenza esiste lo statement INTRINSIC. Esso ha la forma:

```
INTRINSIC funz1, funz2 ..., funzn
```

dove **funz**$_i$ è il nome di una funzione intrinseca. In generale tale enunciato permette di identificare con funz$_i$ il nome di una funzione intrinseca, in particolare permette di specificare che una certa funzione intrinseca può venir usata come argomento da passare ad un sottoprogramma. Così se desideriamo utilizzare il nome SQRT per una "FUNCTION" da noi codificata (ed alternativa, dunque, alla radice quadrata), nonchè le funzioni intrinseche SIN e COS:

```
      EXTERNAL SQRT
      INTRINSIC SIN, COS
      ...
      CALL SUBRTN (A,SQRT)
      ...
      CALL SUBRTN (A,SIN)
```

```
...
CALL SUBRTN (A, COS)

...
STOP
END

SUBROUTINE SUBRTN (X, FRAN)
X = FRAN (X)
RETURN
END

FUNCTION SQRT (Y)
SQRT = Y/(Y-1)
RETURN
END
```

Va tenuto presente che non possono essere usati come argomenti i nomi di quelle funzioni intrinseche che provvedono alle conversioni di tipo, alle relazioni ed alle ricerche di massimo e minimo: tali funzioni sono elencate qui di seguito:

| | | |
|---|---|---|
| INT | LGE | MAX |
| IFIX | LGT | MAX0 |
| IDINT | LLE | AMAX1 |
| FLOAT | LLT | DMAX1 |
| SNGL | | AMAX0 |
| REAL | | MAX1 |
| DBLE | | MIN |
| CMPLX | | MIN0 |
| ICHAR | | AMIN1 |
| CHAR | | DMIN1 |
| | | AMIN0 |
| | | MIN1 |

Un'ultima istruzione dichiarativa riguardante i sottoprogrammi è:

SAVE $\mathbf{var_1}$, $\mathbf{var_2}$, ..., $\mathbf{var_n}$

dove $\mathbf{var_i}$ è il nome di una variabile o di un array.

E' anche possibile:

SAVE /**lab**/

dove **lab** è una label di block common (1): si noti che in quest'ultimo formato non è permesso aggiungere la lista di variabili e che l'uso della SAVE è riferito a tutto il gruppo con quella label.

---

(1) Ovvero il nome di gruppo di una lista di variabili specificate nello statement COMMON (v. parte II).

Lo statement SAVE permette di evitare che al ritorno da un sottoprogramma le variabili ivi definite vengano perdute. Normalmente, dopo l'esecuzione di uno statement di RETURN le grandezze che non compaiono tra gli argomenti di chiamata risultano indefinite: l'istruzione SAVE permette di evitare ciò conservando inalterato il valore di quelle variabili fino alla prossima esecuzione dello stesso sottoprogramma.

Nel caso SAVE non sia seguito da alcun nome si intende che vadano conservati tutti i valori delle variabili localmente definite in quel sottoprogramma.

Nel formato relativo ad una label di block-common l'istruzione SAVE ha l'effetto di conservare tutti i valori delle variabili appartenenti a quel blocco.

Per meglio comprendere l'uso di questo statement esaminiamo l'esempio che segue:

```
      ...
      X = ...
      DO 1 I = 1,N
      ...
      CALL SUBELI (X)
      ...
1     CONTINUE
      ...
      STOP
      END
      SUBROUTINE SUBELI (Y)
      INTEGER CONT
      SAVE CONT
      CONT = CONT + 1
      ...
      IF (CONT — 10)10,11,11
10    Y = Y + 1
      GO TO 12
11    Y = Y—1
12    WRITE (...) CONT
      RETURN
      END
```

il valore di CONT viene salvato al termine di ogni esecuzione di SUBELI (1) e, come conseguenza di ciò, il valore che X assumerà nel main sarà pari ad

(1) Si suppone che (come avviene praticamente su tutti i sistemi) la memoria sia inizializzata automaticamente a zero all'inizio dell'esecuzione, cosi da rendere possibile l'istruzione CONT = CONT + 1 già alla prima chiamata.

X + 1 per le prime nove chiamate, per le eventuali successive assumerà il valore X—1; inoltre ad ogni esecuzione della subroutine verrà scritto il valore di CONT e cioè il valore corrispondente all'ordine di chiamata.

Si noterà che CONT non è stato nominato tra gli argomenti: tra le variabili dichiarate in SAVE non è permessa la presenza di argomenti. Non sarebbe stato vietato aggiungere SAVE CONT anche nel programma principale, ma ciò non avrebbe avuto alcun effetto.

### 3.5 La definizione di nome PROGRAM

Come già sappiamo un'unità di programma è un gruppo di istruzioni eseguibili e non eseguibili che svolgono certe funzioni di elaborazione. Essa potrà consistere del programma principale o di un sottoprogramma richiamabile e dovrà terminare rispettivamente con:

```
STOP     RETURN
END      END
```

Si è visto che il nome dei sottoprogrammi è definibile (anzi, tale definizione è obbligatoria) con gli statement:

BLOCK DATA ... FUNCTION ... SUBROUTINE ... ENTRY ...

ed a quel nome si farà riferimento in una eventuale istruzione di chiamata.

È ora possibile definire un nome anche per il programma principale attraverso lo statement PROGRAM, intendendo, con ciò, battezzare il programma nella sua intierezza con lo scopo, sostanzialmente, di una migliore documentabilità. In alcuni casi il nome assegnato può risultare utile anche per altri motivi: ad esempio quando si voglia collegarlo all'esecuzione di altri programmi e si renda, dunque, necessario comunicare tale nome al sistema operativo.

Lo statement PROGRAM non è obbligatorio, se si desidera usarlo esso ha la forma:

PROGRAM **nome**

dove **nome** è appunto il nome con cui desideriamo battezzare il programma ed è composto secondo le usuali regole (da uno a otto caratteri alfanumerici, di cui almeno il primo alfabetico); tale nome deve essere unico relativamente alle varie unità di programma, ma anche rispetto ad altri nomi poi definiti nel programma stesso (di variabile, od altro che siano). Lo statement PROGRAM deve precedere qualunque altra istruzione del programma.

Si ponga inoltre attenzione al fatto che un sottoprogramma BLOCK DATA non deve far parte del programma principale, esso è un sottoprogramma a

tutti gli effetti, e dunque non deve trovarsi nell'unità di programma battezzata da una PROGRAM. Così la struttura generale dovrà essere del tipo:

```
BLOCK DATA
....
....
....
END
PROGRAM PRINCI
....
....
....
STOP
END
SUBROUTINE ...
...
...
RETURN
END
```

CAPITOLO 4

# ISTRUZIONI DI CONTROLLO

## 4.1 GO TO e DO

Come si era già osservato, il FORTRAN 77 presenta, rispetto al precedente standard, alcune sostanziali innovazioni relative alle istruzioni di controllo.

Prima di esaminare le maggiori novità si tenga presente che esistono anche differenze meno evidenti, ma anch'esse di una certa importanza, spesso relative a quegli statement che mantengono inalterato il loro formato generale e le principali caratteristiche di funzionamento.

E' questo il caso, ad esempio, dello statement GO TO: nel formato che permette il salto incondizionato non vi sono novità; a proposito, invece, del GO TO calcolato va aggiunto qualcosa. Precedentemente sarebbe stato un errore (con molti compilatori) porre nella variabile di tale statement un valore maggiore, o minore, del numero delle label indicate nella lista, ora, invece, vale **per tutti** che l'esecuzione procede normalmente con l'elaborazione dello statement successivo.

Così nel caso della seguente codifica:

```
      J = 0
11    J = J + 1
      GO TO (12,13,14),J
10    IF (...) ...
12    CALL SUB1
      GO TO 11
13    A = A + ...
      GO TO 11
14    CALL SUB2
      GO TO 11
      ...
```

J, assumendo i valori 1, 2, 3 provoca il rimando, rispettivamente alle label 12, 13, 14, quando assume il valore 4 viene eseguito il test codificato alla label 10.

Un altro statement che mantiene praticamente inalterato il proprio formato generale è il DO:

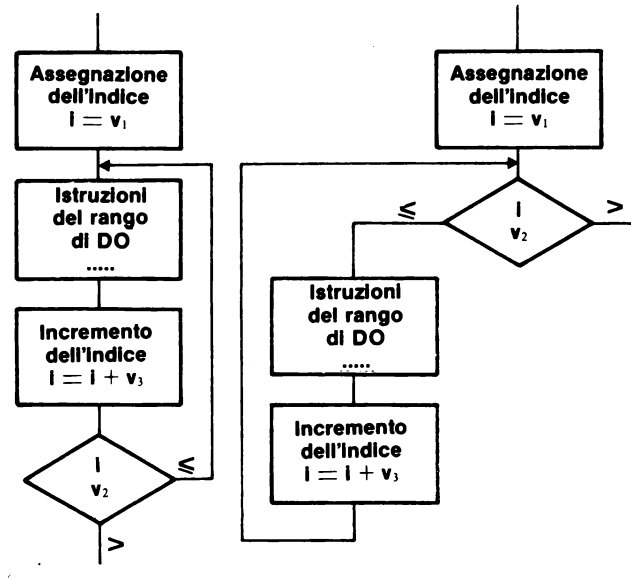
DO **lab, i** = $v_1$, $v_2$, $v_3$

si noti che è possibile inserire una virgola tra **lab** ed **i** (ma non è obbligatoria).

Ancora **lab** è la label dell'ultima istruzione del rango di DO, **i** è l'indice di controllo del ciclo, $v_1$ il valore iniziale, $v_2$ quello finale e $v_3$ quello dell'incremento (è opzionale e se non è presente viene assunto uguale ad uno) assegnato alla variazione dell'indice. I valori $v_1$, $v_2$, $v_3$ possono essere dichiarati esplicitamente, o possono essere delle espressioni: questa volta potranno essere di tipo sia intero che reale che doppia precisione.

Valgono ancora le preccedenti regole di uscita da un DO, in più va tenuto presente che la label **lab** non può essere riferita ad un blocco di IF o ad una END IF (istruzione che vedremo tra breve) e che se un blocco di IF viene inserito nel ciclo di DO esso deve esservi interamente contenuto.

Un'altra particolarità di una certa importanza, e che rappresenta un'innovazione, riguarda il momento in cui viene eseguito il test di fine ciclo: precedentemente esso veniva eseguito subito dopo le istruzioni del rango e il relativo incremento, ora avviene esattamente il contrario.




**FORTRAN IV** **FORTRAN 77**

Cioè: ad ogni iterazione del ciclo viene eseguito l'incremento sommando $v_3$ al valore corrente di **i** solo dopo aver eseguito il controllo per vedere se il valore $v_2$ è stato raggiunto. Così il ciclo:

```
      DO 5 I=1,N
      ...
5     CONTINUE
```

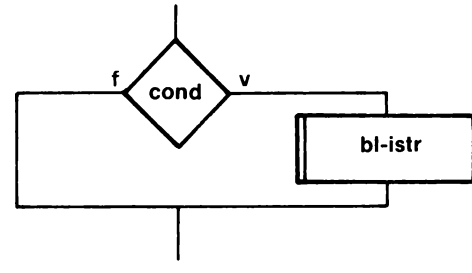
viene eseguito una sola volta per N = 1 ed all'uscita I vale 2, per N = 10 il ciclo viene eseguito 10 volte ed all'uscita I vale 11.

## 4.2 L'istruzione IF...THEN...ELSE

Accanto alle istruzioni di IF logico ed aritmetico viste nella parte II, il FORTRAN 77 permette la codifica di un test in modo strutturato per mezzo di quello che viene chiamato **blocco di IF**. Esso è assai simile all'IF logico, ma ne differisce per il fatto che ora può essere eseguito un gruppo di istruzioni (il blocco, appunto), invece di una sola (scritta sulla stessa riga), se la condizione testata risulta vera; altrimenti, se la condizione risulta falsa, usando la clausola ELSE può venire eseguito un gruppo di istruzioni alternative. Inoltre all'interno del blocco può essere inserita un'altra IF nidificando i vari blocchi come si preferisce.
Questa istruzione va esaminata in almeno tre formati fondamentali: come vedremo, una loro corretta combinazione permette soluzioni di codifica di una certa concisione e di notevole semplicità.

Il primo formato è:

IF *(cond)* THEN
*bl-istr*
END IF




dove:

| | | |
|---|---|---|
| | *cond* | è una condizione logica del tipo che siamo abituati a conoscere, |
| | THEN | serve ad indicare (precedendole) le istruzioni da eseguire quando *cond* è vera, |
| | *bl-istr* | è appunto tale blocco di istruzioni, |
| | END IF | delimita il blocco e va espresso obbligatoriamente. |

Come si è certamente compreso le istruzioni del blocco *bl-istr* non vengono eseguite se *cond* è falsa e in questo formato non esiste un gruppo di istruzioni alternative, ma semplicemente si proseguiirà in sequenza dopo la ENDIF.

Va notato che la presenza del termine THEN è obbligatoria e sta ad indicare che si tratta di un blocco di IF piuttosto che di un IF logico; per lo stesso motivo la prima istruzione del blocco non può iniziare sulla stessa linea del THEN.

Così:

```
...
IF(A.GT.B) THEN
              X=A—B
              I=I+1
ENDIF
...
```

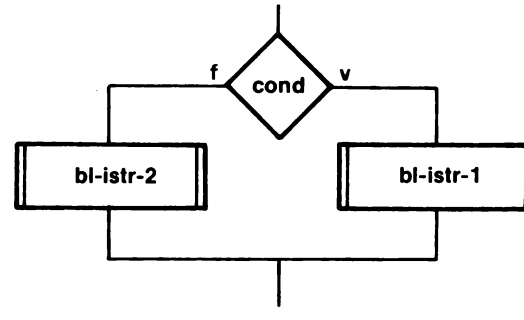
è una codifica corretta: se A è maggiore di B, X assume il valore della differenza A-B ed I viene incrementato di 1; se A è minore di B, X ed I restano con i valori eventualmente definiti in precedenza e si prosegue dopo la ENDIF. Si ricordi che ogni istruzione FORTRAN può iniziare a partire da colonna 7, dunque è permesso scrivere un'istruzione da una posizione più avanzata come è stato fatto per le assegnazioni di X ed I: normalmente il blocco di istruzioni viene scritto in questo modo allo scopo di facilitare la leggibilità del programma.

Si tenga presente che le istruzioni che compongono il blocco di IF possono essere di tipo qualunque (ivi comprese altre IF).

Se si fosse voluta l'esecuzione di qualche altra operazione nel caso di A < B il formato visto non sarebbe stato soddisfacente: sarebbe quindi stato assai più utile il secondo formato.

Esso è del tipo:

IF *(cond)* THEN
*bl-istr-1*
ELSE
*bl-istr-2*
END IF




il blocco *bl-istr-1* viene eseguito se *cond* è vera, altrimenti viene eseguito il blocco *bl-istr-2*.

Una codifica corretta avrebbe potuto essere la seguente:

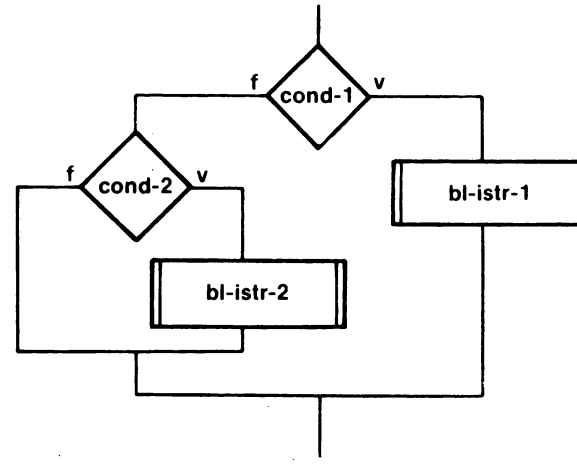
```
                ...
IF(A.GT.B)      THEN
                X=A—B
                I=I+1
ELSE
                X=B—A
                J=J+1
ENDIF
...
```

Il terzo formato riguarda strettamente la clausola ELSE la quale può, in questo caso, essere seguita da un'altra IF:

ELSE IF...

Così questo terzo formato sarà del tipo:

IF *(cond-1)* THEN
  *bl-istr-1*
ELSE IF *(cond-2)* THEN
  *bl-istr-2*
END IF




con gli ovvii significati di *cond-1, cond-2, bl-istr-1* e *bl-istr2.*

Una codifica di tipo corretto sarà, ad esempio:

```
...
IF(A.NE.B) THEN
  X=A+B
  I=I+1
ELSE IF (A.GT.0) THEN
        X=A
        J=J+1
END IF
```

Per comportarsi del tutto coerentemente al primo formato si sarebbe dovuto scrivere:

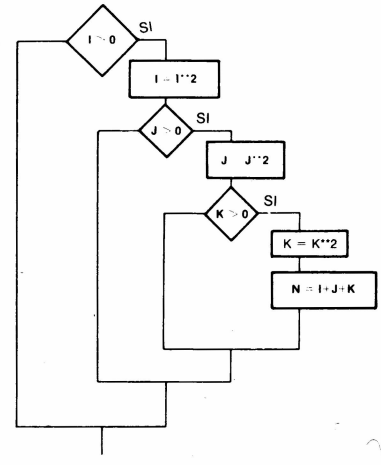
```
IF(A.NE.B) THEN
  X=A+B
  I=I+1
ELSE IF (A.GT.0) THEN
        X=A
        J=J+1
      ENDIF
ENDIF
```

con la prima ENDIF a chiudere il blocco della IF più interna: ciò non è necessario in quanto un'unica ENDIF (come nell'esempio precedente) già chiude tutto il blocco. Combinando i formati visti si possono nidificare in modo composito le varie IF; vanno tenute presenti due regole fondamentali: ogni ENDIF chiude il blocco della IF precedentente o di più IF se queste sono codificate senza ELSE come nel primo formato; ogni ELSE eventualmente

presente è relativa alla IF immediatamente precedente, non già chiusa da un'altra ELSE.

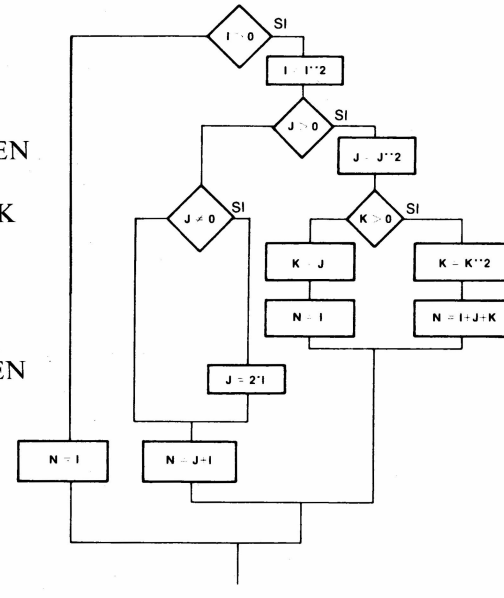
Così si possono scrivere test nidificati come i seguenti:

```
IF    (I.GT.0) THEN
      I=I**2
      IF(J.GT.0) THEN
         J= J**2
         IF (K.GT.0) THEN
            K=K**2
            N=I+J+K
ENDIF
```




oppure anche, se esistono gruppi di istruzioni alternative:

```
IF    (I.GT.0) THEN
      I= I**2
      IF (J.GT.0) THEN
            J = J**2
            IF (K.GT.0) THEN
                  K = K**2
                  N=I+J+K
            ELSE
                  K=J
                  N=I
            ENDIF
      ELSE IF (J.NE.0) THEN
               J=2*I
      ENDIF
      N=J+I
ELSE
      N=I
ENDIF
```




Come si vede ogni gruppo di IF è chiuso dalla sua ENDIF e in ognuno di essi sono presenti blocchi di istruzioni i quali possono contenere o meno altre IF

con relative ELSE; si noti anche che nel caso di ELSE IF (J.NE.0) THEN... la chiusura è stata ottenuta con un unico ENDIF relativo sia al test $J \neq 0$ che $J > 0$.

### 4.3 Alcuni confronti ed esempi

Concludiamo questo capitolo relativo alle istruzioni di controllo, svolgendo alcuni esempi. Essi ci serviranno sia per verificare l'utilità delle istruzioni finora trattate, sia per evidenziare le maggiori potenzialità del nuovo standard; nel primo caso, anzi, eseguiremo un diretto confronto tra i due tipi di codifica.

*Esempio 1* - Sia dato un vettore IV di 10 elementi: esso è stato riempito disordinatamente con i diversi valori degli interi compresi tra —5 e +5 con esclusione dello 0; si desidera portare tali valori nella prima riga di una matrice A di 3 righe e 11 colonne disponendo i valori in ordine crescente e inserendo lo 0; nella seconda riga si vuole scrivere il quadrato di ogni valore; nella terza la radice quadrata dei numeri positivi e lo 0 per i negativi. Cioè, assumendo che IV si presenti nel modo seguente:

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| -3 | +2 | +3 | +4 | -2 | -1 | -4 | -5 | +5 | +1 |

si vuole ottenere A

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| -5 | -4 | -3 | -2 | -1 | 0 | +1 | +2 | +3 | +4 | +5 |
| 25 | 16 | 9 | 4 | 1 | 0 | 1 | 4 | 9 | 16 | 25 |
| 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1.4142 | 1.732 | 2 | 2.236 |

Le due codifiche potranno essere del tipo:

**FORTRAN IV**

```
  DIMENSION IV (10), A (3,11)
  DO 1 K = 1,3
1 A (K,6) = 0
  DO 7 K = 1,10
  J = IV (K) + 6
  A (1,J) = IV (K)
  A (2,J) = IV (K) **2
  IF (IV (K)) 3,3,2
2 A (3,J) = SQRT (IV (K))
  GO TO 7
3 A (3,J) = 0
7 CONTINUE
  ...
```

**FORTRAN 77**

```
  DIMENSION IV (10), A (3, - 5 : 5)
  DO 1 K = 1,3
1 A (K,0) = 0
  DO 7 K = 1,10
  A (1, IV (K)) = IV (K)
  A (2,IV (K)) = IV (K)**2
  IF (IV (K). GT. 0) THEN
    A (3, IV (K)) = SQRT (IV (K))
  ELSE
    A (3, IV (K)) = 0
  ENDIF
7 CONTINUE
  ...
```

Come si noterà (le differenze sono evidenziate in grassetto) il dimensionamento di A da —5 a 5 per l'indice di colonna permette una più semplice codifica del ciclo non essendovi più bisogno dell'indice ausiliario J e l'uso del blocco di IF permette di evitare i rimandi con la GO TO.

*Esempio 2* - Si leggono due valori X ed Y, che esprimono la misura di due angoli in radianti ed un valore numerico intero N; si vuole controllare che X ed Y siano compresi tra 0 e 6.28 e che Y sia maggiore di X segnalando eventuali valori errati; quindi si vuole calcolare il seno ed il coseno di N+1 angoli con ampiezza compresa tra X e Y ed uniformemente distribuiti in questo intervallo, emettendo i risultati in stampa. Se ad esempio X=0.58 Y=0.88 N=3 si vuole calcolare, dopo il controllo:

seno e coseno di 0.58
seno e coseno di 0.68
seno e coseno di 0.78
seno e coseno di 0.88

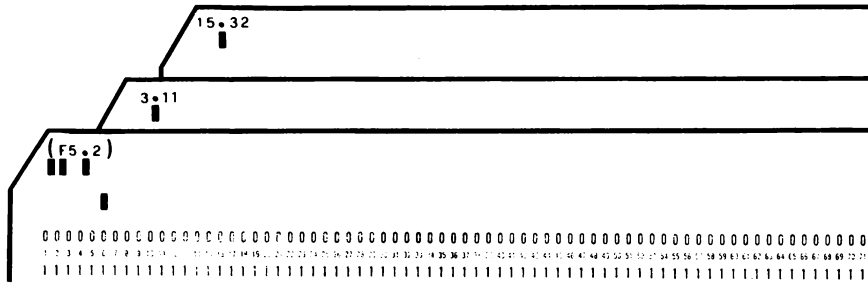
La codifica in FORTRAN 77 potrebbe essere del tipo:

```
      CHARACTER *1A
      ...
      READ (...)X,Y,N
      IF ((X.GT.6.28) .OR.(X.LT.0)) THEN
        A="X"
        WRITE (6,100)A,X
      ELSE IF  ((Y.GT.6.28).OR.(Y.LT.0).OR.(Y.LT.X)) THEN
                A="Y"
                WRITE (6,100)A,Y
           ELSE
                DO 1 R=X,Y,((Y-X)/N)
                Z=SIN(R)
                W=COS(R)
                WRITE (6,...)Z,W
1               CONTINUE
      ENDIF
100   FORMAT (" VALORE ERRATO DI",A1,"=",F5.2)
                ...
```

Senza esaminare in dettaglio come sarebbe stato necessario procedere in FORTRAN IV, si può notare, però, che non avremmo potuto comunque raggiungere una analoga concisione: nei controlli dei valori di X ed Y la codifica delle IF ci avrebbe costretto a ripetuti salti, ad un uso del GO TO questa volta non troppo semplice e quindi di successiva difficile reinterpreta-

zione. L'uso della variabile di caratteri A ci ha permesso di utilizzare un solo FORMAT per la segnalazione di errore (in A si può porre sia X che Y). Infine il ciclo per il calcolo del seno e del coseno è realizzato sui valori reali degli stessi X ed Y evitando (come sarebbe stato necessario in FORTRAN IV) l'uso di variabili ausiliarie in cui porre gli appropriati valori interi ricavati da X ed Y per eseguire il DO.

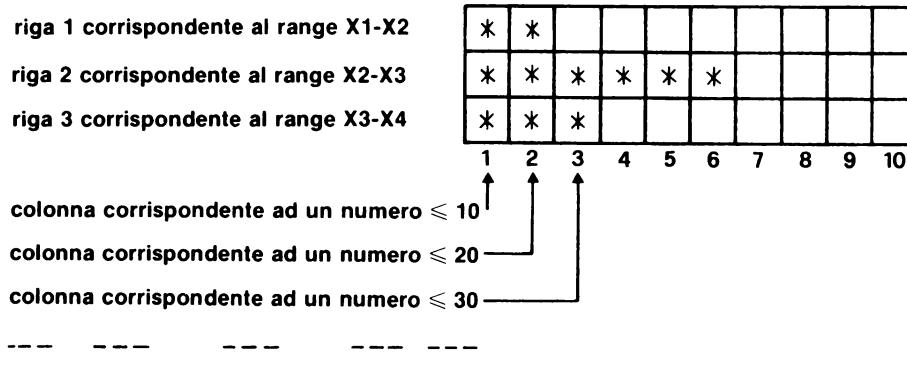
*Esempio 3* - Cento valori numerici reali sono scritti uno per scheda con premesso il FORMAT di lettura sulla prima scheda:



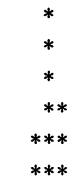

si leggano (con FORMAT variabile) tali valori e, confrontandoli con 4 valori X1, X2, X3, X4, già presenti in memoria, si esegua un conteggio di quanti di essi cadono in ognuno dei range X1-X2 X2-X3 X3-X4 ponendo il risultato in un vettore IC(3). Si generi una matrice di caratteri M(3,10) ponendo degli asterischi nelle componenti delle righe corrispondenti a valori minori di quelli contenuti in IC, in altre parole se IC, ad esempio, contiene:

| 19 | 55 | 26 |
|---|---|---|
| IC(1) | IC(2) | IC(3) |

si vuole una matrice M(3,10) fatta nel modo seguente:



Si mandi infine in stampa la trasposta di tale matrice per ottenere un'immagine della distribuzione.



```
      CHARACTER M (3,10), FORM * 6
      ...
      DO 5 K = 1,3
    5 IC (K) = 0
      READ (05,100) FORM
      DO 10 K = 1,100
      READ (05, FORM)
      IF (X. GE. X1. AND. X. LT. X2) THEN
         IC (1) = IC (1) + 1
      ELSE IF (X.GE.X2.AND.X.LT.X3) THEN
               IC (2) = IC (2) + 1
            ELSE IF ((X.GE.X3).AND.(X.LE.X4)) THEN
                     IC (3) = IC (3) + 1
                  ELSE
                     WRITE (06,200) K, X
      END IF
   10 CONTINUE
  100 FORMAT (A6)
  200 FORMAT ("IL VALORE NUM "13", È FUORI RANGE: X = ",F6.2)
      DO 15 J = 1,3
      DO 15 K = 1,10
      I = K * 10
      IF (IC (J) .LE. I) THEN
         M (J,K) = "*"
      ELSE
         M (J,K) = " "
      ENDIF
   15 CONTINUE
      DO 18 K = 1,10
   18 WRITE (05,300) (M (I,K), I = 1,3)
  300 FORMAT (20X, 3A1)
      ...
      ...
      STOP
      END
```

CAPITOLO 5

# I FILE E L'INPUT-OUTPUT

## 5.1 Caratteristiche dei file

I file trattabili in FORTRAN 77 vanno divisi, come già in FORTRAN IV, in due grossi gruppi: quelli ad accesso **sequenziale** e quelli ad accesso **diretto**. Ancora parliamo di accesso sequenziale quando i record vengono letti o scritti nell'ordine che hanno (od avranno) fisicamente nel file. Parliamo di accesso diretto quando, invece, possiamo trattare il record che desideriamo in modo immediato, indipendentemente dalla sua posizione.

Per questi concetti rimandiamo alla II parte di questo testo dove i due tipi di accesso sono stati trattati un po' più diffusamente: in questa sede ci limiteremo ad osservare che, se non esistono novità sostanziali di struttura, esistono però nuove e notevoli possibilità di trattamento che comportano tutta una serie di differenze nei formati dei relativi statement.

Prima di passare all'esame di questi nuovi formati va segnalata una seconda suddivisione in **external** od **internal** file: gli **external** file sono quelli che già siamo abituati ad usare anche in FORTRAN IV e che fanno riferimento ad una risorsa esterna, nel senso che il loro supporto (schede,nastro,disco,carta, terminale) è trattato da una periferica.

Per **internal** file, invece, si intende né più né meno che una variabile od un array presente in memoria e sul quale intendiamo agire con i verbi READ o WRITE al fine di convertirne il formato (da un formato a caratteri ad uno numerico o viceversa). Tali file andranno ritenuti ad accesso sequenziale e trattati solo sequenzialmente ed il record non sarà altro che la variabile stessa od il singolo elemento dell'array. In pratica si tratterà di considerare un insieme di locazioni di memoria come se fosse un file costituito da una sequenza di caratteri. Da tale zona si potranno leggere quei caratteri (dovrà esserne possibile una interpretazione numerica e dovrà, dunque, trattarsi di cifre o del punto decimale) per assegnare ad altre variabili i corrispondenti valori numerici. Viceversa si potranno scrivere in quelle locazioni i caratteri corrispondenti ai valori di certe altre variabili numeriche. Le modalità secondo le quali avverranno tali conversioni saranno definite per mezzo di FORMAT opportuni. Quando tratteremo i verbi READ e WRITE esamineremo in dettaglio come si possa ottenere ciò praticamente.

Teniamo presente un'altra distinzione: quella tra **file formattati** e **non formattati**.

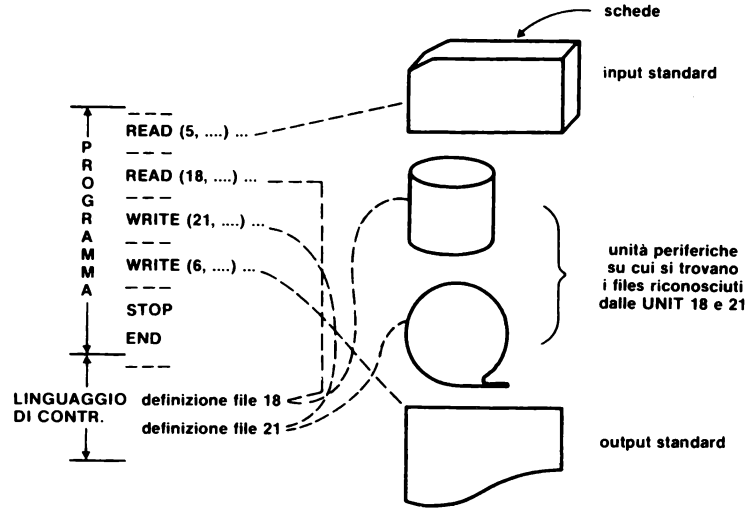
Ricordiamo che l'uso del FORMAT permette di leggere e memorizzare dati secondo i criteri ed i formati, appunto, desiderati, o di converso, di portarli in output in una forma leggibile. Normalmente si dice che i dati durante le operazioni di input-output vengono *editati*: infatti è come se ne producessimo un'*edizione* definendone la lunghezza in caratteri ed il codice di rappresentazione. Si pensi ad una variabile intera il cui contenuto in memoria sia ad esempio il numero 31. Come sappiamo la rappresentazione di tale numero è *binaria*: il numero in memoria è rappresentato bit per bit (11111 è il binario corrispondente). Se al momento di emettere in output tale variabile le assegnamo un FORMAT I3, intendiamo richiedere la conversione da binario a caratteri decimali (leggibili da parte nostra) e intendiamo far stampare tre caratteri, così da ottenere 31.

In FORTRAN 77 sono utilizzabili tutte le specifiche di FORMAT già viste, più alcune nuove di cui parleremo più oltre.

Quando si desidera che un'operazione di input-output non comporti le conversioni di *edizione*, si fa a meno del FORMAT. Ancora le caratteristiche di un input-output non formattato sono le stesse viste nella parte precedente.

Per completare queste osservazioni generali sui file vanno aggiunte alcune precisazioni riguardo la *connessione* tra programma e file, ovvero su come si produca il riconoscimento logico del file da utilizzare negli statement di input-output.

Come si è già detto, scrivere READ(5,...)... significa fare riferimento (con il 5) al lettore di schede, all'unità di input standard del sistema; scrivere WRI-



TE(6,...)... significa fare riferimento alla stampante o all'unità di output standard. Un file qualunque verrà allocato al programma specificando un numero intero qualunque diverso dai precedenti, e definendo poi, esternamente al programma, su quale periferica si trovi quel file. Tale definizione esterna dipenderà dal sistema (ovvero dal tipo di elaboratore).

La cosa più importante è tenere presente che nel programma un certo file è identificato da un preciso valore intero dall'inizio alla fine del suo trattamento.

Anche in FORTRAN 77 tale valore numerico rappresenta l'unità logica associata al file e prende il nome di UNIT ed è attraverso la definizione di questa specifica che un file viene *connesso* al programma.

Tutte queste operazioni vengono normalmente dette **apertura** del file e sono automaticamente eseguite al primo statement di input-output relativo a quel file. La *sconnessione* avverrà solamente alla fine del programma e prenderà il nome di **chiusura** del file.

Quanto detto finora valeva più o meno inalterato anche nel precedente standard del FORTRAN, ora si ha in più la possibilità di provvedere ad aperture o chiusure esplicite: in FORTRAN 77 l'apertura (e dunque la *connessione*) di un file, può essere richiesta attraverso lo statement OPEN (che esamineremo tra breve) senza per questo dover eseguire un'operazione di lettura o scrittura. Altrettanto dicasi per la *sconnessione* (statement CLOSE).

Questa potrebbe sembrare una novità da poco, ma non è così: si pensi ad esempio che in questo modo si rende possibile connettere un file, lavorarvi, sconnetterlo per poi connetterne un altro utilizzando gli stessi statement di input-output. Ma i vantaggi sono ovviamente ben maggiori e potranno venir verificati solo in seguito.

Concludiamo dando l'elenco completo dei verbi di input-output utilizzabili in FORTRAN 77:

| | |
|---|---|
| READ<br>WRITE<br>PRINT | verbi che provvedono all'input-output |
| OPEN<br>CLOSE<br>INQUIRE | verbi di tipo ausiliario |
| REWIND<br>BACKSPACE<br>ENDFILE | verbi di posizionamento |

## 5.2 Verbi che provvedono all'input-output e lista di controllo

I verbi che ci permettono l'input-output in FORTRAN 77 hanno vari formati di cui quello più tipico è il seguente:

verbo(**clist**) **iolist**

Prima di affrontare l'esame di ogni singolo verbo è utile imparare il senso di tale formato e, in particolare, della **clist**. Si tratta di una lista, detta di **controllo**, che include varie specifiche: la definizione di UNIT, il riferimento ad un FORMAT ed altre che esamineremo tra breve. Va tenuto presente che essa può anche avere un formato che già conosciamo dal precedente standard del FORTRAN.

La lista **iolist** è presente per i verbi di lettura e scrittura ed è costituita dall'elenco della variabili che vogliamo in input-output, o meglio del loro nome: può trattarsi di array e può essere presente il DO implicito. Essa ha caratteristiche identiche a quelle della lista corrispondente in FORTRAN IV.

Come si vede, la codifica dei verbi di input-output propria del vecchio standard, potrà essere spesso accettata inalterata in FORTRAN 77.

Le estensioni riguardano sostanzialmente le possibilità forniteci dalla **clist**. Le specifiche di tale lista proprie dei verbi di input-output più usati sono le seguenti:

[UNIT=]n
[FMT=]f
[REC=k]
[IOSTAT=ivar]
[ERR=$l_1$]
[END=$l_2$]

Abbiamo posto in parentesi quadra quanto può essere sottinteso; come si vede le ultime quattro specifiche devono essere scritte per intero o non devono essere scritte del tutto: in realtà ognuna di esse può essere presente o meno secondo precisi criteri che vedremo esaminandone il significato. Abbiamo inoltre sottolineato n poiché si tratta di una specifica obbligatoria.

UNIT=n — serve a specificare il nome logico del file, la UNIT appunto come si è già detto; n deve essere un numero intero positivo o lo zero e può anche trattarsi del nome di una variabile intera contenente il numero desiderato. Nel caso venga scritto solamente n, questo deve essere il primo parametro della **clist**.

FMT = f — serve a definire il FORMAT desiderato, f potrà essere la label del FORMAT definito nella stessa unità di programma, oppure potrà essere il nome di una variabile intera alla quale sia stato assegnato il valore di quella label; f potrà anche essere il nome di una variabile di caratteri contenente il FORMAT (si veda il § **2.3 Alcune note sulle variabili di caratteri**) o addirittura un'espressione di caratteri (purché non contenente l'operatore di concatenazione, né il simbolo *); infine f potrà essere il simbolo per indicare una **formattazione diretta** come vedremo nel prossimo paragrafo. Nel caso venga scritto solo f (omettendo FMT=) questo deve essere il secondo parametro della lista preceduto da una definizione di file priva di UNIT= (insomma, almeno fino a questo punto, può avere il formato proprio del FORTRAN IV).

REC = k — dovrà essere specificato solo nel caso di un accesso diretto e starà ad indicare che intendiamo accedere al k-esimo record; k dovrà essere un valore numerico intero positivo, o un'espressione a risultato intero, od il nome di una variabile intera.

IOSTAT = ivar — serve a specificare lo sato dell'operazione di input-output, poiché viene posto in ivar, che è una variabile intera, un valore numerico corrispondente a quanto accade durante l'operazione di input-output; tale codice dello stato dell'operazione sarà zero per un'operazione andata a buon fine, un valore positivo dipendente dal sistema (ogni tipo di elaboratore presenta i suoi più appropriati codici) nel caso di errori, un valore negativo per la condizione di fine-file.

ERR = $l_1$ — ha lo stesso senso che in FORTRAN IV: in caso di errore nell'operazione di input-output l'esecuzione prosegue dalla label $l_1$.

END = $l_2$ — ha lo stesso senso che in FORTRAN IV: in condizioni di fine-file il controllo passa alla istruzione associata alla label $l_2$.

### 5.3 Il verbo READ per i file esterni

Il verbo READ presenta due formati fondamentali relativamente ad un input da file esterni:

READ(**clist**)**iolist**

READ **f,iolist**

Esaminiamo il primo dei due formati: **clist** e **iolist** hanno la struttura e le caratteristiche viste nel paragrafo precedente. Così:

```
READ(5,100) A,I
```

```
READ(UNIT=5,FMT=100) A,I
```

hanno lo stesso significato e vogliono dire che si desidera leggere dall'unità di input standard del sistema le variabili A ed I con il FORMAT specificato alla label 100; se:

```
100 FORMAT(F5.2,I3)
```

sarebbe stato lo stesso scrivere:

```
READ(UNIT = 5,FMT = "(F5.2,I3)"A,I
```

oppure:

```
READ (UNIT=5,FMT=FORMT) A,I
```

dove FORMT è una variabile di caratteri precedentemente inizializzata con "(F5.2,I3)".

I parametri della **clist**, se specificati per intero, non sono posizionali, così si può scrivere:

```
READ(FMT=900,END=9,ERR=19,UNIT=3,IOSTAT=IS)...
```

oppure

```
READ(3,900,ERR=19,END=9,IOSTAT=IS)...
```

intendendo con ciò che si vogliono leggere le variabili della lista di input dal file identificato con la unità 3, secondo il formato specificato alla label 900, che in caso di errore si vuole un rimando alla label 19 ed il codice dell'errore deve venir posto nella variabile IS, ed infine che al termine delle letture il controllo deve passare alla label 9.

Come esempio di lettura di file non formattati, esaminiamo il seguente, relativo ad un accesso diretto:

```
READ(16,REC=N) I,J
```

Con esso vengono letti senza transcodifica due valori interi per ognuno dei record del file di UNIT = 16 ed assegnati alle variabili I e J; il record in lettura è identificato (o meglio lo è la sua posizione) dal valore contenuto in N.

E' possibile trattare in modo non formattato anche file sequenziali:

```
READ(UNIT=8,IOSTAT=IST) I,J,X(I,J)
```

significa che dalla UNIT 8 sono letti i tre valori di I,J e X(I,J) senza transocdifica e che un eventuale codice di errore viene posto in IST.

Nella lista di controllo si può eseguire anche una **formattazione diretta** per mezzo dell'asterisco. Questo può essere fatto quando i dati in input sono scritti *liberamente* separandoli l'uno dall'altro o con una virgola (preceduta e seguita da blank o da sola) o semplicemente con uno o più blank. Se, ad esempio, il file di input che assegneremo all'unità 3, è scritto nel modo seguente:

```
3.11  7,  ,5
711.3   1,3  6
8.2  ,  2,3  4
.....
```

è possibile scrivere:

```
READ(END=9,UNIT=3,FMT="*") X,J,M,N
```

e alla prima lettura si hanno le seguenti assegnazioni:

X=3.11  J=7  M=0  N=5

alla seconda:

X=711.3  J=1  M=3  N=6

alla terza:

X=8.2  J=2  M=3  N=4

e così via; alla fine-file il controllo passa alla label 9.

Se nel file di input viene posto / esso interrompe la lettura e le variabili di **iolist** che ancora non hanno ricevuto una assegnazione vengono poste a zero.

Il secondo dei formati visti all'inizio, relativamente al verbo READ, ha come caratteristica sostanziale il fatto che la lista di controllo viene omessa quasi per intero: rimane solo l'indicatore della label del FORMAT. Esso significa che non si intende gestire alcuna situazione di errore o fine-file e, soprattutto, che l'unità di input è quella standard di sistema.

Insomma è come scrivere UNIT=5.

Così:

READ 100,A,B,C

significa semplicemente leggere le variabili A B e C da quell'unità secondo le specifiche di FORMAT che si trovano alla label 100.

Questo formato ha, però una sua importanza particolare per quei sistemi che, dotati di un FORTRAN interattivo, permettono di lavorare per mezzo di terminali: essi danno all'utente la facoltà di introdurre il proprio lavoro, il programma, da un terminale, lanciandone di qui l'esecuzione, di qui controllando ciò che avviene e ottennendo risultati in modo immediato. In casi come questo l'unità di input standard diventa il terminale stesso e il formato del verbo READ in questione è quello che viene normalmente usato per comunicare direttamente da terminale l'input desiderato. Anzi in questa modalità di lavoro si scrivono statement del tipo:

READ, A,B,C

sottointendendo anche il FORMAT. E' come scrivere una formattazione diretta: durante l'esecuzione il programma, giungendo a questa istruzione, si interromperà, l'utente introdurrà i tre valori per A,B e C separandoli con una virgola o con un blank, quindi il programma potrà proseguire normalmente.

### 5.4 I verbi PRINT e WRITE per file esterni

L'output verso file esterni è affidato ai due verbi di scrittura qui di seguito riportati con il loro formato:

WRITE(**clist**)**iolist**

PRINT **f,iolist**

La **lista di controllo** è la stessa esaminata nel paragrafo ad essa dedicato e viene usata come nel verbo READ.

L'unica differenza consiste nella mancanza della END=$l_2$ : è infatti ovviamente senza senso il definire un rimando alla fine del file. Nel caso dell'output la segnalazione di fine-file verrà posta dopo l'ultimo record scritto alla fine del trattamento del file stesso. Più precisamente, all'atto della chiusura del file verrà inserito (da parte del sistema ed in modo del tutto automatico) un ultimo record fittizio recante, appunto, la segnalazione di fine. Ciò, almeno, è quanto avviene per i file sequenziali; per quelli ad accesso diretto va tenuto presente che tale segnalazione non è comunque necessaria per la loro logica di trattamento e che dunque il record di fine-file normalmente non viene inseri-

to. Ancora esiste una certa compatibilità con il verbo WRITE del FORTRAN IV, e così scrivere:

```
WRITE(6,100,ERR=2)...
```

oppure

```
WRITE(UNIT=6,ERR=2,FMT=100)...
```

è lo stesso e significa inviare i valori elencati nella **iolist** verso la unità di output standard (normalmente la stampante) secondo le direttive del FORMAT di label 100, in caso di errore si avrà il rimando alla label 2.

I file ad accesso diretto richiedono la specifica REC=:

```
WRITE(9,REC=I,IOSTAT=IS) A,B
```

significa scrivere i valori di A e B nell'I-esimo record di UNIT=9, un eventuale codice di errore verrà posto in IS. Anche nel caso dell'output la definizione del numero ordinale del record per mezzo di REC=k è ammessa solo per i file ad accesso diretto ed in tal caso, anzi, è obbligatoria.

Anche i file ad accesso sequenziale possono essere non formattati:

```
WRITE(UNIT=10)I,L,M
```

significa scrivere i valori di I, L ed M senza transcodifica sul file sequenziale di UNIT 10.

Per quanto riguarda i file formattati va tenuto presente che la specifica FMT permette una più diretta codifica dell'output quando si abbiano da definire delle costanti di caratteri.

Si osservi il seguente esempio:

```
WRITE(UNIT=6,FMT=" ('VAR.A=',F5.2)") A
```

il FORMAT è stato inserito come costante di caratteri ed al suo interno è stata inserita a propria volta la costante di caratteri 'VAR.A='.

Anche nel caso dell'output è possibile una formattazione diretta e valgono le regole generali viste per l'input: i valori emessi saranno separati l'uno dall'altro da un blank e, in caso di stampa, verrà scritta una riga per record. I formati secondo i quali verrà fatta la scrittura saranno quelli standard di sistema.

Un'ultima forma interessante della scrittura del verbo di output è quella abbreviata. Essa è possibile quando si desidera utilizzare l'unità di output standard, grazie al verbo PRINT:

PRINT 190,A,B,C

significa inviare in stampa le variabili A B e C come è specificato nel FORMAT di label 190. Ancora per quei sistemi che permettono di lavorare con un FORTRAN interattivo, la risorsa di output standard diventa il terminale. In tal caso è possibile anche sottintendere il FORMAT:

PRINT,A,B,C

e ancora la stampa avverrà secondo i formati standard del sistema.

*Esempio*:

Vediamo, a questo punto, un esempio (1) di programma che legga un file sequenziale contenente dei dati numerici (non sappiamo quanti siano, sappiamo solo che si tratta di interi di 3 cifre scritti uno per record). Nel caso di un errore di lettura desideriamo che compaia, in output, una segnalazione. Desideriamo che di ogni numero positivo sia calcolata la radice quadrata e cubica e che numero e radici vengano stampati; per i numeri negativi si manderà in stampa lo zero al posto delle radici.

```
      PROGRAM LETTWR
C LETTURA DEI NUMERI DEL FILE "19"
C SCRITTA DI OGNUNO DI ESSI E
C DELLE RADICI QUADRATE E CUBICHE.
      IUN = 19
      ISTERR = 0
    3 READ (FMT="(I3)",UNIT=IUN,ERR=9,IOSTAT=ISTERR,END=99) K
      IF    (K.GT.0)THEN
            R2=SQRT (K)
            R3=K**(1./3.)
      ELSE
            R2=0
            R3=0
      END IF
      WRITE (UNIT=6,FMT=200) K,R2,R3
  200 FORMAT ("N=",I3,"R2=",F7.4"R3=",F7.4)
      GO TO 3
    9 PRINT 100,ISTERR
  100 FORMAT("****SI HA UN ERRORE DI TIPO",13,"***")
   99 STOP
      END
```

(1) Per questo esempio, e per gli altri che seguono, si è utilizzato, per l'esecuzione effettiva, un sistema DPS/8 Honeywell.

con un input come il seguente:

```
025
111
000
-36
8 1
625
049
144
027
-01
001
```

si ottiene in output:

```
N= 25 R2= 5.0000 R3= 2.9240
N=111 R2=10.5357 R3= 4.8059
N=  0 R2= 0.     R3= 0.
N=-36 R2= 0.     R3= 0.
N=801 R2=28.3019 R3= 9.2870
N=625 R2=25.0000 R3= 8.5499
N= 49 R2= 7.0000 R3= 3.6593
N=144 R2=12.0000 R3= 5.2415
N= 27 R2= 5.1962 R3= 3.0000
N= -1 R2= 0.     R3= 0.
N=  1 R2= 1.0000 R3= 1.0000
```

mentre con l'input:

```
025
111
000
-36
 81
625
 49
144
AAA
-01
001
```

si ottiene:

```
N= 25 R2= 5.0000 R3= 2.9240
N=111 R2=10.5357 R3= 4.8059
N=  0 R2= 0.     R3= 0.
N=-36 R2= 0.     R3= 0.
N= 81 R2= 9.0000 R3= 4.3267
N=625 R2=25.0000 R3= 8.5499
N= 49 R2= 7.0000 R3= 3.6593
N=144 R2=12.0000 R3= 5.2415
* * * SI HA UN ERRORE DI TIPO 32 * * *
```

come si può notare il nono record (che contiene "AAA") genera una situazione di errore di codice (nel nostro sistema) 32.

### 5.5 I verbi READ e WRITE per file interni

Per i file interni i verbi di lettura e scrittura sono ancora READ e WRITE. In questo caso si parla di input o di output soltanto in senso lato: infatti il file, come si è già detto, è presente in memoria (di fatto è una zona di memoria) e la funzione dei due verbi è quella di trasportare il contenuto in un'altra zona di memoria transcodificandolo.

Il verbo READ serve per passare da un formato a caratteri ad un formato numerico, l'*input* è costituito dal file interno (una variabile od un array di caratteri numerici o, al più, il punto decimale), la **iolist** è costituita dall'elenco delle variabili numeriche di destinazione.

Il verbo WRITE serve per passare da un formato numerico ad un formato a caratteri, l'*output* è costituito dal file interno e la **iolist** dall'elenco delle variabili numeriche emittenti.

Il formato della **clist** è quello che già conosciamo (e che vale per i due verbi relativamente a file esterni), ma con l'esclusione di IOSTAT.

Il formato dei verbi è dunque:

READ(**clist**)**iolist**

WRITE(**clist**)**iolist**

dove **clist** è:

[UNIT=]var
[FMT=]f
[ERR=$l_1$]
[END=$l_2$]

abbiamo sottolineato var e f in quanto sono due specifiche che vanno comunque espresse, tutto il resto è facoltativo e perciò è stato posto in parentesi quadra.

Il riconoscimento del fatto che si tratta di un file interno avviene per mezzo della UNIT; vediamolo esaminando le varie specifiche:

UNIT=var è il nome della variabile o dell'array di caratteri che costituisce il file interno: per il verbo READ si tratterà dell'input, per il verbo WRITE dell'output;

FMT= f è il riferimento al FORMAT relativo alla **iolist**; esso è particolarmente importante in questo caso poiché definisce in quale modo deve avvenire la transcodifica;

ERR=$l_1$

END= $l_2$ hanno l'usuale e già visto significato; si noti però che ora la END è utilizzabile anche nella WRITE e che comunque ha un significato particolare e leggermente diverso da quello relativo a file esterni: nella WRITE si avrà una condizione di END quando lo spazio disponibile sul file interno è terminato prima che tutte le variabili di **iolist** siano state transcodificate e trasferite; nella READ, quando lo spazio del file interno è esaurito prima che tutte le variabili di **iolist** siano state riempite.

Vediamo un esempio di transcodifica da formato a caratteri a quello numerico:

```
      CHARACTER*11 A
      A="12.3945.6.1"
      ...
      READ(UNIT=A,FMT=2) X,I,Y,Z
    2 FORMAT(F4.1,I1,F4.1,F2.1)
      ...
```

le variabili della **iolist** conterranno i valori riportati di seguito, in formato numerico:

X=12.3 I=9 Y=45.6 Z=0.1

La variabile A conserva inalterato il proprio contenuto ed è ancora normalmente utilizzabile.

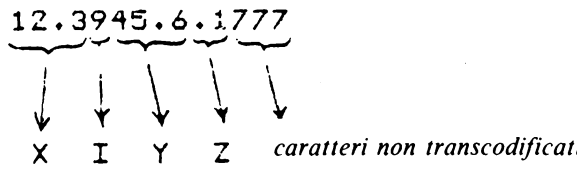
Come si vede è il FORMAT a determinare il modo in cui devono essere scanditi i vari caratteri e dunque quali siano i valori da assegnare alle variabili della **iolist**.

Se il campo emittente contiene un numero maggiore di caratteri di quelli definiti nel FORMAT delle variabili riceventi, in essi rimangono, dopo la READ, dei caratteri non transcodificati e non assegnati ad alcuna variabile. Di contro se il campo emittente è troppo corto per produrre tutte le assegnazioni delle variabili presenti nella lista (il numero di caratteri del FORMAT è maggiore di quello del file interno) si ha la già vista condizione di END e le variabili su cui non ha agito la transcodifica non risultano inizializzate (come si dice sono a contenuto *indefinito*) oppure hanno l'eventuale precedente valore.

Se, ad esempio, si avesse A di 15 caratteri:

```
      CHARACTER*15 A
      A="12.3945.6.17777"
      ...
      READ(UNIT=A,FMT=2) X,I,Y,Z
    2 FORMAT(F4.1,I1,F4.1,F2.1)
      ...
```

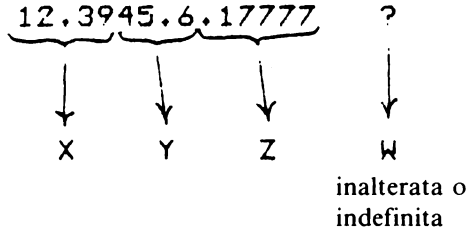
tutto avverrebbe secondo il seguente schema:




Se poi si avesse una lettura del tipo (la A è la stessa di cui sopra):

```
      READ(UNIT=A,FMT=2) X,Y,Z,W
2     FORMAT(F5.2,F4.1,F6.5,F5.2)
```

tutto avverrebbe secondo il seguente schema:




La transcodifica da valori numerici a caratteri viene eseguita per mezzo del verbo WRITE. Nella UNIT si specificherà il file interno e le variabili numeriche saranno elencate nella **iolist**. Così:

```
      CHARACTER *20 A
      ...
      X=1.1
      I=9
      Y=2.2
      Z=0.1
      WRITE(UNIT=A,FMT=3) X,I,Y,Z
3     FORMAT(F4.1,"I=",I1,"*",F3.1,"*-*-*",F3.1,*"0")
      ...
```

produce in A i caratteri:

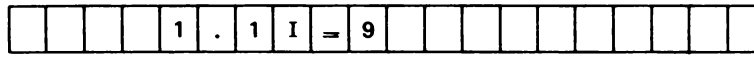
| | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 1 | . | 1 | I | = | 9 | * | 2 | . | 2 | * | - | * | - | * | 0 | . | 1 | 0 |

Si tenga presente che se il numero previsto per i caratteri del file interno (ora *ricevente*) è maggiore di quello specificato dal FORMAT delle variabili

emittenti, si ha un accostamento a sinistra dei caratteri transcodificati ed un riempimento a destra con blank (così se si vogliono dei blank a sinistra, essi vanno inseriti esplicitamente). Se nell'esempio precedente la scrittura ed il suo FORMAT fossero stati:

```
      WRITE(UNIT=A,FMT=3) X,I
3     FORMAT("    ",F4.1,"I=",I1)
```

si sarebbe ottenuto:

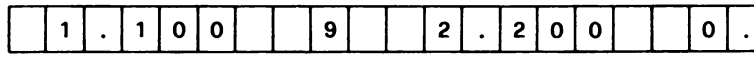



qualunque fosse il precedente contenuto di A.

Nel caso contrario, in cui il campo ricevente termina prima di aver completato la transcodifica, si verifica, come si è già visto, la condizione di END e si ha un troncamento. Se, sempre con i valori dell'esempio precedente, si avesse:

```
      WRITE(UNIT=A,FMT=3) X,I,Y,Z
3     FORMAT(F6.3,"    ",I1,2F7.3)
```

in A verrebbe posto:




con la perdita della parte decimale di Z.

Per concludere: si tenga presente che sui file interni è possibile agire con i soli verbi di READ e WRITE. Ogni altro verbo di input-output è vietato; tuttavia la variabile che costituisce il file interno è normalmente trattabile con tutti gli altri statement, come, appunto, ogni normale variabile di caratteri.

## 5.6 I verbi ausiliari OPEN e CLOSE

Come si è già avuto occasione di notare, in FORTRAN 77 è possibile richiedere le aperture e le chiusure dei file esterni in modo esplicito. A queste funzioni sono devoluti due verbi ausiliari relativi all'input-output, che ora esamineremo: si tratta dei verbi OPEN e CLOSE.

Si è definita l'apertura come un insieme di operazioni tra cui, sostanzialmente, una serie di controlli e la connessione del file esterno ad una determinata UNIT.

Ciò è ottenuto per mezzo del verbo OPEN. Il suo formato è:

OPEN(**clist**)

Esso può essere usato tanto per eseguire una prima connessione quanto per cambiare certe specifiche relative ad un file che sia già stato connesso.

In quest'ultimo caso il file dovrà venir chiuso e poi riaperto con le nuove specifiche.

La lista di controllo, questa volta, differisce dalle precedenti in modo notevole; le possibili specifiche sono:

[UNIT=]n
[ERR=l]
[IOSTAT=ivar]
[FILE=nome]
[STATUS=stt]
[ACCESS=aexp]
[FORM=frm]
[RECL=h]
[BLANK=bexp]

dove:

UNIT=n — serve, come al solito, a definire la connessione ad una certa unità; **n** è l'unico parametro obbligatorio di tutta la **clist**;

ERR=l — è l'usuale specifica per il trasferimento di controllo in caso di errore;

IOSTAT=ivar — è, ancora, l'usuale specifica per il deposito, in **ivar**, di un eventuale codice di errore;

FILE=nome — serve per definire la connessione di un file di nome "**nome**" con l'unità specificata nella UNIT; "**nome**" è una espressione di caratteri (racchiusi dunque tra apici) e deve coincidere con il nome di un file conosciuto dal sistema; questa specifica è particolarmente utile quando sia possibile lavorare con file su disco permanentemente a disposizione dell'utente e battezzati, con un nome appunto, riconoscibile da parte del sistema;

STATUS=stt — tale specifica è relativa al modo in cui si riserva lo spazio relativo ad un file; "**stt**" è una espressione di caratteri (racchiusa tra apici) da scegliersi tra le seguenti:
"**OLD**": indica che il file è già esistente; si trova su disco, è

permanente ed il suo nome è quello specificato nella FILE (che diviene dunque obbligatoria);
"**NEW**": lo spazio su disco ancora non è stato riservato, ma lo sarà da questo momento e si disporrà, d'ora in avanti, di un file permanente con nome uguale a quello specificato nella FILE (obbligatoria); corrisponde a ciò che viene comunemente detto *creazione del file permanente*;
"**SCRATCH**": lo spazio ancora non esiste, il file viene creato in quel momento senza un nome ( e dunque è vietata la presenza della FILE), all'atto della chisura il file verrà rilasciato: si tratta di un file **temporaneo** la cui vita si protrae solo per il tempo che intercorre tra apertura e chisura;
"**UNKNOWN**": non si desidera specificare un particolare STATUS, una sua definizione sarà automatica e prodotta secondo criteri specifici strettamente dipendenti dal tipo di sistema su cui si lavora; quando la specifica di STATUS è omessa viene assunto automaticamente questo parametro;

ACCES = aexp serve a specificare, come dice il nome, il tipo di accesso; "**aexp**" è un'espressione di caratteri (racchiusa tra apici) che può essere:
"**SEQUENTIAL**": per un accesso sequenziale;
"**DIRECT**": per un accesso diretto;
nel caso tale specifica venga omessa, viene assunto un accesso sequenziale;

FORM = frm serve a specificare se il file è formattato o meno; "**frm**" è una espressione di caratteri (racchiusa tra apici) che può essere:
"**FORMATTED**": per file formattati; se FORM è omessa ed il file è sequenziale viene assunto tale parametro;
"**UNFORMATTED**": per file non formattati; se FORM è omessa ed il file è ad accesso diretto viene assunto tale parametro;

RECL = h serve a specificare la lunghezza dei record, h è un'espressione intera che definisce questa lunghezza; tale specifica va usata per file ad accesso diretto e solo in tal caso; se il file è formattato la lunghezza è misurata in caratteri, se il file non è formattato l'unità di misura normalmente è la **parola** (ma in generale tale unità dipende dal sistema);

BLANK = bexp serve a gestire i blank eventualmente presenti in un file di input; "**bexp**" è un'espressione di caratteri (tra apici) e può essere:
"**NULL**": i blank vengono ignorati a meno che tutto il record non contenga solo blank: in tal caso il contenuto è interpretato come 0;

"**ZERO**": i blank prima di un numero vengono ignorati, quelli a seguito di un numero, o inseriti tra le varie cifre, sono interpretati come 0;
nel caso la specifica venga omessa viene assunto "NULL".

Per eseguire la *sconnessione* da una certa UNIT, ovvero la chiusura di un file si utilizza il verbo CLOSE (1); esso ha il formato:

CLOSE(**clist**)

e la **clist** può essere composta da:

[UNIT=]n
[ERR=l]
[IOSTAT=ivar]
[STATUS=sta]

UNIT=n è l'usuale specifica di unità ed n è ancora l'unico parametro obbligatorio;

ERR=l è l'usuale specifica per le situazioni d'errore;

IOSTAT=ivar è l'usuale specifica per la gestione di un codice d'errore;

STATUS=sta è ancora riferita allo spazio relativo al file, ma ha un senso diverso che nella OPEN; infatti "**sta**" (che è ancora un'espressione di caratteri) può essere:
"**KEEP**": sta a specificare che il file continua ad esistere dopo la CLOSE (e non fa differenza che si tratti di uno permanente dichiarato OLD all'atto dell'apertura o di un file creato attraverso una NEW); tale parametro non deve essere usato per un file temporaneo, quando cioè in apertura sia stato definito SCRATCH;
"**DELETE**": sta a significare che il file deve comunque essere rilasciato;
nel caso lo STATUS venga omesso viene assunto KEEP (eccettuati i file SCRATCH che vengono comunque rilasciati).

(1) Per un file sequenziale in output esso ha anche la funzione di provocare l'inserimento del record di segnalazione di fine file.

Si è già detto che le aperture e le chiusure possono essere implicite, il che corrisponde ad omettere per intero le clausole di OPEN e CLOSE. In tal caso, si è detto, la connessione avviene al primo statement di input-output e la sconnessione alla fine del programma. I parametri che vengono assunti in questi casi sono quelli che abbiamo segnalato volta per volta come impliciti, ovvero quelli che verrebbero assunti quando manca la singola specifica.

*Esempio 1:*

Riprendiamo l'esempio del paragrafo 5.4 e vediamo cosa si sarebbe dovuto scrivere se avessimo voluto procedere ad una connessione esplicita del file "19" in modo da poter eseguire un controllo su eventuali errori di "apertura" e, anche, definire il file attraverso il suo nome di sistema, dichiararne il tipo, dichiarare che eventuali blank inseriti nelle cifre che compongono i numeri (non in testa) vanno intesi come zeri.

```
      PROGRAM LETTWR
C LETTURA DEI NUMERI DEL FILE "19"
C SCRITTURA DI OGNUNO DI ESSI E
C DELLE RADICI QUADRATE E CUBICHE.
      CHARACTER OLDNEW *7, ZERNUL *4
      OLDNEW="OLD"
      ZERNUL="ZERO"
      IUN=19
      ISTERR=0
      OPEN    (UNIT=IUN, ERR=8, IOSTAT=ISTERR, FILE="/CATALOGO/NOMEFILE",
     1         STATUS=OLDNEW, ACCESS="SEQUENTIAL", FORM="FORMATTED",
     2         BLANK=ZERNUL)
   3  READ    (FMT="(I3)", UNIT=IUN, ERR=9, IOSTAT=ISTERR, END=98) K
      IF    (K.GT.0) THEN
            R2=SQRT (K)
            R3=K** (1/3)
       ELSE
            R2=0
            R3=0
      END IF
      WRITE (UNIT=6, FMT=200) K, R2, R3
 200  FORMAT ("N=",I3, "R2=", F7.4, "R3=", F7.4)
      GO TO 3
   8  PRINT 300, ISTERR
 300  FORMAT ("ERRORE DI OPEN - TIPO", I3)
      GO TO 99
   9  PRINT 100, ISTERR
 100  FORMAT ("*** SI HA UN ERRORE DI TIPO", I3, " * * *")
  98  CLOSE (UNIT=IUN, STATUS="KEEP")
  99  STOP
      END
```

con un input come il seguente:

```
025
111
000
-36
8 1
625
049
144
027
-01
001
```

si ottiene in output:

```
N= 25 R2= 5.0000 R3= 2.9240
N=111 R2=10.5357 R3= 4.8059
N=  0 R2= 0.     R3= 0.
N=-36 R2= 0.     R3= 0.
N=801 R2=28.3019 R3= 9.2870
N=625 R2=25.0000 R3= 8.5499
N= 49 R2= 7.0000 R3= 3.6593
N=144 R2=12.0000 R3= 5.2415
N= 27 R2= 5.1962 R3= 3.0000
N= -1 R2= 0.     R3= 0.
N=  1 R2= 1.0000 R3= 1.0000
```

modificando, invece, nel modo che segue l'istruzione di apertura:

```
      OPEN(UNIT=IUN,ERR=8,IOSTAT=ISTERR)
```

con lo stesso input si ottiene la stampa:

```
* * * SI HA UN ERRORE DI TIPO 33 * * *
```

infatti, nel nostro sistema, il codice di errore 33 sta per "tentativo di lettura su file aperto in output" che è quanto abbiamo implicitamente provocato omettendo STATUS: viene assunto UNKNOWN che sempre per il nostro sistema, sottointende output quando manchi anche la specifica FILE. Cambiando, poi, l'istruzione di apertura così:

```
      OPEN(UNIT=IUN,ERR=8,IOSTAT=ISTERR,
     1     STATUS=OLDNEW,ACCESS="SEQUENTIAL",FORM="FORMATTED",
     2     BLANK=ZERNUL)
```

si ottiene in stampa:

```
ERRORE DI OPEN - TIPO 47
```

mancando, infatti, la specifica FILE (d'altro canto obbligatoria quando sia presente uno STATUS OLD) il sistema ci segnala, con il codice 47, la assenza del file permanente da connettere alla UNIT 19.

*Esempio 2:*

Nel sistema sono presenti dei file con nomi del tipo: NOME10, NOME11, ..., NOME21. Non sappiamo quali di questi esistano e quali no. Desideriamo esaminare quelli esistenti stampandone i primi tre record (dovrebbero contenerne un numero maggiore) e segnalare i casi in cui ve ne siano di meno. Vogliamo anche segnalare gli altri possibili errori con frasi opportune.

```
   CHARACTER NM *2, FLNOME *6, SK *80
   I=10
 9 ISTT=0
   WRITE(UNIT=NM,FMT=1) I
 1 FORMAT(I2)
   FLNOME="NOME"//NM
   OPEN(UNIT=07,STATUS="OLD",FILE=FLNOME,ERR=4)
   DO 2 K=1,3
   READ(UNIT=07,FMT=10,IOSTAT=ISTT,ERR=7,END=5) SK
10 FORMAT(A80)
 2 WRITE(UNIT=06,FMT=12,IOSTAT=ISTT,ERR=8) K,FLNOME,SK
12 FORMAT(" REC. NUM.",I2," DEL FILE ",A7,/" --->",A80)
   CLOSE(UNIT=07,STATUS="KEEP")
   GO TO 6
 4 WRITE(06,13) FLNOME
13 FORMAT(" IL FILE ",A7," NON ESISTE")
 6 I=I+1
   IF(I.GT.21) THEN
     WRITE(06,11)
11   FORMAT(" SONO STATI  ESAMINATI 12 NOMI - STOP")
     STOP
   ELSE
     WRITE(UNIT=06,FMT="(' ')")
   ENDIF
   GO TO 9
 5 WRITE(UNIT=06,FMT=50) FLNOME
   CLOSE(UNIT=07)
   GO TO 6
 7 WRITE(UNIT=06,FMT=60) FLNOME
   GO TO 6
 8 WRITE(06,70)  FLNOME
   GO TO 6
50 FORMAT(" IL FILE ",A7," CONTIENE MENO DI 3 RECORD.")
60 FORMAT(" IL FILE ",A7," NON E' LEGGIBILE")
70 FORMAT(" IL FILE ",A7," NON PUO' ESSERE STAMPATO")
   END
```

In realtà esistono soltanto i seguenti quattro file di cui possiamo osservare i primi record (il file NOME12 è composto di soli due record):

FILE NOME 10

```
AAAAA 3124563218975257777.68468456854 000
AA01A 9845216356485228956.84984955555 000
AA02A 5463256321589582986.56956955777 000
AA03A 2984726543878787812.89245733333 000
AA04A 1258754612265652154.16885144411 000
.............
```

FILE NOME 11

```
01AAA 66666666          77.6            01
02N1A 55555555          56.8            01
03L2A 4444444444         6.5            01
04P3A 11111                             01
10J4A 125555           154.1            01
..............
```

FILE NOME 12

```
01AAA 66666666          77.6            02
02N1A 55555555          56.8            02
```

FILE NOME 15

```
01XXX 00000006           .6             05
02N1X 00000066           .8             05
03L2X 00000001           .5             05
04P3X 00000020           .5             05
10J4X 26548456          1.1             05
..............
```

l'output ottenuto con il lancio del programma è riportato qui di seguito:

```
REC. NUM. 1 DEL FILE  NOME10
--->AAAAA 3124563218975257777.68468456854 000
REC. NUM. 2 DEL FILE  NOME10
--->AA01A 9845216356485228956.84984955555 000
REC. NUM. 3 DEL FILE  NOME10
--->AA02A 5463256321589582986.56956955777 000

REC. NUM. 1 DEL FILE  NOME11
--->01AAA 66666666          77.6            01
REC. NUM. 2 DEL FILE  NOME11
--->02N1A 55555555          56.8            01
REC. NUM. 3 DEL FILE  NOME11
--->03L2A 4444444444         6.5            01

REC. NUM. 1 DEL FILE  NOME12
--->01AAA 66666666          77.6            02
REC. NUM. 2 DEL FILE  NOME12
--->02N1A 55555555          56.8            02
IL FILE  NOME12 CONTIENE MENO DI 3 RECORD.
```

```
IL FILE  NOME13 NON ESISTE

IL FILE  NOME14 NON ESISTE

REC. NUM. 1 DEL FILE  NOME15
--->01XXX 00000006            .6              05
REC. NUM. 2 DEL FILE  NOME15
--->02N1X 00000066            .8              05
REC. NUM. 3 DEL FILE  NOME15
--->03L2X 00000001            .5              05

IL FILE  NOME16 NON ESISTE

IL FILE  NOME17 NON ESISTE

IL FILE  NOME18 NON ESISTE

IL FILE  NOME19 NON ESISTE

IL FILE  NOME20 NON ESISTE

IL FILE  NOME21 NON ESISTE
SONO STATI  ESAMINATI 12 NOMI - STOP
```

Come si noterà i file non esistenti sono stati rilevati per mezzo della gestione di errore sulla OPEN. La condizione di fine-file (END = 5) sulla READ ci ha permesso di riconoscere che il file FILE12 contiene solo 2 record.

### 5.7 Il verbo ausiliario INQUIRE

Un altro verbo ausiliario di notevole importanza è INQUIRE. Esso viene usato per determinare le caratteristiche e le proprietà di un file: lo stato della connessione, il tipo di accesso, la formattazione o meno e via dicendo. Tali informazioni verranno messe a disposizione dell'utente secondo precisi criteri ed in funzione di quanto specificato nella **clist** del verbo.

Esso ha il formato:

INQUIRE(**clist**)

e la **clist** permette le seguenti specifiche:

$$\begin{cases} \text{UNIT} = \text{n} \\ \text{FILE} = \text{nome} \end{cases}$$

$$\left\{\begin{array}{l}
[\text{ERR} = \text{l}] \\
[\text{IOSTAT} = \text{ivar}_1] \\
[\text{EXIST} = \text{lvar}_1] \\
[\text{OPENED} = \text{lvar}_2] \\
[\text{NAMED} = \text{lvar}_3] \\
[\text{NAME} = \text{cvar}_1] \\
[\text{NUMBER} = \text{ivar}_2] \\
[\text{ACCESS} = \text{cvar}_2] \\
[\text{SEQUENTIAL} = \text{cvar}_3] \\
[\text{DIRECT} = \text{cvar}_4] \\
[\text{FORM} = \text{cvar}_5] \\
[\text{FORMATTED} = \text{cvar}_6] \\
[\text{UNFORMATTED} = \text{cvar}_7] \\
[\text{RECL} = \text{ivar}_3] \\
[\text{NEXTREC} = \text{ivar}_4] \\
[\text{BLANK} = \text{cvar}_8]
\end{array}\right.$$

La parentesi graffa comprende la UNIT e la FILE sta a significare che potrà essere presente una ed una sola delle due specifiche: quale sia il file da esaminare verrà determinato o precisandone l'unità, e si parlerà di **inquire by unit**, o definendone il nome, e si parlerà di **inquire by file**.

La seconda parentesi graffa sta ad indicare che è utilizzabile una sola specifica per volta e che quindi dovremo scrivere una INQUIRE per ognuna delle caratteristiche che intendiamo esaminare.

L'esame del file, in una **inquire by unit**, può essere fatto sia che il file sia connesso, sia che non lo sia, addirittura può non esistere l'unità dichiarata: sarà la successiva specifica a permettere di informarci sulla situazione nella maniera più appropriata.

Allo stesso modo, in una **inquire by file**, il file dichiarato potrà anche non esistere ed ancora la specifica condurrà alle opportune informazioni.

Esaminiamo ora le singole specifiche precisando che verranno tralasciate quelle già viste nelle precedenti elencazioni. Si tratta delle: UNIT, FILE, ERR, IOSTAT che hanno lo stesso significato analizzato per i verbi dei precedenti paragrafi.

EXIST = $\text{lvar}_1$ — **$\text{lvar}_1$** è una variabile logica che al termine dell'operazione di INQUIRE contiene il valore **.TRUE.** se il file specificato esiste, **.FALSE.** in caso contrario;

OPENED = $\text{lvar}_2$ — **$\text{lvar}_2$** è una variabile logica che al termine dell'operazione di INQUIRE contiene il valore **.TRUE.** se il file specificato è *aperto*, altrimenti contiene il valore **.FALSE.**;

NAMED = $lvar_3$ — **$lvar_3$** è una variabile logica che dopo l'INQUIRE contiene il valore **.TRUE.** se il file ha un nome (è permanente), altrimenti contiene il valore **.FALSE.**;

NAME = $cvar_1$ — **$cvar_1$** è una variabile di caratteri in cui viene posto il nome del file qualora esso ne abbia uno; se il file non ha nome, **$cvar_1$** non risulta definito;

NUMBER = $ivar_2$ — **$ivar_2$** è una variabile intera in cui viene posto il valore numerico corrispondente all'unità se il file è connesso, altrimenti non risulta definita;

ACCESS = $cvar_2$ — **$cvar_2$** è una variabile di caratteri in cui viene posto il contenuto **SEQUENTIAL** o **DIRECT** a seconda che il file sia connesso ad accesso sequenziale o diretto; se il file non è connesso **$cvar_2$** non risulta definita;

SEQUENTIAL = $cvar_3$ — **$cvar_3$** è una variabile di caratteri in cui viene posto il contenuto **YES** se il file permette un accesso sequenziale (va notato che, in alcuni casi, file creati per un accesso sequenziale permettono anche un accesso diretto o viceversa, ciò dipende strettamente dal sistema); vi viene posto il contenuto **NO** se l'accesso sequenziale non è permesso; oppure il contenuto **UNKNOWN** se il sistema non è in grado di determinare se questo accesso è permesso;

DIRECT = $cvar_4$ — **$cvar_4$** è una variabile di caratteri in cui viene posto uno dei contenuti: **YES**, **NO** o **UNKNOWN** a seconda che il file permetta l'accesso diretto, non lo permetta o che il sistema non sia in grado di determinarlo;

FORM = $cvar_5$ — **$cvar_5$** è una variabile di caratteri in cui viene posto **FORMATTED** o **UNFORMATTED** per file rispettivamente formattati o non formattati; se il file non è connesso **$cvar_5$** risulta indefinita;

FORMATTED = $cvar_6$ — **$cvar_6$** è una variabile di caratteri in cui viene posto **YES** se le caratteristiche del file permettono che esso sia formattato, **NO** se non lo permettono e **UNKNOWN** se il sistema non è in grado di determinarlo; infine, è indefinita per un file non connesso;

UNFORMATTED = $cvar_7$ — **$cvar_7$** è una variabile di caratteri in cui viene posto **YES** se è permesso che il file sia non formattato, **NO** in caso contrario e **UNKNOWN** se non c'è possibilità di decisione; è indefinita per un file non connesso;

RECL = $ivar_3$ — **$ivar_3$** è una variabile intera in cui viene posta la lunghezza del record del file in esame; è utilizzabile solo nel caso di un accesso diretto ed ancora se il file è formattato è espressa in caratteri, se non è formattato è in parole (o in altra unità propria del sistema); se il file non è connesso risulta indefinita;

NEXTREC = $ivar_4$ — ancora è utilizzabile solo per file ad accesso diretto, **$ivar_4$** è una variabile intera in cui viene posto il numero ordinale del prossimo record (dopo quello appena trattato); se il file è stato aperto, ma ancora non è stata eseguita alcuna READ né alcuna WRITE, in essa viene posto il valore 1; essa risulta indefinita quando il file non è connesso o quando si sia verificata una condizione di errore;

BLANK = $cvar_8$ — **$cvar_8$** è una variabile di caratteri in cui viene posto **NULL** o **ZERO** a seconda di quanto determinato da questa stessa specifica nel verbo di OPEN (o dalla apertura implicita); risulterà indefinita per file non-formattati o non connessi.

*Esempio 1:*

Si desidera totalizzare una serie di valori numerici scritti su dei file. Questi hanno la seguente struttura:

FILE - A

```
010 02.14
020 05.05
030 01.22
040 01.15
050 03.55
060 05.05
070 01.06
080 01.44
090 04.04
100 01.25
110 08.07
120 06.03
130 01.01
140 01.05
150 01.25
160 09.07
170 01.27
```

FILE - B

```
010 01.06
020 01.04
030 01.04
040 01.08
050 01.02
060 01.03
070 01.06
080 01.04
090 -1.01
100 -1.06
110 -1.05
.......
```

FILE - C

```
010 25.36
020 65.25
030 85.14
040 65.28
050 89.12
060 00.23
070 12.36
080 52.34
090 89.41
100 15.36
110 45.25
120 82.14
130 65.21
140 19.22
150 05.93
160 22.36
170 54.37
180 85.21
```

I valori da totalizzare partono dal quinto carattere di ogni file (le prime tre cifre rappresentano una semplice numerazione sequenziale). Nel caso che durante una prima lettura (l'allocazione del primo file va fatta connettendolo alla UNIT 9) si incontrino dei valori negativi, la totalizzazione va interrotta, si deve chiudere il file e passare alla lettura di quello successivo (connettendolo, questa volta, alla UNIT 19). Nella stampa finale accanto al totale va emessa una segnalazione che evidenzi se è stato utilizzato un unico file o anche quello alternativo.

Per ottenere ciò possiamo utilmente ricorrete ad una INQUIRE by unit che ci permetta di verificare se, alla fine, la UNIT 9 sia ancora connessa (specifica OPENED) o meno; il programma può essere scritto nel modo seguente, (si osservi, in particolare, l'istruzione di label 2 e la IF successiva):

```
      PROGRAM INQUNE
      CHARACTER *27 NTT
      LOGICAL L
      IUN=9
    1 READ(UNIT=IUN,FMT="(4X,F5.2)",END=2) X
      IF(X.LT.0) THEN
        IUN=19
        CLOSE(UNIT=09)
      ELSE
        TOT=TOT+X
      ENDIF
      GO TO 1
    2 INQUIRE(UNIT=9,OPENED=L)
      IF(L) THEN
        NTT='  ***** UNICO FILE IN INPUT'
      ELSE
        NTT=' INPUT CON FILE ALTERNATIVO'
      ENDIF
      WRITE(6,100) NTT,TOT
  100 FORMAT(A27,' *** TOT.=',F7.2)
      STOP
      END
```

Quando il primo file in input è FILE-A (o, meglio, quando alla UNIT 9 viene connesso FILE-A), la totalizzazione procede fino alla fine-file e la stampa conclusiva è:

```
***** UNICO FILE IN INPUT *** TOT.=  53.70
```

se, invece, FILE-B è il primo ad essere letto, non appena il programma giunge al primo valore negativo la UNIT 9 viene chiusa e la totalizzazione prosegue su FILE-C (UNIT 19). La stampa finale è allora:

```
INPUT CON FILE ALTERNATIVO *** TOT.= 887.91
```

Si noterà che in questo esempio la connessione dei tre file è stata fatta esternamente al programma (e non con una OPEN che collegasse i nomi

FILE-A, FILE-B e FILE-C alle UNIT 9 o 19). Come si proceda nel fare ciò è strettamente legato al sistema su cui si lavora e, dunque, non ci riguarda in questa sede.

*Esempio 2:*

Il file di nome NOME10 (che è uguale al file FILE-C dell'esercizio precedente) contiene dei valori numerici che devono essere utilizzati in una certa elaborazione. La UNIT di input deve essere gestibile dinamicamente, deve, anzi, essere definibile al momento del lancio. Per questo è bene scrivere un programma interattivo che inizi con una lettura, dalla risorsa standard di sistema (label 1), che ne fornisce il valore. La elaborazione avviene in una subroutine che tra i suoi argomenti contiene anche la variabile IU utilizzata proprio per la definizione di UNIT (label 9). Al ritorno da tale subroutine il valore di IU potrebbe anche essere cambiato. Perciò, se si vuole che alla fine compaia una segnalazione riguardante, tra l'altro, l'ultima UNIT utilizzata, è necessario ricavarne il valore con la specifica NUMBER in una istruzione INQUIRE (label 2).

```
      PROGRAM INQFLE
      WRITE(06,FMT="(' DIMMI LA UNIT DA USARE')")
    1 READ(UNIT=05,FMT="(I2)") IU
      OPEN(UNIT=IU,FILE="NOME10")
    7 READ(UNIT=IU,FMT="(4X,F5.2)",END=6) X
      Y=X**5/(SQRT(X))
    9 CALL SUBNAT(Y,IU,...)
      .......
      GO TO ...
    6 .......
    2 INQUIRE(FILE="NOME10",NUMBER=K)
      WRITE(.....
      WRITE(06,100) Y,K
  100 FORMAT(" ULTIMO VALORE DI Y=",F10.2," ULTIMA UNIT --> ",I2)
      STOP
      END
      SUBROUTINE SUBNAT(Y,IU,...)
      .......
      RETURN
      END
```

Un lancio eseguito utilizzando la UNIT 56, in cui la subroutine non ha modificato i valori iniziali, ha prodotto una stampa del tipo:

```
DIMMI LA UNIT DA USARE
=
56

.......
.....
...

ULTIMO VALORE DI Y=2082915.31 ULTIMA UNIT --> 56
```

Abbiamo volutamente tralasciato la codifica della subroutine e la stampa per esteso dei risultati dell'elaborazione per centrare meglio l'attenzione sull'input-output ed in particolare sul modo di funzionare dell'INQUIRE.

### 5.8 I verbi di posizionamento REWIND, BACKSPACE ed ENDFILE

I verbi di posizionamento hanno in FORTRAN 77 lo stesso significato che avevano nel precedente standard, cambia solo quanto riguarda alcune caratteristiche di formato: in particolare è ora permesso l'uso della lista di controllo. Essa non è però obbligatoria ed il modo più semplice di scrivere i tre verbi è identico nei due standard.

Prima di esaminarli in dettaglio è necessario ricordare che, ancora, essi sono rivolti solo a file trattati sequenzialmente e che si trovino su supporto magnetico.

La lista di controllo è comune ai tre verbi:

[UNIT = ]n
[IOSTAT = sta]
[ERR = l]

Il significato delle specifiche è già conosciuto poiché è identico a quello prima esaminato per gli altri verbi di input-output. Va tenuto presente che per ogni comando di posizionamento si può specificare una ed una sola delle specifiche IOSTAT od ERR.

Per posizionarsi all'inizio di un file, cioè sul primo record anche qualora siano già stati trattati vari altri record, si usa il verbo REWIND; formati permessi sono:

REWIND **n**
REWIND (**clist**)

con l'ovvio significato di **clist** (la lista di controllo) e di **n** (l'unità).
Si noti che, se al momento della REWIND fossimo già posizionati sul primo record, questo verbo non avrebbe semplicemente alcun effetto.

Per posizionarsi sul record precedente all'ultimo trattato, si usa il verbo BACKSPACE che potremo scrivere in uno dei due modi seguenti:

BACKSPACE **n**
BACKSPACE (**clist**)

con l'ovvio significato di **clist** e di **n**.
Se, al momento di tale richiesta, ci si trovasse sul primo record del file, il verbo non avrebbe alcun effetto. Se ci si trovasse posizionati sul record

fittizio, che reca, sui file sequenziali, la segnalazione di fine-file, si tornerebbe sull'ultimo record effettivo.

E' proibito l'uso del verbo su un file che non sia connesso.

Quando si desidera inserire, alla fine di un file, il record di fine-file, si usa il verbo ENDFILE. Esso ha la funzione di scrivere quel record e dunque di **terminare** il file.

I formati permessi sono:

ENDFILE **n**
ENDFILE (**clist**)

con il solito significato di **clist** ed **n**. Questo verbo *termina* il file, non lo *chiude*: ciò significa che dopo la sua azione è ancora possibile eseguire un riposizionamento all'indietro per mezzo di BACKSPACE o REWIND e continuare il trattamento.

### 5.9 I descrittori di edizione nei FORMAT

Le varie specifiche di FORMAT utilizzabili in FORTRAN IV lo sono ancora in FORTRAN 77. Si è già detto che ora esse vengono normalmente dette **descrittori di edizione** e va aggiunto che vengono classificate in due gruppi: quello dei **descrittori di edizione ripetibili** e quello dei **non ripetibili**. Per i primi sarà possibile l'uso di un fattore di ripetizione, per i secondi no. Questa suddivisione può essere ritenuta valida anche per le specifiche del precedente standard. Comunque la lista (le differenze sono evidenziate in grassetto) è la seguente:

| **ripetibile** | **non ripetibile** |
|---|---|
| In | "..." oppure '...' oppure nH... |
| **In.m** | Tn |
| Fn.m | **TLn** |
| En.h | **TRn** |
| En.h**Ee** | nX |
| Dn.h | **SP** |
| Gn.h | **SS** |
| Gn.h**Ee** | **S** |
| Ln | mP |
| An | **BN** |
| **A** | **BZ** |
| | **:** |
| | / |

dove **n** ed **e** sono costanti intere non segnate diverse da zero, **h** ed **m** sono costanti intere non segnate.

Prima di procedere all'esame delle singole differenze va ricordato che, per un output su stampa, il controllo del carrello è ottenuto ancora utilizzando come primo carattere della riga lo spazio, 0,1,+ con significato inalterato.

**In.m** è usato per variabili intere: deve essere $m \leqslant n$; in input è come scrivere In; in output significa che verranno stampati almeno m caratteri con eventuali zeri in testa; se il valore da mettere in output è zero e $m = 0$ il campo di output è posto a blank:

| Valore | FORMAT | Output |
|---|---|---|
| 16 | I6.3 | 016 |
| 161 | I3.2 | 161 |
| 161 | I4.0 | 161 |
| 0 | I4.0 | |

**...Ee** è usato nei FORMAT E e G; esso non ha alcun effetto in input, mentre in output significa che si desidera la stampa di un esponente di almeno **e** cifre.

**A** è un'estensione della specifica An utilizzabile solo in output; significa che la lunghezza del campo di output deve essere uguale a quella della variabile di caratteri relativa.

**TLn** permette di spostare il posizionamento all'indietro di n posizioni; si ponga attenzione che non si tratta di un posizionamento assoluto, ma relativo: si retrocede di n posizioni rispetto alla posizione già raggiunta.

**TRn** come il precedente ma in avanti: si tratta di n posizioni in avanti, rispetto a quella raggiunta.

**SP** non ha effetto in input; normalmente, come sappiamo, un valore numerico positivo viene stampato privo del segno + (1), se invece si desidera la stampa anche di tale carattere è sufficiente porre SP prima del *descrittore di edizione* relativo alla variabile numerica; il suo effetto permane per tutte le variabili numeriche presenti in quel FORMAT a meno che non venga specificato qualcosa in contrario con SS o S.

---

(1) ma dipende dal sistema, a volte esso viene comunque stampato.

**SS** il segno + non deve venire stampato; il suo effetto permane per tutto il FORMAT o fin quando non sia presente un SP o un S; ha senso solo in output.

**S** riporta il sistema al suo normale modo di funzionare e di solito, dunque ha lo stesso senso di SS; se nel FORMAT non è presente né un precedente SS né un SP esso è del tutto inutile; ha senso solo in output.

**BN** non ha senso in output; serve a cambiare un'eventuale specifica BLANK = "**ZERO**" data esplicitamente od implicitamente all'atto della OPEN (è dunque analogo a BLANK = "**NULL**") e significa che tutti i blank eventualmente presenti nel campo di input devono venire ignorati: ad esempio, il campo di input 12 3 viene interpretato come 123. Se tutto il campo è a blank viene interpretato come zero.

**BZ** non ha senso in output; serve a cambiare una eventuale specifica BLANK = "**NULL**" data esplicitamente o implicitamente all'atto della OPEN (è dunque analogo a BLANK = "**ZERO**") e significa che i blank devono venire sostituiti con zero, così 12 3 viene interpretato come 12030.

**:** come sappiamo il FORMAT viene esaminato dall'inizio alla fine mettendo in corrispondenza ogni variabile della lista di input-output con il suo descrittore (va tenuto conto, nello stabilire tale corrispondenza, della presenza di parentesi o di fattori ripetitivi), la lista non deve terminare prima che il FORMAT sia esaurito (si avrebbe un errore); la presenza di due punti serve ad ovviare a questa regola e significa che in quella posizione l'esame del FORMAT può essere interrotto se non esistono altre variabili nella lista; se ve ne sono, invece, i due punti vengono ignorati. L'ovvio vantaggio sta nel fatto che con i due punti è possibile utilizzare lo stesso FORMAT per un più grande numero di istruzioni di input-output.

```
      READ(UNIT=1,FMT=10) I,L,M,N
      ...
      WRITE(UNIT=6,FMT=10) I,L
      ...
      WRITE(UNIT=9,FMT=10) I,L,M
      ...
   10 FORMAT(I3,I6:I5:I5)
```

Concludiamo ricordando che le modalità secondo le quali viene eseguito l'input-output per file formattati non si esauriscono in quelle relative alla elencazione che abbiamo esaminato all'inizio di questo paragrafo. Esiste, ad esempio, la possibilità di una formattazione diretta (si veda il § 5.3 IL VERBO

READ PER I FILE ESTERNI) ed è ancora possibile l'uso della NAME-LIST che non ha subito variazioni.

### 5.10 Un'applicazione

A conclusione di questo nostro esame del FORTRAN, vogliamo esporre l'esempio di un programma completo. Esamineremo una semplice applicazione del FORTRAN 77 ad alcuni problemi di ordine matematico. Precisiamo subito che non si tratta di nulla di particolarmente complesso; né si tratta di risolvere chissà quali superiori questioni di matematica applicata. Lo scopo dell'esempio è quello di mostrare come si procede nella soluzione di un programma concreto e come appare un programma reale nella sua interezza.

Data una funzione reale del tipo $y = f(x)$ definita in un intervallo [a, b] deve essere possibile: 1) calcolare i suoi valori in una serie di punti appartenenti all'intervallo, 2) trovare i punti in cui si annulla (con una certa approssimazione), 3) calcolare l'integrale definito in [a, b] con il metodo dei trapezi, o 4) con il metodo di Simpson.

Le funzionalità sono state affidate a quattro subroutine separate: le esamineremo tra poco analizzando brevemente anche i metodi (o gli algoritmi solutivi) utilizzati. Il programma principale permette di eseguire la scelta della subroutine, cioè della funzionalità, la richiama in esecuzione ed infine ripropone la scelta. La subroutine oltre ad eseguire quanto richiesto (spesso richiamando altri sottoprogrammi), provvede all'input-output.

Per le caratteristiche stesse del problema, il programma è nato come interattivo: si desiderava qualcosa di sufficientemente veloce, che fornisse risposte immediate, soprattutto si desiderava poter ridefinire volta per volta le funzionalità da eseguire, l'intervallo di definizione, l'approssimazione etc. Ciò va tenuto presente nel leggere il programma e va tenuto presente anche che il lancio si intende eseguito per mezzo di un terminale: gli input affidati all'unità standard di sistema (come READ, I e READ, A, B nel programma principale) permetteranno di attendere la risposta dell'utente e l'esecuzione riprenderà in funzione di questa; gli output su unità standard di sistema (WRITE (06, ...)) saranno tutti inviati al terminale stesso. Qualora si desideri utilizzare questo programma per un lancio senza terminali, cioè utilizzando come unità standard il lettore di schede e la stampante, sarà sufficiente modificare opportunamente le READ e le WRITE affidando le risposte di input (previste all'inizio, una volta per tutte) ad una serie ordinata di schede e sopprimendo quelle segnalazioni di output (come: DICHIARARE L'INTERVALLO ...) che in questo caso perdono senso.

Il calcolo della funzione è affidato ad un sottoprogramma FUNCTION di nome **Y** (si tratta dell'ultimo sottoprogramma del listato che segue questi commenti: in esso è presente, come esempio, la codifica della parabola $y = x^2—4x + 3$) che viene richiamata in più occasioni dalle altre subroutine. Questa parte andrà codificata volta per volta a seconda della funzione che si desidera analizzare.

La ricerca dei valori della funzione in una serie di punti è affidata alla subroutine di nome **VFPNTI**. I punti sono elencati su di un file che la subroutine provvede a leggere dopo averne chiesto il nome; una volta eseguito un controllo sull'intervallo di definizione (i punti fuori range vengono segnalati), viene emessa una tabella dei valori.

Per quanto riguarda la ricerca dei punti di nullo (affidata alla subroutine **ZERI**) si è scelto di procedere scandendo l'intervallo di definizione per centesimi dell'intervallo stesso. Volendo aumentare la precisione è sufficiente modificare il calcolo dell'incremento **H** in modo da procedere per $10^{-3}$ o $10^{-4}$ etc. Quando il valore della funzione è approssimato a zero per meno di una certa quantità che viene chiesta come dato di input (**EPSILN**) i valori della funzione e del punto vengono inviati in output. Si sarebbe potuto andare alla ricerca di quegli intervalli in cui la funzione cambia segno, o diminuisce progressivamente in valore assoluto, per poi analizzare solo quelli riducendone sempre più l'ampiezza. Forse si sarebbe ottenuta una maggiore velocità di esecuzione ed anche, a seconda del metodo, una maggiore precisione. Tuttavia ciò sarebbe andato a scapito di quella semplicità cui ci si voleva attenere in questo esempio.

La subroutine di nome **INTGRT** permette l'integrazione con il metodo dei trapezi. Ricordiamo che con tale metodo si approssima la curva che rappresenta la funzione a dei segmenti di retta. Suddividendo l'intervallo di integrazione in n intervalli sufficientemente piccoli, si ottiene una discreta precisione. La formula, per l'intervallo di integrazione [a, b] ed incrementi di ampiezza h, è la seguente:

$$\int_a^b f(x)\,dx = \frac{h}{2}(f(a)+f(b)) + h\,(f(x_2)+f(x_3)+\ldots+f(x_{n-1}))$$

nel nostro programma **Y** è, come sempre, la funzione e **D** è l'incremento h della formula. Esso è determinato nella subroutine **ORDD** la quale, valutato l'ordine di grandezza dell'intervallo di integrazione, lo pone uguale ad una frazione dell'intervallo stesso.

Il metodo di Simpson è preferibile per le funzioni che, nei vari intervallini, sono ben approssimabili a rami di parabole. L'integrale è calcolato in **INTGRS** secondo la formula:

$$\int_a^b f(x)\,dx = \frac{h}{3}(f(a) + f(b) + 2\,\Sigma\, f(x_{2i}) + 4\,\Sigma\, f(x_{2i+1})$$

cioè: 1/3 dell'incremento per la somma del primo e dell'ultimo valore della funzione, di due volte i valori di posto pari e di quattro volte quelli di posto

dispari. Ancora, nel programma, **Y** è la funzione e **D** è l'incremento, sempre determinato dalla subroutine **ORDD**.

Il listato del programma è:

```
      PROGRAM STREAL
C ***************************************************
C  MAIN (SCELTA DELLE FUNZIONALITA', INPUT DEL RANGE
C       DI DEFINIZIONE E RICHIAMO DELLE SUBROUTINES).
C ***************************************************
      DOUBLE PRECISION A,B,Y
    1 WRITE(UNIT=06,FMT=100)
  100 FORMAT(' RISPONDERE CON :',
     1/'    1 - PER CALCOLARE IL VALORE DELLA FUNZIONE IN PUNTI DATI',
     2/'    2 -  "       "     GLI ZERI IN UN ASSEGNATO INTERVALLO',
     3/'    3 -  "       "     L'' INTEGRALE CON IL METODO DEI TRAPEZI',
     4/'    4 -  "       "     L'' INTEGRALE CON LA REGOLA DI SIMPSON',
     5/'    0 - PER TERMINARE.')
      READ,I
      IF(I.LT.0.OR.I.GT.4) THEN
         WRITE(06,FMT='(" ",I3," - RISPOSTA NON AMMESSA !")') I
         GO TO 1
      ELSE IF(I.EQ.0) THEN
         STOP
      ENDIF
      WRITE(06,101)
  101 FORMAT(" DICHIARARE L' INTERVALLO DI DEFINIZIONE NELLA FORMA   A,B")
      READ,A,B
      GO TO (5,2,3,4) I
    5 CALL VFPNTI(A,B)
         GO TO 1
    2 CALL ZERI(A,B)
         GO TO 1
    3 CALL INTGRT(A,B)
         GO TO 1
    4 CALL INTGRS(A,B)
         GO TO 1
      END



C ***************************************************
C  CALCOLO DEI VALORI DELLA FUNZIONE NEI PUNTI LETTI
C  DAL FILE DI UNIT 77 (NOME IN INPUT - MAX 40 CRT.)
C ***************************************************
      SUBROUTINE VFPNTI(A,B)
      DOUBLE PRECISION A,B,C,X,Y
      CHARACTER NOMEF *40
      WRITE(06,FMT="(' DICHIARARE IL NOME DEL FILE SU CUI SI TROVANO',
     &' I VALORI DEI PUNTI')")
      READ,NOMEF
      OPEN(UNIT=77,FILE=NOMEF)
      WRITE(06,FMT="(//11X,'PUNTO',5X,'VALORE DELLA FUNZ.'/)")
    1 READ(UNIT=77,FMT="(6X,F5.2)",END=9) X
      IF(X.LT.A.OR.X.GT.B) THEN
         WRITE(6,90) X
      ELSE
         C=Y(X)
         WRITE(6,91) X,C
      ENDIF
      GO TO 1
```

```
   90 FORMAT(11X,F5.2,"---> FUORI RANGE")
   91 FORMAT(11X,F5.2,8X,F8.2)
    9 WRITE(06,FMT="(' ')")
      CLOSE(77)
      RETURN
      END
C ************************************************
C  CALCOLO APPROSSIMATO DEGLI ZERI DI UNA FUNZIONE
C  NELL' INTERVALLO A,B . L'APPROSSIMAZIONE E' RI-
C  CHIESTA COME DATO DI INPUT (EPSILN). LA SCANSIO-
C  DELL' INTERVALLO  PROCEDE PER  CENTESIMI  DELLO
C  INTERVALLO STESSO.
C ************************************************
      SUBROUTINE ZERI(A,B)
      IMPLICIT DOUBLE PRECISION (A-C,H,Y)
      CHARACTER ZEROUT *10
      ZEROUT="ZERI      "
      WRITE(06,FMT="(' DICHIARARE IL GRADO DI APPROSSIMAZIONE')")
      READ,EPSILN
      WRITE(06,100)
  100 FORMAT(//11X,"FUNZIONE",5X,"PUNTO"/)
      H=0
    1 C1=A+H
      IF(C1.GT.B) GO TO 9
      IF(DABS(Y(C1)).LE.EPSILN) THEN
         WRITE(06,FMT="(11X,D8.2,3X,F10.3)") Y(C1),C1
         ZEROUT="ALTRI ZERI"
      END IF
      H=H+(B-A)/100
      GO TO 1
    9 WRITE(06,FMT="(11X,'NON CI SONO ',A//)") ZEROUT
      RETURN
      END
C ************************************************
C  CALCOLO DELL' INTEGRALE CON IL METODO DEI TRAPEZI.
C ************************************************
      SUBROUTINE INTGRT(A,B)
      IMPLICIT DOUBLE PRECISION (A-D,Y)
      D=0
      CALL ORDD(A,B,D)
      DINTG=0
      DO 20 C=A+D,B-D,D
      C1=C
      DINTG=DINTG+Y(C1)


   20 CONTINUE
      DINTG=(D/2)*(Y(A)+Y(B))+D*DINTG
      WRITE(06,50) DINTG
   50 FORMAT(" METODO DEI TRAPEZI - INTEGRALE =",F18.3//)
      RETURN
      END
C ************************************************
C  CALCOLO DELL' INTEGRALE CON LA REGOLA DI SIMPSON.
C ************************************************
      SUBROUTINE INTGRS(A,B)
      IMPLICIT DOUBLE PRECISION (A-D,Y)
      D=0
      CALL ORDD(A,B,D)
      DT1=0
      DT2=0
      I=1
      DO 20 C=A+D,B-D,D
      C1=C
      I=I+1
      IF(MOD(I,2).EQ.0) THEN
         DT1=DT1+Y(C1)
      ELSE
```

```
         DT2=DT2+Y(C1)
      ENDIF
   20 CONTINUE
      DINTG=(D/3)*(Y(A)+Y(B)+2*DT1+4*DT2)
      WRITE(06,50) DINTG
   50 FORMAT(" REGOLA DI SIMPSON - INTEGRALE =",F18.3//)
      RETURN
      END
C *************************************************
      SUBROUTINE ORDD(A,B,D)
      DOUBLE PRECISION A,B,D
      I=10
      DO 1 K=1,6
      IF((B-A).LE.I) THEN
         D=(B-A)/(I*10)
         GO TO 2
      ELSE
         I=10*I
      END IF
      IF(I.GT.1000000) THEN
         WRITE(06,FMT="(' INTERVALLO > 1.000.000 NON PERMESSO')")
         STOP
      END IF
    1 CONTINUE
    2 RETURN
      END
C *************************************************
C  QUESTE ISTRUZIONI RIGUARDANO LA FUNZIONE (VANNO
C  CODIFICATE VOLTA PER VOLTA A SECONDA DELLE ESI-
C  GENZE) - ESEMPIO: PARABOLA    Y=X**2-4*X+3    .
C *************************************************
      DOUBLE PRECISION FUNCTION Y(X)
      DOUBLE PRECISION X
      Y=X**2-4*X+3
      RETURN
      END
```

Vediamo ora, nel caso della parabola codificata sopra, un esempio di esecuzione. Il file su cui sono scritti i punti ha nome FLPNTI ed è il seguente:

```
77777 -2.00
77778 -1.05
77779 -1.00
77780 -0.05
77781  0.00
77782  0.20
77783 01.00
77784 01.50
77785 02.00
77786 02.85
77787 03.00
77788 03.14
77789 03.15
77790 06.28
77799 12.56
```

i primi sei caratteri vengono tralasciati in lettura e, dunque, si tratta di 15 valori compresi tra —2 e 12,56. Rispondendo con 1 per la scelta della funzionalità, si ha:

```
RISPONDERE CON :
  1 - PER CALCOLARE IL VALORE DELLA FUNZIONE IN PUNTI DATI
  2 -  "       "     GLI ZERI IN UN ASSEGNATO INTERVALLO
  3 -  "       '     L' INTEGRALE CON IL METODO DEI TRAPEZI
  4 -  "       "     L' INTEGRALE CON LA REGOLA DI SIMPSON
  0 - PER TERMINARE.
=1
DICHIARARE L' INTERVALLO DI DEFINIZIONE NELLA FORMA   A,B
=-2,5
DICHIARARE IL NOME DEL FILE SU CUI SI TROVANO I VALORI DEI PUNTI
=FLPNTI


          PUNTO      VALORE DELLA FUNZ.

          -2.00            15.00
          -1.05             8.30
          -1.00             8.00
          -0.05             3.20
           0.               3.00
           0.20             2.24
           1.00             0.
           1.50            -0.75
           2.00            -1.00
           2.85            -0.28
           3.00             0.
           3.14             0.30
           3.15             0.32
           6.28---> FUORI RANGE
          12.56---> FUORI RANGE
```

proseguendo, per quanto riguarda la ricerca degli zeri:

```
RISPONDERE CON :
  1 - PER CALCOLARE IL VALORE DELLA FUNZIONE IN PUNTI DATI
  2 -  "       "     GLI ZERI IN UN ASSEGNATO INTERVALLO
  3 -  "       "     L' INTEGRALE CON IL METODO DEI TRAPEZI
  4 -  "       "     L' INTEGRALE CON LA REGOLA DI SIMPSON
  0 - PER TERMINARE.
=2
DICHIARARE L' INTERVALLO DI DEFINIZIONE NELLA FORMA   A,B
=0,5
DICHIARARE IL GRADO DI APPROSSIMAZIONE
=0.001


          FUNZIONE      PUNTO

          0.87D-18         1.000
          0.43D-17         3.000
          NON CI SONO ALTRI ZERI
```

e gli integrali:

```
RISPONDERE CON :
  1 - PER CALCOLARE IL VALORE DELLA FUNZIONE IN PUNTI DATI
  2 -  "       "     GLI ZERI IN UN ASSEGNATO INTERVALLO
  3 -  "       "     L' INTEGRALE CON IL METODO DEI TRAPEZI
  4 -  "       "     L' INTEGRALE CON LA REGOLA DI SIMPSON
  0 - PER TERMINARE.
=3
DICHIARARE L' INTERVALLO DI DEFINIZIONE NELLA FORMA   A,B
=1,3
METODO DEI TRAPEZI - INTEGRALE =            -1.333


RISPONDERE CON :
  1 - PER CALCOLARE IL VALORE DELLA FUNZIONE IN PUNTI DATI
  2 -  "       "     GLI ZERI IN UN ASSEGNATO INTERVALLO
  3 -  "       "     L' INTEGRALE CON IL METODO DEI TRAPEZI
  4 -  "       "     L' INTEGRALE CON LA REGOLA DI SIMPSON
  0 - PER TERMINARE.
=4
DICHIARARE L' INTERVALLO DI DEFINIZIONE NELLA FORMA   A,B
=1,3
REGOLA DI SIMPSON - INTEGRALE =             -1.333


RISPONDERE CON :
  1 - PER CALCOLARE IL VALORE DELLA FUNZIONE IN PUNTI DATI
  2 -  "       "     GLI ZERI IN UN ASSEGNATO INTERVALLO
  3 -  "       "     L' INTEGRALE CON IL METODO DEI TRAPEZI
  4 -  "       "     L' INTEGRALE CON LA REGOLA DI SIMPSON
  0 - PER TERMINARE.
=0
```

In un altro caso si è voluto eseguire un'analisi della cubica $y = 10x^3 - 10x$. Prima di tutto è stato necessario ricodificare la funzione:

```
C ******************************************************
C  QUESTE ISTRUZIONI RIGUARDANO LA FUNZIONE (VANNO
C  CODIFICATE VOLTA PER VOLTA A SECONDA DELLE ESI-
C  GENZE) - ESEMPIO: CUBICA    Y=10*(X**3-X)
C ******************************************************
      DOUBLE PRECISION FUNCTION Y(X)
      DOUBLE PRECISION X
      IF(ABS(X).LE.0.00001) THEN
         Y=0
      ELSE
         Y=10*(X**3-X)
      ENDIF
      RETURN
      END
```

poi si è deciso di rendere maggiormente raffinata la scansione dell'intervallo per una più precisa ricerca degli zeri, ciò è stato ottenuto modificando, nella subroutine **ZERI**, la definizione di H in questo modo:

```
H=H+(B-A)/100000
```

a questo punto si è mandato in esecuzione il programma. Si noti che si è ancora usato il file **FLPNTI** per il valore dei punti della prima funzionalità. L'effetto del lancio, visto nella sua interezza, è il seguente:

```
RISPONDERE CON :
  1 - PER CALCOLARE IL VALORE DELLA FUNZIONE IN PUNTI DATI
  2 -  "       "     GLI ZERI IN UN ASSEGNATO INTERVALLO
  3 -  "       "     L' INTEGRALE CON IL METODO DEI TRAPEZI
  4 -  "       "     L' INTEGRALE CON LA REGOLA DI SIMPSON
  0 - PER TERMINARE.
=1
DICHIARARE L' INTERVALLO DI DEFINIZIONE NELLA FORMA   A,B
=-1.5,1.5
DICHIARARE IL NOME DEL FILE SU CUI SI TROVANO I VALORI DEI PUNTI
=FLPNTI

              PUNTO      VALORE DELLA FUNZ.

              -2.00---> FUORI RANGE
              -1.05          -1.08
              -1.00           0.
              -0.05           0.50
               0.             0.
               0.20          -1.92
               1.00           0.
               1.50          18.75
               2.00---> FUORI RANGE
               2.85---> FUORI RANGE
               3.00---> FUORI RANGE
               3.14---> FUORI RANGE
               3.15---> FUORI RANGE
               6.28---> FUORI RANGE
              12.56---> FUORI RANGE



  RISPONDERE CON :
    1 - PER CALCOLARE IL VALORE DELLA FUNZIONE IN PUNTI DATI
    2 -  "       "     GLI ZERI IN UN ASSEGNATO INTERVALLO
    3 -  "       "     L' INTEGRALE CON IL METODO DEI TRAPEZI
    4 -  "       "     L' INTEGRALE CON LA REGOLA DI SIMPSON
    0 - PER TERMINARE.
  =2
  DICHIARARE L' INTERVALLO DI DEFINIZIONE NELLA FORMA   A,B
  =-2,2
  DICHIARARE IL GRADO DI APPROSSIMAZIONE
  =0.00001
```

```
          FUNZIONE      PUNTO

          -.16D-14       -1.000
          0.              0.000
          0.10D-12        1.000
          NON CI SONO ALTRI ZERI



RISPONDERE CON :
  1 - PER CALCOLARE IL VALORE DELLA FUNZIONE IN PUNTI DATI
  2 -  "       "     GLI ZERI IN UN ASSEGNATO INTERVALLO
  3 -  "       "     L' INTEGRALE CON IL METODO DEI TRAPEZI
  4 -  "       "     L' INTEGRALE CON LA REGOLA DI SIMPSON
  0 - PER TERMINARE.
=3
DICHIARARE L' INTERVALLO DI DEFINIZIONE NELLA FORMA   A,B
=0,1
METODO DEI TRAPEZI - INTEGRALE =            -2.500

RISPONDERE CON :
  1 - PER CALCOLARE IL VALORE DELLA FUNZIONE IN PUNTI DATI
  2 -  "       "     GLI ZERI IN UN ASSEGNATO INTERVALLO
  3 -  "       "     L' INTEGRALE CON IL METODO DEI TRAPEZI
  4 -  "       "     L' INTEGRALE CON LA REGOLA DI SIMPSON
  0 - PER TERMINARE.
=4
DICHIARARE L' INTERVALLO DI DEFINIZIONE NELLA FORMA   A,B
=0,1
REGOLA DI SIMPSON - INTEGRALE =             -2.500

RISPONDERE CON :
  1 - PER CALCOLARE IL VALORE DELLA FUNZIONE IN PUNTI DATI
  2 -  "       "     GLI ZERI IN UN ASSEGNATO INTERVALLO
  3 -  "       "     L' INTEGRALE CON IL METODO DEI TRAPEZI
  4 -  "       "     L' INTEGRALE CON LA REGOLA DI SIMPSON
  0 - PER TERMINARE.
=5
  5 - RISPOSTA NON AMMESSA !
RISPONDERE CON :
  1 - PER CALCOLARE IL VALORE DELLA FUNZIONE IN PUNTI DATI
  2 -  "       "     GLI ZERI IN UN ASSEGNATO INTERVALLO
  3 -  "       "     L' INTEGRALE CON IL METODO DEI TRAPEZI
  4 -  "       "     L' INTEGRALE CON LA REGOLA DI SIMPSON
  0 - PER TERMINARE.
=0
```

Si osservi che è stata volutamente data la risposta errata 5 alla scelta di funzionalità per verificare il corretto funzionamento del controllo. Alla risposta 0, il programma termina normalmente.

Il programma porebbe venir facilmente migliorato ottimizzando i tempi di esecuzione e migliorando la precisione, tuttavia, come si è già detto, evidenti motivi di semplicità e chiarezza ci hanno fatto preferire questa esposizione.

Comunque va ancora notato che esso è facilmente modificabile e che l'aggiunta di nuove routine per altre funzionalità richiede interventi assai limitati come l'inserimento di poche nuove istruzioni nel programma principale.

APPENDICE A

# LA PROGRAMMAZIONE STRUTTURATA

La Programmazione Strutturata nasce in seguito ad un'esigenza di semplificazione e di standardizzazione. Programmare secondo metodi non strutturati significa che la logica del programma, gli algoritmi solutivi e, più in generale, la struttura del programma stesso sono lasciati all'inventiva od all'esperienza del programmatore.

Che possano esistere programmi scritti malamente e programmi, invece, agili e veloci è scontato come è altrettanto scontato che ciò dipende dalle scelte che un programmatore più o meno abile può fare.

Nel caso della programmazione tecnico-scientifica, spesso l'alta complessità logica ed algoritmica conduce a situazioni irrisolvibili quando non si disponga di un poco di creatività e ciò al punto che può risultare inevitabile affidare ogni scelta all'inventiva, appunto, del programmatore. Ciò accade praticamente ogni volta che la parte algoritmica è preponderante rispetto all'input-output: il programma ha input di dimensioni assai limitate (o non ha addirittura input), o produce output altrettanto limitati (magari un unico risultato numerico). Nel caso invece in cui l'elaborazione algoritmica non costituisca la parte preponderante e cioè quando si abbiano input ed output di una certa mole, disporre di regole fisse che determinino la stesura del programma comporta notevoli vantaggi. Prima di farne un appropriato esame, aggiungiamo subito che questo è il caso di tutti i programmi gestionali e che, per questi, procedere ad una strutturazione si rivela sicuramente vantaggioso.

Lo stabilire una serie di regole fisse significa definire una metodologia completa che permetta di codificare il programma in modo automatico. L'ottenere la struttura del programma in modo diretto, applicando rigorosamente delle regole metodologiche alla descrizione delle specifiche del problema ed alla descrizione della struttura dei dati di input-output, significa poter scrivere dei programmi standardizzati. Così due programmatori diversi dovranno essere condotti a produrre programmi strutturalmente identici, dunque facilmente rileggibili da chiunque altro conosca la metodologia, facilmente modificabili e facilmente manutenibili (non si dimentichi che oggi uno dei costi più alti è proprio rappresentato dalla manutenzione).

Ovviamente una strutturazione così normalizzata potrà impedire che i programmi risultino ottimizzati: eventuali soluzioni agili ed eleganti, dovute all'inventiva del buon programmatore, saranno inattuabili e la logica solutiva, a volte, potrà sembrare addirittura più complessa del necessario.

L'uniformità e l'automaticità di produzione dei programmi, ottenute per mezzo della strutturazione, non possono che andare a scapito della dinamicità e della brevità.

Uno dei principi basilari nell'applicazione delle metologie di programmazione strutturata è quello stesso descritto nel testo e che prende il nome di "top-down". Il metodo, cioè, di vedere il problema prima sinteticamente, nella sua intierezza, per poi specificare i singoli dettagli via via in passi successivi.

Con un parallelo ormai noto potremmo dire: nel progettare un palazzo prima se ne disegnerà la struttura generale, poi i muri e gli altri elementi che ne concorrono alla composizione e solo all'ultimo momento dovrà essere presa una decisione circa le piastrelle o gli infissi.

Se in alcuni casi la Programmazione Strutturata può rivelarsi inefficace, questo principio va invece sicuramente ritenuto vantaggioso in ogni possibile situazione. Ciò è vero per la semplicità e la potenza del metodo che propone: esso potrà essere osservato in forza di un'impostazione mentale assai naturale e tecnicamente applicato tendendo conto del già nominato teorema di Jacopini-Böhm. Con le tre sole possibilità di *sequenza, alternativa* e *iterazione* saremo in grado di risolvere qualunque problema vedendolo prima sinteticamente e poi via via nei dettagli.

Per tornare all'argomento "Programmazione Strutturata", va osservato che è proprio a partire da queste posizioni che sono nate e sono andate evolvendosi alcune metodologie.

Già nel 1968 usciva su Communications of ACM un articolo di E. W.Dijkstra, "GO TO Statement Considered Harmful": nel rilevare come l'uso dello statement GO TO potesse risultare dannoso perché probabile causa di complicazione (e dunque successiva illeggibilità del programma) esso sanciva la nascita dell strutturazione.

In ordine temporale le metodologie di programmazione strutturata che hanno visto la luce negli anni 70 sono: Warnier, Jackson e PHOS (1). Non ne

---

(1) La metodologia Warnier nasce nel 1970 con la programmazione di "Lois pour la construction des programmes", la Jackson nel 1975 con la pubblicazione di "Principles of program design", la metodologia PHOS (Program Hierarchy from Output Structure) nasce nel 1976 da una collaborazione tra l'Istituto di Cibernetica dell'Università di Milano e la Honeywell I.S.I.

affronteremo un'analisi molto dettagliata, né faremo dei veri e propri raffronti; vedremo invece, in breve, quali siano le caratteristiche generali comuni e come si debba procedere nell'applicazione e, infine, in quali punti consistano le differenze più significative.

Le tre metodologie hanno in comune questo modo di procedere:

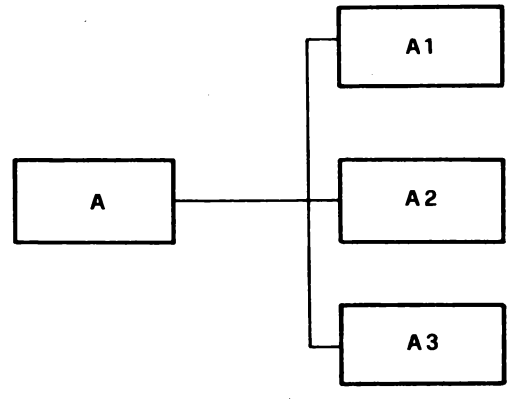
1° passo) a partire dalle specifiche del problema (o dall'analisi) si analizzano le strutture delle informazioni di input e di output e se ne disegnano le strutture secondo criteri e simbologie ben definite;

2° passo) si confrontano le strutture di input e di output verificando se tra di esse esiste compatibilità; in caso di incompatibilità si procede ad una serie di modifiche (ancora secondo certe rigorose regole); proprio in quest'ultimo punto sta il segno di una maggiore o minore efficacia della metodologia: nel prevedere una più o meno completa serie di situazioni di conflitto e nel fornire regole sufficientemente semplici per la loro soluzione;

3° passo) si disegna la struttura del programma a partire dal raffronto dell'input e dell'output.

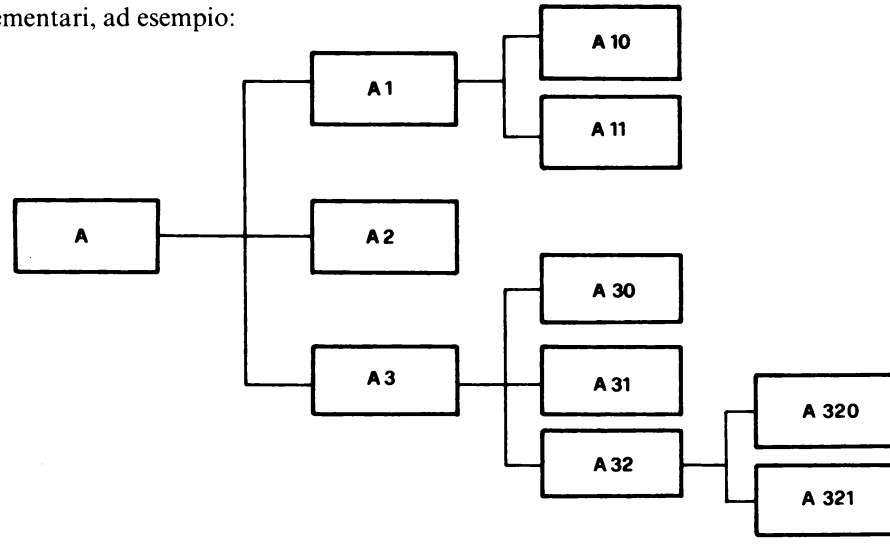
Prima di esaminare le differenze generali più significative, può valer la pena analizzare più da vicino una delle metodologie per capire meglio cosa significhi procedere per i passi ora visti. Per questo scopo abbiamo scelto la metodologia PHOS sia per la sua semplicità che per la sua completezza.

Per quanto riguarda la simbologia si tenga presente che una sequenza (di operazioni o di informazioni indifferentemente) va sempre vista come un *gruppo*, così se un certo trattamento A (il gruppo di operazioni) è composto dai trattamenti A1, A2, A3 si disegnerà la struttura:



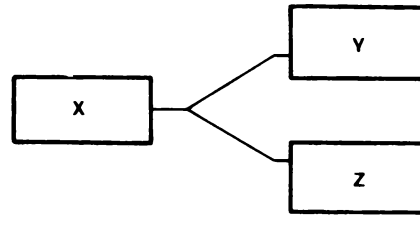

che corrisponderà alla *sequenza*. Ognuno dei moduli A1, A2, A3, potrà essere a sua volta scomposto in altri trattamenti fino a giungere alle operazioni

elementari, ad esempio:



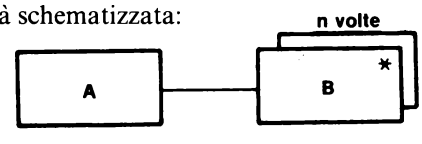

significa che le operazioni elementari sono quelle di nome A10, A11, A2, A30, A31, A320, A321.

L'*alternativa* verrà schematizzata:



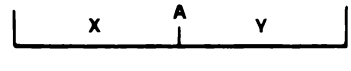

e ciò significa che il trattamento X è composto o dal trattamento Y o dal trattamento Z (questi, a loro volta potranno essere composti da altri trattamenti).

L'*iterazione* verrà schematizzata:




e ciò significherà che il trattamento A è composto da un'iterazione (n volte) del trattamento B.

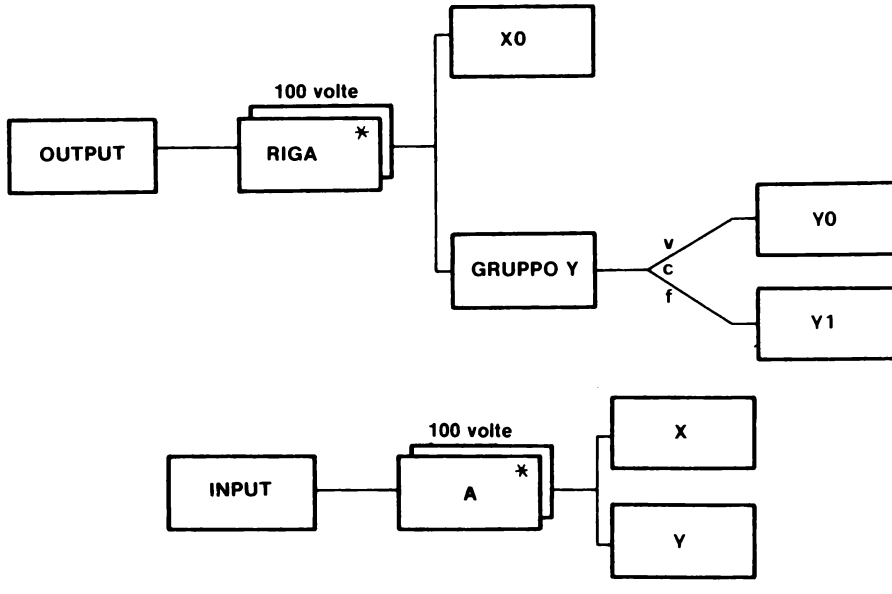
Ammettiamo ora di dover scrivere un programma il cui input consista di 100 record A suddivisi in due campi X, Y con contenuto numerico:



ammettiamo di dover leggere tali informazioni e di dover calcolare i valori: X0 a partire da X, Y0 o Y1 a partire da Y a seconda che una certa condizione C sia rispettivamente vera o falsa e, infine, di dover stampare le 100 righe:

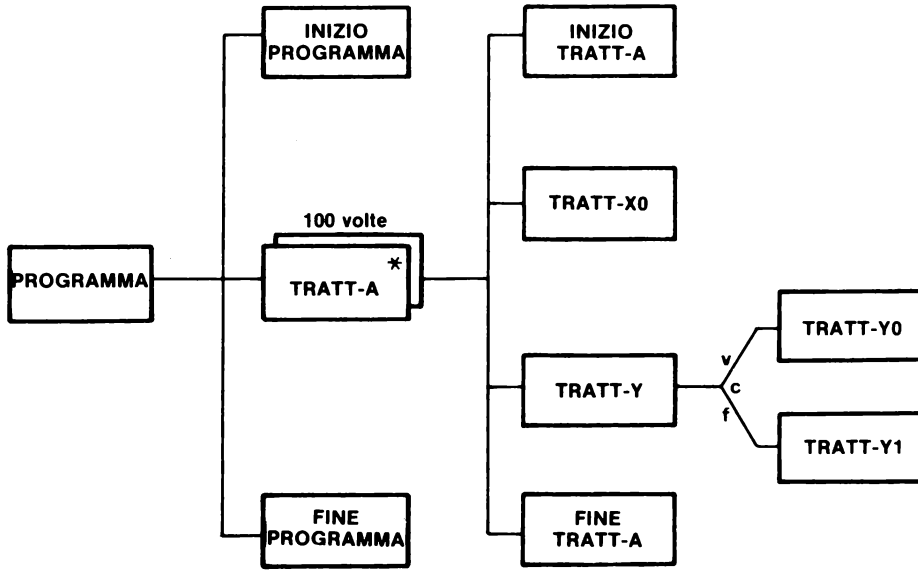
| X0 | Y0 o Y1 |
|---|---|
| xxxxxx | xxxxxxx |
| xxxxxx | xxxxxxx |
| xxxxxx | xxxxxxx |
| ... | ... |

Eseguiamo il primo passo, cioè il disegno delle strutture dell'input e dell'output:




Il confronto tra input ed output non presenta conflitti (volutamente tralasciamo l'esame di questi casi) e quindi si può procedere al disegno della struttura del programma. La metodologia prevede che, partendo dall'output, si disegnino i moduli di trattamento corrispondenti e che a quell'output condurranno. In pratica si tratta di sostituire a RIGA il trattamento per ottenere la riga di stampa, a X0 il trattamento per ottenere quel valore (è per questo che, nei moduli che seguono, i nomi saranno del tipo TRATT ...) e così di seguito. Un altro importante accorgimento sarà quello di inserire per ognuno dei trattamenti non elementari un inizio ed una fine del trattamento stesso nei quali dovranno venir inserite, successivamente in fase di codifica, le eventuali istruzioni di inizializzazione o di riazzeramento di totalizzatori, di aree di deposito e via dicendo. Va anche osservato che tali moduli devono essere presenti per un rispetto formale della metodologia, ma, ed accade spesso, possono essere vuoti.

Tenendo presente tutto ciò, la struttura del programma sarà:




nel nostro caso l'inizio e la fine sia del programma che del trattamento sono, ad eccezione di FINE TRATT-A, dei moduli vuoti.

A questo punto resta un'ultima cosa da fare: inserire le letture e scritture. Solitamente esse vanno poste nei moduli di inizio e fine, ma esiste una casistica particolareggiata ed una serie di regole precise per la loro allocazione nei vari moduli. Nel nostro caso, la lettura e la scrittura potranno venir poste rispettivamente in INIZIO TRATT-A e FINE TRATT-A.

La metodologia vorrebbe che la successiva codifica procedesse seguendo i vari livelli gerarchici: il primo livello (INIZIO PROG, TRATT-A, FINE PROG) costituirebbe il main, gli altri livelli non rappresenterebbero altro che i sottoprogrammi richiamati volta per volta dai livelli superiori. Il FORTRAN si presta abbastanza poco a questo metodo di codifica (1): nel nostro caso infatti, a rigore, avremmo dovuto codificare come mostrato nel primo dei due modi qui di seguito riportati, ovvii i motivi di semplicità, però, fanno preferire il secondo:

(1) Si prestano bene, invece, il Cobol con l'istruzione PERFORM ed il Pascal con i suoi semplici richiami a PROCEDURE.

**strutt.**

```
      PROGRAM PROG
      COMMON ...
      CALL INPROG
      DO 1 I=1,100
      CALL TRATTA
    1 CONTINUE
      CALL FINPRG
      STOP
      END
      SUBROUTINE INPROG
      RETURN
      END
      SUBROUTINE TRATTA
      COMMON ...
      CALL INTRTA
      CALL TRATTX
      CALL TRATTY
      CALL FINTRA
      RETURN
      END
      SUBROUTINE FINPRG
      RETURN
      END
      SUBROUTINE INTRTA
      COMMON ...
      READ (...) X,Y
      ...
      RETURN
      END
      SUBROUTINE TRATTX
      COMMON ...
      X0 = ...
      ...
      RETURN
      END
      SUBROUTINE TRATTY
      COMMON ...
      IF(c) THEN
         Y0= ...
      ELSE
         Y1= ...
      ENDIF
      RETURN
      END
      SUBROUTINE FINTRA
```

**non strutt.**

```
      PROGRAM PROG
      COMMON
      ...
      DO 1 I=1,100
      READ (...) X,Y
      CALL TRATTA
      WRITE ...
    1 CONTINUE
      STOP
      END
      SUBROUTINE TRATTA
      COMMON ...
      X0= ...
      IF (c) THEN
         Y0= ...
      ELSE
         Y1= ...
      ENDIF
      RETURN
      END
```

```
COMMON ...
WRITE ...
RETURN
END
```

Non ci sembra di dover approfondire le ragioni per le quali si è detta preferibile la seconda codifica (forse va anche aggiunto che in un caso semplice come quello dell'esempio si sarebbe potuto portare ogni cosa a livello del main), tuttavia abbiamo voluto codificare anche nel modo rigorosamente voluto dalla metodologia, perchè fosse possibile capire quale ne sia la logica e per evidenziare come, avendo ottenuto dei moduli indipendenti di programma, sia ora più facile intervenire singolarmente su ognuno di essi per eventuali modifiche.

Desideriamo ribadire, comunque, che il motivo per cui non si è scelto, nel testo, di adottare la metodologia strutturata non sta nella scarsa adattabilità al linguaggio, ma piuttosto nel fatto che, nei programmi con preponderanza algoritmica, l'input-output non è quasi mai tale da permetterne una strutturazione ed un disegno utilizzabili nei passi successivi.

Per concludere, ci pare valga la pena esaminare i principi generali delle altre metodologie. Si è visto che la prima a nascere fu la Warnier. Essa non va ritenuta ormai sorpassata dalle successive: il rigore con il quale ne sono state definite le regole di applicazione, gli assiomi base ed i suoi presupposti teorici ne fanno sicuramente una delle più precise e consequenziali.

Essa prevede che le informazioni di input-output siano viste come insiemi. La strutturazione di tali insiemi procede per scomposizioni in sottoinsiemi fino alla determinazione dei componenti elementari. Questo procedimento, tradotto in una serie di tabelle gerarchiche, condurrà alla strutturazione dello schema di programma.

L'unica notazione negativa può riguardare il fatto che nei passaggi di strutturazione intervengono elementi matematici e logici non sempre di semplice applicazione.

Infine, per quanto riguarda la Jackson va detto che essa non è, a stretto rigore di termini, una vera e propria metodologia, bensì un insieme di regole di carattere pratico che compongono un metodo facilmente utilizzabile, ma non altrettanto rigoroso e completo: al programmatore resta un certo grado di libertà e certe soluzioni sono devolute alla sua abilità.

Ancora le informazioni di input-output sono descritte e schematizzate (questa volta singolarmente file per file) e confrontate conducendo poi al programma. Grande vantaggio del metodo è la sua semplicità e praticità.

APPENDICE B

# ORDINE DELLE ISTRUZIONI DEL FORTRAN IV

| | | | | |
|---|---|---|---|---|
| | Dichiarazioni delle unità di programma: | | | |
| | BLOCK DATA<br>SUBROUTINE | | | FUNCTION |
| | Eunciati di specificazione: | | | |
| | | | INTEGER<br>REAL<br>DOUBLE PRECISION<br>COMPLEX<br>LOGICAL | COMMON<br>IMPLICIT<br>EXTERNAL<br>DIMENSION |
| Commenti | | | EQUIVALENCE | |
| | ENTRY | | Statement Function | |
| | | | Istruzioni eseguibili: | |
| | | | Enunciati di assegnazione: | logici,<br>aritmetici |
| | FORMAT | | | |
| | | DATA | IF aritmetico<br>IF logico<br>GO TO incondizionato<br>GO TO calcolato<br>GO TO assegnato | WRITE<br>REWIND<br>BACKSPACE<br>ENDFILE<br>DEFINE FILE |
| | NAMELIST | | ASSIGN<br>DO<br>CONTINUE | FIND<br>PAUSE<br>STOP<br>CALL |
| | | | READ | RETURN |
| | END | | | |

Le righe verticali separano quelle istruzioni che possono essere liberamente mischiate: per esempio un commento può essere messo ovunque nel programma prima dell'istruzione END, mentre un'istruzione FORMAT può comparire ovunque, purché dopo una eventuale istruzione di dichiarazione di una unità di programma e prima della END.

Le righe orizzontali separano le istruzioni che devono essere presenti nell'ordine indicato. Così gli Statement Functions devono seguire gli eventuali enunciati di specificazione e devono precedere le istruzioni eseguibili.

Le linee orizzontali tratteggiate indicano che l'EQUIVALENCE deve seguire qualunque enunciato di specificazione relativo alle variabili che devono essere dichiarate equivalenti; per quanto riguarda il DATA, esso deve seguire ogni enunciato di specificazione che faccia riferimento alle variabili che da esso devono essere inizializzate.

N.B. Tutti i vari enunciati che non rientrano nella categoria *istruzioni eseguibili* sono da considerarsi: *Istruzioni non eseguibili.*

APPENDICE C

# TAVOLE DELLE DIFFERENZE TRA FORTRAN IV E FORTRAN 77

Nelle tabelle che seguono sono elencati gli statement presenti rispettivamente nei due standard.

Quando uno statement è presente in entrambe le colonne senza alcun richiamo, significa che non è cambiato né il senso, né il formato di scrittura.

I richiami permettono di leggere alcuni commenti sulle differenze: essi sono presenti quando ci sono estensioni, o quando lo statement è completamente nuovo. Il commento porta anche l'indicazione della pagina del testo dove l'argomento è stato discusso e si riferisce, comunque, alle nuove possibilità.

| | **DICHIARAZIONI** | |
|---|---|---|
| **FORTRAN IV** | **FORTRAN 77** | **richiami** |
| INTEGER | INTEGER | |
| REAL | REAL | |
| DOUBLE PRECISION | DOUBLE PRECISION | |
| COMPLEX | COMPLEX | (1) |
| LOGICAL | LOGICAL | |
| — | CHARACTER | (2) |
| — | PARAMETER | (3) |
| DIMENSION | DIMENSION | (4) |
| COMMON | COMMON | |
| EQUIVALENCE | EQUIVALENCE | |
| EXTERNAL | EXTERNAL | (5) |
| — | INTRINSIC | (5) |
| — | SAVE | (6) |
| DATA | DATA | |

(1) Questo formato ora accetta anche valori interi (pag. 205)
(2) Completamente nuova in F77 (pag. 199)
(3) Completamente nuova in F77 (pag. 205)
(4) Estensione: è possibile definire il campo di variabilità degli indici per qualunque valore intero (pag. 207)
(5) L'uso di EXTERNAL diviene limitato ai sottoprogrammi esterni, per le funzioni intrinseche è obbligatorio l'uso di INTRINSIC (pag. 209)
(6) Completamente nuova in F77 (pag. 210)

| DEFINIZIONI DI PROGRAMMI E SOTTOPROGRAMMI | | |
|---|---|---|
| **FORTRAN IV** | **FORTRAN 77** | **richiami** |
| —<br>BLOCK DATA<br>FUNCTION<br>SUBROUTINE<br>ENTRY | PROGRAM<br>BLOCK DATA<br>FUNCTION<br>SOBROUTINE<br>ENTRY | (1) |
| (1) Completamente nuova in F77 (pag. 212) | | |

| ISTRUZIONI DI CONTROLLO | | |
|---|---|---|
| **FORTRAN IV** | **FORTRAN 77** | **richiami** |
| ASSIGN | ASSIGN | (1) |
| GO TO | GO TO | (2) |
| IF | IF | |
| — | IF ... THEN ... ELSE | (3) |
| — | ENDIF | (3) |
| DO | DO | (4) |
| CONTINUE | CONTINUE | |
| CALL | CALL | |
| RETURN | RETURN | |
| PAUSE | PAUSE | |
| STOP | STOP | |
| END | END | |
| (1) La variabile di assegnazione può essere riutilizzata (pag. 95)<br>(2) Nel GO TO calcolato, l'elaborazione procede sull'istruzione successiva se il valore di rimando non è corretto (pag. 215)<br>(3) Completamente nuova in F77 (pag. 217)<br>(4) Alcune differenze sul controllo d'uscita e sull'indice di ciclo (pag. 216) | | |

| ISTRUZIONI RELATIVE ALL'INPUT OUTPUT | | |
|---|---|---|
| **FORTRAN IV** | **FORTRAN 77** | **richiami** |
| READ | READ | (1) (3) |
| WRITE | WRITE | (2) (3) |
| — | PRINT | (4) |
| — | OPEN | (5) |
| — | CLOSE | (6) |
| — | INQUIRE | (7) |
| FIND | — | |
| REWIND | REWIND | (8) |
| BACKSPACE | BACKSPACE | (9) |
| ENDFILE | ENDFILE | (10) |
| FORMAT | FORMAT | (11) |
| NAMELIST | NAMELIST | |

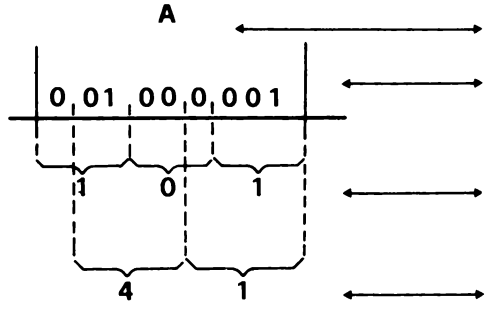
(1) Notevoli estensioni (pag. 229)
(2) Notevoli estensioni (pag. 232)
(3) Estensione all'uso di files interni pag. (236)
(4) Completamente nuova in F77 (pag. 232)
(5) Completamente nuova in F77 (pag. 239)
(6) Completamente nuova in F77 (pag. 239)
(7) Completamente nuova in F77 )pag. 247)
(8) Alcune estensioni (pag. 253)
(9) Alcune estensioni (pag. 253)
(10) Alcune estensioni (pag. 253)
(11) Alcune estensioni (pag. 253)

APPENDICE D

# I CODICI DI RAPPRESENTAZIONE DEI CARATTERI

Nel testo si è fatto riferimento, alcune volte, ai codici ASCII, EBCDIC, al codice di perforazione Hollerith e via dicendo. Si tratta o del modo in cui i caratteri sono immagazzinati in memoria o del modo in cui sono perforati su scheda. Qui di seguito riportiamo una tabella dei codici più diffusi. Per la leggibilità di tale tabella sono forse necessarie alcune spiegazioni: ad esempio si cerchi il corrispondente della lettera A nella colonna del codice ASCII.

Si sarà trovato che la colonna intestata con la cifra 8 (o, come diremo, la colonna dell'*ottale*) reca scritto 101. Questo sta a significare che ognuna di queste tre cifre esprime un numero di base 8, corrispondente a tre bit di memoria secondo la traduzione in binario di ognuna di esse. Forse tale concetto potrà risultare più semplice osservando lo schema che segue. Esso rappresenta un byte di 9 bit nel quale la lettera A è memorizzata secondo il formato a caratteri:




il byte contiene il carattere A;

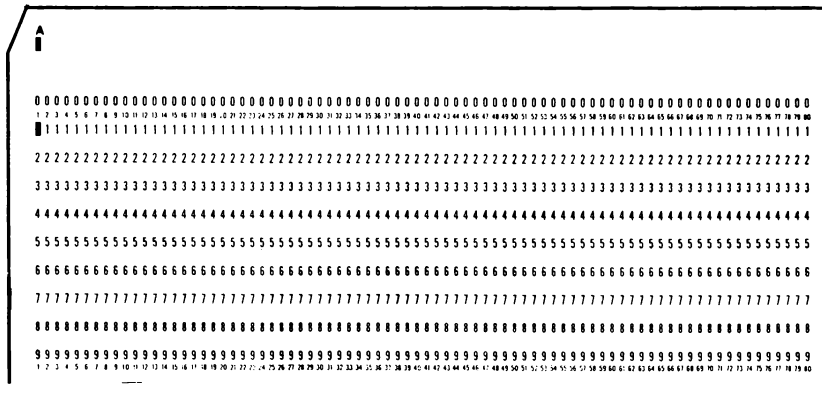
questa ne è la codifica binaria nonché l'effettiva rappresentazione in memoria;

questa ne è la rappresentazione in codice ottale come può essere letta nella tabella;

questa ne è la rappresentazione esadecimale come può essere letta nella tabella.

come si vede in corrispondenza di ogni terna binaria è stata scritta una cifra in ottale. Nella tabella è riportata anche la rappresentazione esadecimale (la colonna intestata con 16: in molti sistemi il byte ha una dimensione di 8 bit e, dunque, viene usata questa rappresentazione). Nella tabella la codifica Hollerith della lettera A, invece, reca scritto: 12-1. Ciò sta a significare che in una

scheda per rappresentare tale carattere sono presenti due perforazioni, la prima nella riga 12, la seconda nella 1:



La tabella serve per conoscere quale sia la rappresentazione dei caratteri nei vari codici; in memoria, ovviamente, non sarà mai presente una codifica Hollerith (propria delle schede), ma non è neppure detto che i dati si trovino sempre in memoria secondo la rappresentazione a caratteri, anzi questo è il caso del solo formato CHARACTER. Gli altri formati, quelli numerici, fanno riferimento alla traduzione del numero in base binaria, come si è detto nel testo, e dunque la tabella non serve per i casi di costanti o variabili appartenenti alle categorie INTEGER, REAL, DOUBLE PRECISION e COMPLEX.

Ancora un avvertimento per quanto riguarda i caratteri che si trovano dall'inizio della tabella fino all'ottale 37, cioè i primi 31, nonché l'ultimo: si tratta di caratteri particolari che non hanno una rappresentazione usuale concreta, si tratta di *segnali* che il sistema usa nel caso della *trasmissione dati* e che spesso possono essere comunicati al sistema dal programmatore trattandoli come veri e propri caratteri. L'argomento non ci riguarda in questa sede, tuttavia abbiamo voluto riportarli ugualmente per completezza.

Il nome di alcuni simboli e di alcuni caratteri è in inglese o perché così sono stati battezzati quando i vari codici sono stati stabiliti, o perché l'uso vuole che così vengano comunque chiamati.

| CARATTERE O SIMBOLO | NOME | CODICI | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | ASCII | | EBCDIC | | BCD | | HOLLERITH |
| | | 16 | 8 | 16 | 8 | 16 | 8 | |
| NUL | Null | 00 | 000 | 00 | 000 | | | |
| SOH | Start of Heading | 01 | 001 | 01 | 001 | | | |
| STX | Start of Text | 02 | 002 | 02 | 002 | | | |
| ETX | End of Text | 03 | 003 | 03 | 003 | | | |
| EOT | End of Transmission | 04 | 004 | 37 | 067 | | | |

| CARATTERE O SIMBOLO | NOME | CODICI | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | ASCII | | EBCDIC | | BCD | | HOLLERITH |
| | | 16 | 8 | 16 | 8 | 16 | 8 | |
| ENQ | Enquiry | 05 | 005 | 2D | 055 | | | |
| ACK | Acknowledge | 06 | 006 | 2E | 056 | | | |
| BEL | Bell (Audible Signal) | 07 | 007 | 2F | 057 | | | |
| BS | Backspace | 08 | 010 | 16 | 026 | | | |
| HT | Horizontal Tab (Punch Card Skip) | 09 | 011 | 05 | 005 | | | |
| LF | Line Feed | 0A | 012 | 25 | 045 | | | |
| VT | Vertical Tabulation | 0B | 013 | 0B | 013 | | | |
| FF | Form Feed | 0C | 014 | 0C | 014 | | | |
| CR | Carriage Return | 0D | 015 | 0D | 015 | | | |
| SO | Shift Out | 0E | 016 | 0E | 016 | | | |
| SI | Shift In | 0F | 017 | 0F | 017 | | | |
| DLE | Data Link Escape | 10 | 020 | 10 | 020 | | | |
| DC1 | Device Control 1 | 11 | 021 | 11 | 021 | | | |
| DC2 | Device Control 2 | 12 | 022 | 12 | 022 | | | |
| DC3 | Device Control 3 | 13 | 023 | 13 | 023 | | | |
| DC4 | Device Control 4 | 14 | 024 | 3C | 074 | | | |
| NAK | Negative Acknowledge | 15 | 025 | 3D | 075 | | | |
| SYN | Synchronous Idle | 16 | 026 | 32 | 062 | | | |
| ETB | End of Transmission Block | 17 | 027 | 26 | 046 | | | |
| CAN | Cancel | 18 | 030 | 18 | 030 | | | |
| EM | End of Medium | 19 | 031 | 19 | 031 | | | |
| SUB | Substitute | 1A | 032 | 3F | 077 | | | |
| ESC | Escape | 1B | 033 | 27 | 047 | | | |
| IFS | File Separator | 1C | 034 | 1C | 034 | | | |
| IOS | Group Separator | 1D | 035 | 1D | 035 | | | |
| IRS | Record Separator | 1E | 036 | 1E | 036 | | | |
| IUS | Unit Separator | 1F | 037 | 1F | 037 | | | |
| | Blank | 20 | 040 | 40 | 100 | 10 | 20 | Blank |
| ! | Punto esclamativo | 21 | 041 | 4F | 117 | 3F | 77 | 0-7-8 |
| " | Virgolette | 22 | 042 | 7F | 177 | 3E | 76 | 0-6-8 |
| # | | 23 | 043 | 7B | 173 | 0B | 13 | 3-8 |
| $ | | 24 | 044 | 5B | 130 | 2B | 53 | 11-3-8 |
| % | | 25 | 045 | 6C | 154 | 3C | 74 | 0-4-8 |
| & | Ampersand | 26 | 046 | 50 | 120 | 1A | 32 | 12 |
| ' | Apostrofo | 27 | 047 | 70 | 175 | 2F | 57 | 11-7-8 |
| ( | Parentesi aperta | 28 | 050 | 4D | 115 | 1D | 35 | 12-5-8 |
| ) | Parentesi chiusa | 29 | 051 | 5D | 135 | 2D | 55 | 11-5-8 |
| * | Asterisco | 2A | 052 | 5C | 134 | 2C | 54 | 11-4-8 |
| + | Più | 2B | 053 | 4E | 116 | 30 | 60 | 12-0 |
| , | Virgola | 2C | 054 | 6B | 153 | 3B | 73 | 0-3-8 |
| - | Trattino | 2D | 055 | 60 | 140 | 2A | 52 | 11 |
| . | Punto | 2E | 056 | 4B | 113 | 1B | 33 | 12-3-8 |
| / | Barra | 2F | 057 | 61 | 141 | 31 | 61 | 0-1 |
| 0 | | 30 | 060 | F0 | 360 | 00 | 00 | 0 |
| 1 | | 31 | 061 | F1 | 361 | 01 | 01 | 1 |
| 2 | | 32 | 062 | F2 | 362 | 02 | 02 | 2 |
| 3 | | 33 | 063 | F3 | 363 | 03 | 03 | 3 |
| 4 | | 34 | 064 | F4 | 364 | 04 | 04 | 4 |
| 5 | | 35 | 065 | F5 | 365 | 05 | 05 | 5 |
| 6 | | 36 | 066 | F6 | 366 | 06 | 06 | 6 |
| 7 | | 37 | 067 | F7 | 367 | 07 | 07 | 7 |
| 8 | | 38 | 070 | F8 | 370 | 08 | 10 | 8 |
| 9 | | 39 | 071 | F9 | 371 | 09 | 11 | 9 |
| : | Due punti | 3A | 072 | 7A | 172 | 0D | 15 | 5-8 |
| ; | Punto e virgola | 3B | 073 | 5E | 136 | 2E | 56 | 11-6-8 |
| < | Minore | 3C | 074 | 4C | 114 | 1E | 36 | 12-6-8 |
| = | Uguale | 3D | 075 | 7E | 176 | 3D | 75 | 0-5-8 |
| > | Maggiore | 3E | 076 | 6E | 156 | 0E | 16 | 6-8 |
| ? | Punto interrogativo | 3F | 077 | 6F | 157 | 0F | 17 | 7-8 |
| @ | A commerciale | 40 | 100 | 7C | 174 | 0C | 14 | 4-8 |
| A | | 41 | 101 | C1 | 301 | 11 | 21 | 12-1 |
| B | | 42 | 102 | C2 | 302 | 12 | 22 | 12-2 |
| C | | 43 | 103 | C3 | 303 | 13 | 23 | 12-3 |
| D | | 44 | 104 | C4 | 304 | 14 | 24 | 12-4 |
| E | | 45 | 105 | C5 | 305 | 15 | 25 | 12-5 |

| CARATTERE O SIMBOLO | NOME | CODICI | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | ASCII | | EBCDIC | | BCD | | HOLLERITH |
| | | 16 | 8 | 16 | 8 | 16 | 8 | |
| F | | 46 | 106 | C6 | 306 | 16 | 26 | 12-6 |
| G | | 47 | 107 | C7 | 307 | 17 | 27 | 12-7 |
| H | | 48 | 110 | C8 | 310 | 18 | 30 | 12-8 |
| I | | 49 | 111 | C9 | 311 | 19 | 31 | 12-9 |
| J | | 4A | 112 | D1 | 321 | 21 | 41 | 11-1 |
| K | | 4B | 113 | D2 | 322 | 22 | 42 | 11-2 |
| L | | 4C | 114 | D3 | 323 | 23 | 43 | 11-3 |
| M | | 4D | 115 | D4 | 324 | 24 | 44 | 11-4 |
| N | | 4E | 116 | D5 | 325 | 25 | 45 | 11-5 |
| O | | 4F | 117 | D6 | 326 | 26 | 46 | 11-6 |
| P | | 50 | 120 | D7 | 327 | 27 | 47 | 11-7 |
| Q | | 51 | 121 | D8 | 330 | 28 | 50 | 11-8 |
| R | | 52 | 122 | D9 | 331 | 29 | 51 | 11-9 |
| S | | 53 | 123 | E2 | 342 | 32 | 62 | 0-2 |
| T | | 54 | 124 | E3 | 343 | 33 | 63 | 0-3 |
| U | | 55 | 125 | E4 | 344 | 34 | 64 | 0-4 |
| V | | 56 | 126 | E5 | 345 | 35 | 65 | 0-5 |
| W | | 57 | 127 | E6 | 346 | 36 | 66 | 0-6 |
| X | | 58 | 130 | E7 | 347 | 37 | 67 | 0-7 |
| y | | 59 | 131 | E8 | 350 | 38 | 70 | 0-8 |
| Z | | 5A | 132 | E9 | 351 | 39 | 71 | 0-9 |
| [ | | 5B | 133 | 4A | 112 | 0A | 12 | 2-8 |
| | Reverse Slant | 5C | 134 | E0 | 340 | 1F | 37 | 12-7-8 |
| ] | | 5D | 135 | 5A | 132 | 1C | 34 | 12-4-8 |
| ^ | Accento circonflesso | 5E | 136 | 5F | 137 | 20 | 40 | 11-0 |
| — | Underline | 5F | 137 | 6D | 155 | 3A | 72 | 0-2-8 |
| ` | Accento grave | 60 | 140 | 79 | 171 | | | |
| a | | 61 | 141 | 81 | 201 | | | |
| b | | 62 | 142 | 82 | 202 | | | |
| c | | 63 | 143 | 83 | 203 | | | |
| d | | 64 | 144 | 84 | 204 | | | |
| e | | 65 | 145 | 85 | 205 | | | |
| f | | 66 | 146 | 86 | 206 | | | |
| g | | 67 | 147 | 87 | 207 | | | |
| h | | 68 | 150 | 88 | 210 | | | |
| i | | 69 | 151 | 89 | 211 | | | |
| j | | 6A | 152 | 91 | 221 | | | |
| k | | 6B | 153 | 92 | 222 | | | |
| l | | 6C | 154 | 93 | 223 | | | |
| m | | 6D | 155 | 94 | 224 | | | |
| n | | 6E | 156 | 95 | 225 | | | |
| o | | 6F | 157 | 96 | 226 | | | |
| p | | 70 | 160 | 97 | 227 | | | |
| q | | 71 | 161 | 98 | 230 | | | |
| r | | 72 | 162 | 99 | 231 | | | |
| s | | 73 | 163 | A2 | 242 | | | |
| t | | 74 | 164 | A3 | 243 | | | |
| u | | 75 | 165 | A4 | 244 | | | |
| v | | 76 | 166 | A5 | 245 | | | |
| w | | 77 | 167 | A6 | 246 | | | |
| x | | 78 | 170 | A7 | 247 | | | |
| y | | 79 | 171 | A8 | 250 | | | |
| z | | 7A | 172 | A9 | 251 | | | |
| ( | | 7B | 173 | C0 | 300 | | | |
| \| | Broken Vertical Line | 7C | 174 | 6A | 152 | | | |
| ) | | 7D | 175 | D0 | 320 | | | |
| ~ | Tilde | 7E | 176 | A1 | 241 | | | |
| DEL | Delete | 7F | 177 | 07 | 007 | | | 12-7-9 |

APPENDICE E

# ALCUNI STRUMENTI MATEMATICI

Questa appendice ha lo scopo di rendere facilmente comprensibili alcuni degli strumenti matematici normalmente utilizzati nella programmazione tecnico-scientifica. Spesso alcune conoscenze di questo tipo si rendono indispensabili e chi affronta tali argomenti essendone privo si trova in qualche difficoltà: a costoro essa è specificatamente diretta e (visto l'obbiettivo) non contiene definizioni accurate né trattazioni di tipo assiomatico. La sua finalità è essenzialmente pratica ed il suo carattere vuol essere soprattutto divulgativo.

Nella maggior parte dei casi gli argomenti che sgomentano l'inesperto sono: i vettori e le matrici e le numerazioni a base non decimale. Di tali questioni cercheremo, in breve, di dare un'idea.

VETTORI E MATRICI - Quando si pensa ad una grandezza si pensa, normalmente, ad una quantità. Ciò non è vero in assoluto, anzi in natura esistono grandezze la cui definizione richiede più di una indicazione, più di un numero (la quantità, o la quantizzazione di quella indicazione). Si pensi semplicemente alla definizione della posizione di un oggetto nella nostra stanza o, se si preferisce, alla definizione di un punto: dovremo almeno specificare a che altezza si trova dal pavimento, a quale distanza dalla parete alla nostra sinistra ed a quale dalla parete di fronte a noi. Insomma, la definizione della posizione avrà richiesto almeno tre numeri. Ebbene tale terna di numeri non è altro che un *vettore.*

Va forse aggiunto che la maggior parte delle grandezze fisiche richiede, come nel caso della posizione, più indicazioni o più numeri e che, in matematica, è possibile ragionare su grandezze definite da più di tre numeri, anzi è possibile parlare di grandezze definite da n numeri.

Proprio per meglio capire cosa sia un vettore fissiamo l'attenzione su quei tre, (ma anche quattro, o ... n numeri andrebbero bene): in pratica si è detto che una grandezza **X** può essere espressa da più quantità, o che, insomma, si può scrivere:

$$\mathbf{X} = [x_1 , x_2 , ..., x_n ]$$

e che la grandezza **X** si dice *vettore*. I vari valori $x_i$ verranno detti *componenti del vettore* ed n la sua *dimensione*. Il valore di i (relativamente ad $x_i$ ), ovvero la posizione ordinale della componente, verrà detto *indice* del *vettore*.

Si badi bene che con ciò non si è assolutamente voluto dare una definizione rigorosa in temini matematici o fisici. Quest'ultima richiederebbe le conoscenze di ben altri concetti e X dovrebbe soddisfare precise condizioni. Si è solo cercato di suggerire un concetto con finalità, come si è già detto all'inizio, essenzialmente pratiche. Non torneremo più su questo punto: non si tratta qui di fare un sunto di definizioni matematiche, ma di comprendere come vadano trattate, almeno nella programmazione, certi tipi di grandezze.

Lasciamo dunque da parte il fatto che il vettore X in realtà dovrebbe essere un elemento dello spazio ad n dimensioni (il matematico già lo sa) e notiamo invece che nelle applicazioni si ha spesso a che fare con grandezze di questo tipo: esse ci sono ovviamente necessarie trattando problemi di tipo matematico, fisico, o comunque scientifico in generale, ma anche in quei più semplici casi cui la pratica quotidiana spesso ci conduce.

Riduciamo la questione e riferiamoci ad un esempio: immaginiamo di voler calcolare la media delle temperature rilevate durante una giornata. Di esse è stata fatta una misurazione, con un preciso termometro, ogni due ore e ci troviamo, dunque, a disporre di dodici numeri; dovremo farne la somma e dividere per dodici.

Quei dodici numeri non hanno, in verità, la caratteristica di un vettore (essi non esprimono una singola e tipica grandezza fisica), tuttavia per la nostra applicazione pratica, per il nostro programma, potrà essere assai comodo definire una tabella di dodici elementi (i dodici valori della temperatura, appunto) in modo da poterla facilmente scorrere, sulla quale poter effettuare immediate verifiche, somme e totalizzazioni, sulla quale, in poche parole, poter eseguire facili elaborazioni.

Ora, se in fisica ed in matematica il *vettore* ha un suo senso specifico e ci permette di definire correttamente delle grandezze ben precise, è però immediato accorgersi che la sua *struttura tabellare* può soccorrerci in una miriade di altri piccoli e grandi problemi. In effetti, nella pratica della programmazione FORTRAN, non si fa alcuna differenza di terminologia (ci piacerebbe poter dire: ingiustamente) tra la *tabella* delle dodici temperature e la terna di numeri che definisce la posizione: entrambe vengono dette *vettori* (o, spesso, come anche nel testo, *array*), il primo a 12 componenti, il secondo a 3.

Desideriamo ribadire che tra le due cose esiste una sostanziale differenza: la prima è una *tabella*, il secondo un *vettore* nel senso fisico del termine. Tuttavia l'elaboratore non ama simili sottigliezze. Per la nostra macchina più o meno intelligente entrambe non sono che zone di memoria univocamente identificabili e del tutto analoghe quanto a struttura e caratteristiche di utilizzo. Così, se nel nostro programma abbiamo a che fare con le temperature T rilevate in una posizione P della nostra stanza, possiamo definire e dimensionare i due vettori T e P con l'istruzione opportuna (DIMENSION, v. nel testo) immagazzinando poi in 12 celle di memoria consecutive le temperature

ed in altre 3 (anch'esse tra loro consecutive) le indicazioni di posizione. Dopodichè la temperatura rilevata tra la mezzanotte e le due verrà identificata da T(1), quella tra le due e le quattro da T(2) e così via, mentre l'altezza dal pavimento dal punto in cui si trova il termometro con P(1) e le distanze dalle pareti con P(2) e P(3).

Tuttavia, né per le grandezze vettoriali, né per le descirizioni di tipo tabellare, la faccenda si ferma qui. Alcune grandezze fisiche possono venir definite non più da un solo vettore, ma da un accostamento di più vettori; pensiamo di mettere più vettori di uguale dimensione n uno sopra l'altro per m volte: avremo definito una *matrice* di m righe ed n colonne, o, come si dice più propriamente, di *ordine* (a volte *dimensioni*) m per n.

Al solito, riducendo il problema a questioni più semplici e guardando maggiormente alla pratica, il concetto di matrice può risultare più facilmente comprensibile. Allora, cambiando completamente esempio, immaginiamo (beati noi) di disporre di tre automobili e di voler memorizzare nel nostro elaboratore il consumo di carburante che esse hanno prodotto in quattro giorni consecutivi compiendo più o meno lo stesso percorso: si potrebbe compilare una tabella, poniamo, come la seguente:

| | **giorno 1** | **giorno 2** | **giorno 3** | **giorno 4** |
|---|---|---|---|---|
| auto 1 | 20 | 25 | 50 | 43 |
| auto 2 | 25 | 31 | 80 | 51 |
| auto 3 | 15 | 18 | 62 | 35 |

per poi passarla alla memoria dell'elaboratore battezzandola con un nome qualunque, poniamo AUT. I valori numerici andaranno a disporsi in dodici celle di memoria consecutive identificabili una per una. Per la definizione non dovremo fare altro che dimensionare (ancora, in FORTRAN, con lo statement DIMENSION) una *matrice* (o *array*) di tre righe e quattro colonne e per identificare la singola componente usare gli indici opportuni: così AUT(1,1) individuerà il consumo della prima auto, il primo giorno, AUT(1,2) quello della prima auto il secondo giorno, AUT(2,3) quello della seconda auto il terzo giorno e così via.

C'è da augurarsi che, posta in questi termini, la questione sembri banale. In effetti è così, tuttavia esistono notevoli sviluppi del concetto di *matrice* e ben presto l'esperienza mostrerà quanto essa possa risultare utile nelle più diverse occasioni.

E' possibile accennare ad alcuni di quegli sviluppi esaminando la seguente generalizzazione: proviamo ad immaginare un insieme di tabelle come la precedente, scritte su più fogli, ad esempio cinque. Si potrà definire una *matrice* con un numero di indici pari a 3, il primo (che può variare da 1 a 3) riferito all'auto, il secondo (da 1 a 4) riferito ai giorni, il terzo (da 1 a 5) riferito al foglio di rilevamento. Ancora sarà possibile una corretta memorizzazione e

individuazione delle singole componenti. Ma la generalizzazione non termina qui: va ricordato infatti che il numero di indici può essere anche più elevato di tre conducendo a degli *array* di una certa complessità e che, almeno per le nostre capacità, non sono facilmente raffigurabili o graficamente rappresentabili (il FORTRAN permette fino a sette indici). Essi saranno tuttavia di sicura utilità ed il nostro elaboratore, con la sua mancanza di immaginazione figurativa, potrà agevolmente trattarli.

NUMERAZIONI A BASE NON DECIMALE - La nascita delle numerazioni a base non decimale è anteriore all'uso dell'elaboratore, ma è proprio grazie a quest'ultimo che la più semplice di esse, la numerazione binaria, ha assunto irrinunciabile importanza.

Tutti i circuiti di un elaboratore non possono (per ora) che venir costruiti in un modo: essi, stimolati da una corrente elettrica o da un campo magnetico, danno solo risposte del tipo SI - NO, forniscono una corrente di risposta oppure nessuna, i nuclei magnetici si trovano in uno stato di magnetizzazione o nel suo opposto. Questi stati (o risposte) possono allora venir fatti corrispondere a due simboli: ad esempio 0 e 1 che ci permettano di rappresentare ogni situazione. Si pensi alla memoria: un insieme di nucleini magnetici (bit di informazione) magnetizzati (1) e non magnetizzati (0) potrà recare un'informazione anche assai complessa (dipenderà dalla decodifica di quegli 1 e 0), potrà recare la registrazione di un numero qualora se ne pensi una rappresentazione opportuna dopo averlo convertito dalla scrittura in cui siamo abituati a pensarlo.

Proprio qui sta il punto: siamo abituati a pensare i numeri in base 10, ma cosa ci impedisce di scriverli in altro modo? Possiamo sicuramente rispondere che si tratta solo di convenzione ed abitudine.

A pensarci bene l'uso della base dieci nasce con ogni probabilità dal fatto che possediamo dieci dita e che, dunque, i nostri antenati non hanno fatto altro che abituarsi ad enumerare proprio utilizzando gli strumenti più a portata di ...mano. E noi cosa abbiamo aggiunto? Ad ogni entità numerica abbiamo messo in relazione un simbolo, esauriti i primi dieci (le cifre da 0 a 9) abbiamo costruito simboli delle altre, successive *quantità*, accostando tra loro i primi dieci e sempre partendo con quello che indica l'unità.

Prima di proseguire su questa strada e per meglio comprendere il senso di questo discorso, è bene fare alcune riflessioni, slegarci dal nostro usuale concetto di numerazione ed affrontare il problema alla radice. Immaginiamo di avere a nostra disposizione solo i seguenti quattro simboli:

$$0 * \triangle \uparrow$$

e di volere, con questi, costruire una numerazione. Essa si dirà in base 4 poiché, per l'appunto, disponiamo solo di quattro simboli. Inizieremo col

designare la *quantità nulla* con 0, la quantità riferibile ad un solo oggetto con *, a due oggetti con △ , a tre con ↑ , e poi? Potremmo tranquillamente scrivere *0 per indicarne quattro, ** per indicarne cinque, *△ per indicarne sei, *↑ per indicare sette, e quindi ricominciare con gli accostamenti in sequenza: △ 0 ad indicarne otto, △ * ad indicarne nove e così via, per poi passare all'uso di tre simboli, quattro e via dicendo fino a rendere completa la nostra numerazione.

A questo punto si sarà sicuramente notata la perfetta analogia con il metodo abituale di numerazione in base dieci; se ciò non fosse accaduto si passi ad un attento esame della tabella che segue, ma non prima di aver osservato che se avessimo avuto a disposizione l'esiguo numero di due soli simboli, poniamo 0 e 1 come nella numerazione binaria, il concetto non sarebbe affatto cambiato. Semmai sarebbe stato più lungo scrivere il numero, vista l'elevata quantità di accostamenti necessari (ma nell'elaboratore di spazio ce n'è sicuramente a sufficienza).

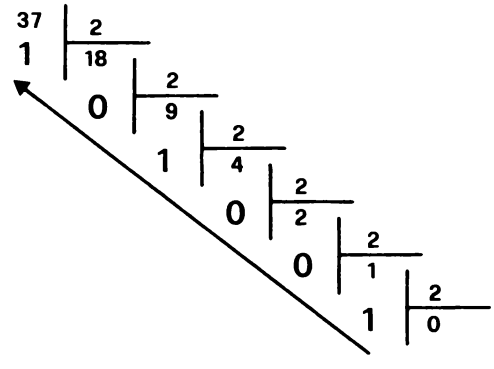
Nella tabella è presente la numerazione con quei nostri quattro simboli, la numerazione decimale (più che altro come riferimento), la binaria con i simboli 0 e 1 ed infine la ottale composta con i simboli delle cifre da 0 a 7:

| | **decimale** | **binaria** | **ottale** |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| * | 1 | 1 | 1 |
| △ | 2 | 10 | 2 |
| ↑ | 3 | 11 | 3 |
| *0 | 4 | 100 | 4 |
| ** | 5 | 101 | 5 |
| *△ | 6 | 110 | 6 |
| *↑ | 7 | 111 | 7 |
| △0 | 8 | 1000 | 10 |
| △* | 9 | 1001 | 11 |
| △ △ | 10 | 1010 | 12 |
| △ ↑ | 11 | 1011 | 13 |
| ↑0 | 12 | 1100 | 14 |
| ↑* | 13 | 1101 | 15 |
| ↑ △ | 14 | 1110 | 16 |
| ↑ ↑ | 15 | 1111 | 17 |
| *00 | 16 | 10000 | 20 |
| *0* | 17 | 10001 | 21 |
| *0△ | 18 | 10010 | 22 |
| *0↑ | 19 | 10011 | 23 |
| **0 | 20 | 10100 | 24 |
| *** | 21 | 10101 | 25 |
| **△ | 22 | 10110 | 26 |
| **↑ | 23 | 10111 | 27 |
| *△0 | 24 | 11000 | 30 |
| ... | ... | ... | ... |
| ... | ... | ... | ... |

Come si può vedere tutto sta nel far variare il simbolo a destra più velocemente degli altri; una volta esaurita la lista ripartire accostando a sinistra quello corrispondente all'unità e procedere così. Il trucco è così semplice! Forse lo sono meno le conversioni da una base ad un'altra, quando non si disponga di una tabella costruita ad hoc, e le operazioni aritmetiche. Almeno delle conversioni daremo un cenno tra breve, per ora basti notare come con la numerazione binaria, a patto di un po' di pazienza, si possa esprimere qualunque numero si desideri. Ciò non è poco: il problema di memorizzare delle quantità nel nostro elaboratore è sicuramente risolto. E lo è anche per quanto riguarda altri tipi di informazione: per le lettere dell'alfabeto o altri simboli sarà sufficiente stabilire dei codici di corrispondenza tra configurazione (non più numero) binaria ed informazione stessa e ciò potrà valere anche per le stesse istruzioni, gli ordini che noi intendiamo comunicare all'elaboratore. Questi ultimi codici non avranno, ovviamente, nulla a che fare con la numerazione in base due vera e propria, ma, ed è quel che più conta, sarà stato stabilito in questo modo un *linguaggio* comprensibile da parte dell'elaboratore ed, in definitiva, un mezzo di comunicazione tra l'uomo ed i circuiti della macchina.

Per quanto riguarda il problema delle conversioni ci limiteremo ad esporre due esempi che mostrino come si passi da base decimale a base due e viceversa: il metodo potrà essere generalizzato ad altre basi.

Se si volesse convertire il numero decimale 37 nel corrispondente in base due, andrebbero eseguite le seguenti successive divisioni per 2:

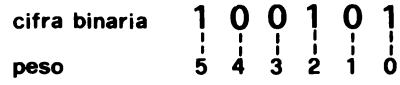



i resti andrebbero quindi letti in senso inverso a produrre il numero binario:

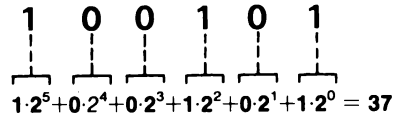
1 0 0 1 0 1

La conversione contraria si basa sul fatto che ogni cifra binaria ha, in

funzione della sua posizione, un *peso* che cresce spostandosi da destra verso sinistra:




si ottiene il numero decimale moltiplicando ogni cifra binaria per 2 elevato al peso corrispondente ed eseguendo la somma:




Tali metodi sono ovviamente generalizzabili alla base ottale (ma anche ad altre basi) procedendo con le divisioni per 8 in un caso e con la somma delle potenze di 8 nell'altro.

# BIBLIOGRAFIA

American National Standard Institute - PROGRAMMING LANGUAGE FORTRAN (New York, 1978)

Carboni G., Farchi G., Giucci D. - INTRODUZIONE ALLA PROGRAMMAZIONE E LINGUAGGIO FORTRAN (Libreria Eredi Virgilio Veschi, Roma 1967)

Dreyfus M. - FORTRAN IV (Dunod, Paris 1967)

Falzone V., Costantini E. - ELABORATORI ELETTRONICI E TECNICA DELLA PROGRAMMAZIONE IN FORTRAN ED ALGOL (Hoepli, Milano 1967)

Katzan H. - FORTRAN 77 (Computer Science Series of Van Nostrand, Reinhold Company, New York 1978)

Lipschutz S., Poe A. - PROGRAMMARE IN FORTRAN (Etas Libri, Milano 1980)

Mc Cracken D. - GUIDA ALLA PROGRAMMAZIONE DEL FORTRAN IV (Edizioni Bizzarri, Roma 1974)

Morpurgo R. - PROGRAMMAZIONE DEI CALCOLATORI ELETTRONICI (Tamburini Editore, Milano 1972)

Ridolfi P. - IL FORTRAN (Franco Angeli Editore, Milano 1975)

Ridolfi P. - APPLICAZIONI DEL FORTRAN (Franco Angeli Editore, Milano 1976)

Thorin M. - EXERCISES COMMENTES DE FORTRAN (Masson, Paris 1978)

**Documentazione case costruttrici:**

G.E.I.S.

FORTRAN 77 - General Electric 1977

FORTRAN 77 - Network Information Service 1977

HONEYWELL

FORTRAN 77 REFERENCE MANUAL LEV. 66 - Honeywell 1980

FORTRAN 77 USER'S GUIDE LEV. 66 - Honeywell 1980

FORTRAN REFERENCE MANUAL LEV. 66 - Honeywell 1979

FORTRAN SUBROUTINE LIBRARIES LEV. 66 - Honeywell 1975

FORTRAN PROGRAM LOGIC MANUAL LEV. 66 - Honeywell 1977

FORTRAN STUDENT HANDBOOK - Honeywell I.S.I. 1981

UNIVAC
FUNDAMENTALS OF FORTRAN - UNIVAC 1974
FORTRAN PROGRAMMING REFERENCE - UNIVAC 1977
FORTRAN IV PROGRAMMING REFERENCE - UNIVAC 1980
EXTENDED FORTRAN SUPPL. REF. - UNIVAC 1976
FORTRAN IV OS3 PROGRAMMING REFERENCE - UNIVAC 1980
FORTRAN V PROGRAMMING REFERENCE - UNIVAC 1974
FORTRAN ASCII L6R1 PROGRAMMING REFERENCE - UNIVAC 1978

*RINGRAZIAMENTI*

*Un sentito ringraziamento riteniamo di dover esprimere nei riguardi di Rosalba VIANO che si è assunta l'oneroso impegno di rendere disponibile in breve tempo il primo dattiloscritto.*

*Gli Autori*

# INDICE ANALITICO

# DAL FORTRAN IV AL FORTRAN 77

ROBERTO FARABONE - ROBERTO DORETTI

Data la sua popolarità e diffusione, libri sul Fortran certamente non mancano. Tuttavia questo libro si distingue per una serie di caratteristiche che non è dato di trovare generalmente riunite: la chiarezza di esposizione e la gradualità con cui il lettore viene introdotto nell'argomento, la ricchezza di esempi e l'impostazione top-down seguita diffusamente nel testo, la completezza e l'aggiornamento del materiale. Dietro queste caratteristiche traspare la lunga esperienza degli autori, sia in campo didattico, sia nell'applicazione a problemi reali.
In questi ultimi anni, è vero, sono nati e si sono sviluppati linguaggi più potenti e anche più eleganti, ma il Fortran rimane sino ad oggi il più noto e più utilizzato fra coloro che si servono degli strumenti informatici per risolvere le problematiche a carattere tecnico-scientifico.
Il libro è alla sua seconda edizione. Ciò che ha spinto alla revisione del testo non è stata tanto la legittima volontà di eliminare inevitabili inesattezze, ma la necessità di adeguarsi al rapido cambiamento che l'informatica ha imposto in questi ultimi anni, soprattutto nel modo di lavorare e di affrontare determinati problemi.
Le modifiche o le aggiunte occupano poco spazio, è il discorso nel suo complesso che assume un aspetto diverso. La nuova impostazione fa infatti riferimento a quei mezzi che la recente tecnologia ha messo a disposizione, quali i terminali e il Personal Computer, che consentono di scrivere i programmi in modo interattivo, rendendo il colloquio uomo-macchina più facile e vantaggioso.
Degli stessi autori il Gruppo Editoriale Jackson ha pubblicato "Esercizi di Fortran" che completa e arricchisce la trattazione e ha il preciso scopo di facilitare l'apprendimento pratico del linguaggio.

**Roberto DORETTI: laureatosi in Fisica all'Università degli Studi di Milano, svolge la sua attività presso il Servizio Formazione di una grande azienda costruttrice di elaboratori.**
**Roberto FARABONE: laureato in Fisica all'Università degli Studi di Milano, ha lavorato alcuni anni come sistemista in una grande azienda produttrice di elaboratori, entrando in seguito al Servizio Addestramento come istruttore; qui si è occupato di corsi base, di linguaggi e sistemi operativi, indirizzandosi in seguito alle tematiche di AI, sempre tenendo in particolare considerazione gli aspetti legati alla formazione. È autore, per il Gruppo Editoriale Jackson, dei volumi "Logica e diagrammi a blocchi" con R. Viano, "Fondamenti di COMMON LISP" con L. Pinotti.**

**GRUPPO EDITORIALE JACKSON**

ISBN 88-7056-119-4

**L. 35.000** Cod. 517P

ROBERTO FARABONE
ROBERTO DORETTI

# DAL FORTRAN IV AL FORTRAN 77