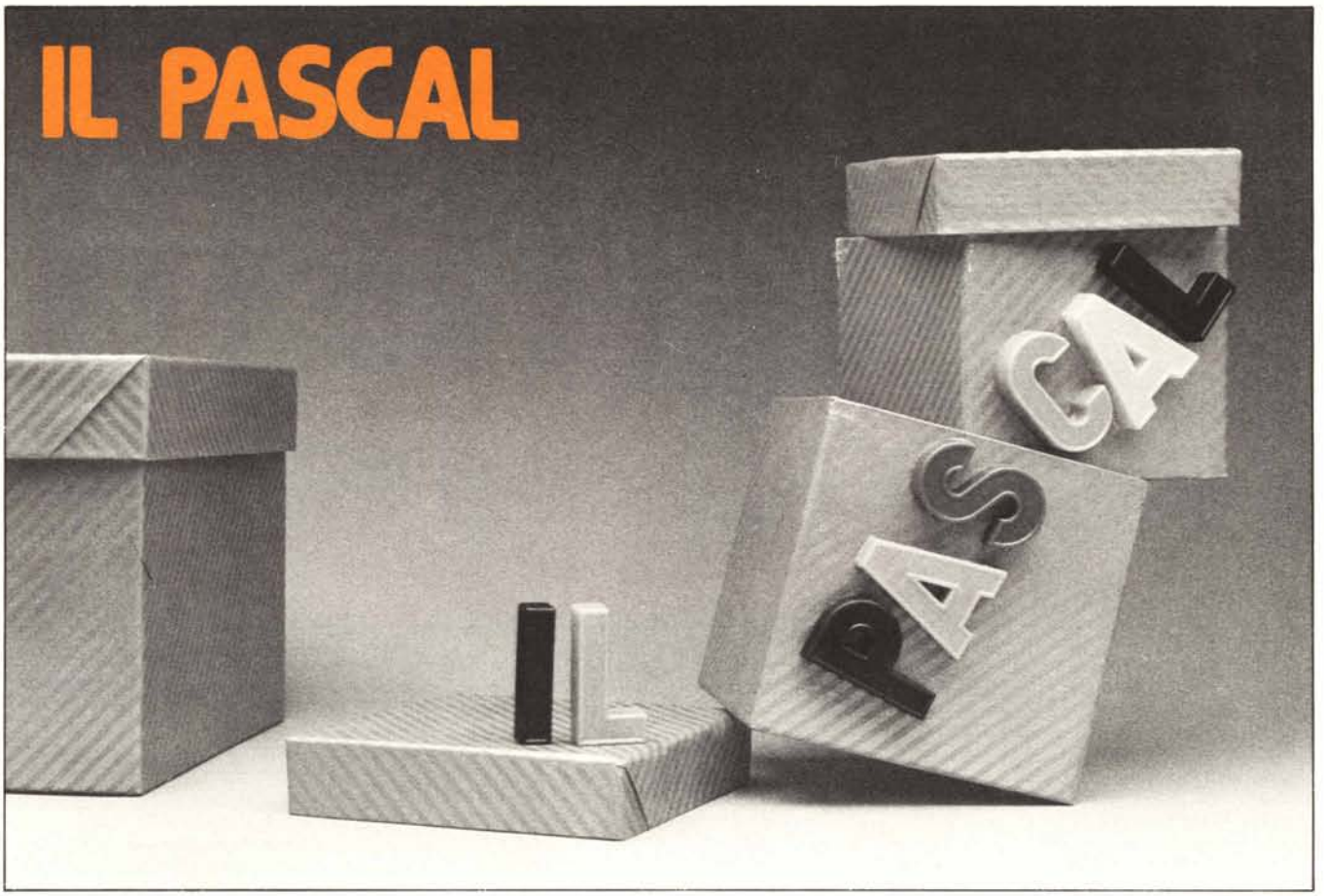
IL PASCAL



Quarta parte

Le istruzioni di programma

Dopo avere speso tre intere puntate a descrivere le strutture di dati su cui il linguaggio lavora, iniziamo finalmente a vedere come si può lavorare su questi dati: la parte esecutiva del PASCAL non è meno affascinante di quella dichiarativa.

Se avessimo concluso il nostro microcorso sul PASCAL alla fine della scorsa puntata e fossimo passati direttamente a programmare, ci saremmo trovati come Michelangelo davanti al suo Mosè, e forse avremmo tirato anche noi la nostra brava martellata su un immaginario ginocchio del nostro software esclamando: "Perché non parli?"

In effetti, con quanto abbiamo visto finora (a parte alcuni accenni necessari per gli esempi), possiamo costruire le più complesse e intelligenti strutture di dati, ma ci mancano ancora gli strumenti per potere eseguire su di esse le operazioni necessarie.

Scopo di questa puntata è di illustrare l'insieme di istruzioni del linguaggio PASCAL, e vedere come le poche strutture di assegnamento e di controllo definite da Wirth — poche in confronto alla marea di dichiarazioni che abbiamo esaminato fino ad ora — bastino ed avanzino per potere programmare nel modo più stringato ed essenziale possibile.

La programmazione strutturata

Le istruzioni cosiddette operative di un linguaggio possono dividersi in tre tipi fondamentali:

— istruzioni di assegnamento
— istruzioni di controllo
— istruzioni di gestione delle periferiche.

Lasciando da parte l'ultimo tipo (formato da istruzioni come OPEN, GET, CLOSE etc.) che dipende più dal sistema operativo che dal linguaggio in sé, è evidente a colpo d'occhio come un linguaggio si differenzi da un altro più per le istruzioni di controllo che per quelle di assegnamento.

Infatti la generica istruzione di assegnamento del tipo

variabile = espressione

è uguale in tutti i linguaggi ad alto livello, dal FORTRAN al BASIC fino al PASCAL, e soltanto in pochi linguaggi specializzati come il PL/1 esistono istruzioni di assegnamento particolari, capaci ad esempio di dare in blocco un valore a tutti gli elementi di una matrice.

Il discorso cambia entrando nel campo delle istruzioni di controllo: nei linguaggi ad alto livello queste ultime sono le dirette discendenti delle istruzioni di salto (più o meno condizionato) dei linguaggi assemblatori, ed hanno il compito di "correggere il tiro" del contatore di programma durante l'esecuzione in modo da eseguire cicli o percorsi alternativi.

Questo, e solo questo, è lo scopo di tutte le istruzioni di controllo, dalla semplice GOTO alle più complesse nidificazioni di IF... THEN... ELSE ai cicli FOR... NEXT, tanto per restare in ambiente BASIC.

Se è possibile scrivere un programma senza istruzioni di controllo, e quindi con sole istruzioni di assegnamento, non è possibile, o meglio non ha senso, scrivere un programma senza istruzioni di assegnamento: infatti un programma con sole istruzioni di controllo si limiterebbe a far saltabeccare il PC da un posto all'altro della memoria senza compiere alcuna operazione effettiva. Le istruzioni di assegnamento sono dunque il cuore di qualsiasi linguaggio, mentre quelle di controllo sono un corollario necessario ma non sufficiente.

Con tutto ciò, le istruzioni di controllo vengono ad essere, nel campo del software pratico, molto più sofisticate di quelle di assegnamento, se non altro perché richiedono un maggiore sforzo logico per essere controllate dal programmatore. Mentre infatti è abbastanza facile venire a capo di un'operazione di assegnamento, per complicata che sia (basta un minimo di attenzione alle parentesi, e al limite la si spezza usando variabili intermedie), è spesso complicato e richiede una grande padronanza della logica, orizzontarsi in mezzo ad una giungla di GOTO e di IF.

I programmatori FORTRAN, ad esempio, per non perdersi nei programmi di un certo volume, usavano congiungere con tratti di penna le istruzioni GOTO con le loro destinazioni; il risultato di quest'operazione diede il nome di "spaghetti-like programming" al modo di gestire il controllo tipico del FORTRAN (e, in un certo senso, anche del BASIC), ossia usando le sole istruzioni IF e GOTO.

Fu per eliminare le difficoltà della "programmazione a spaghetti" che si decise di sviluppare linguaggi con istruzioni di controllo più sofisticate e comprensibili al programmatore. Si può dire che il concetto stesso di programmazione strutturata, e quindi l'essenza del PASCAL, derivano da questa esigenza.

La programmazione strutturata ha infatti co-

Fig.1a

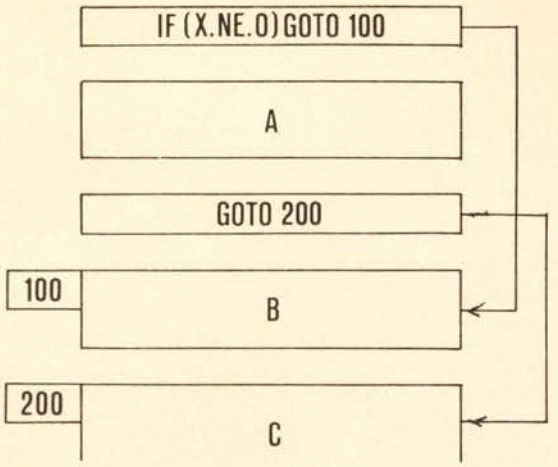
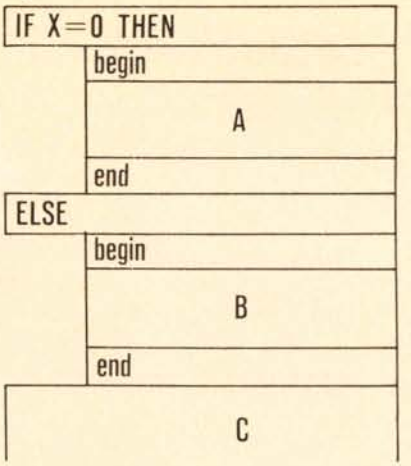


Fig.1b



me caratteristica fondamentale una suddivisione del programma in moduli, che possono presentarsi in successione, ma che più spesso stanno uno dentro l'altro come le scatole cinesi. L'uso spinto di questa tecnica — che, beninteso, richiede le istruzioni adatte e non può essere quindi adoperata in linguaggi come il FORTRAN — porta alla quasi totale eliminazione dell'istruzione GOTO, e di conseguenza della famigerata programmazione a spaghetti. Ciò ha portato alcuni sprovveduti a definire la programmazione strutturata come quel tipo di programmazione ove non compare l'istruzione GOTO: l'assenza di salti diretti è un effetto, non una caratteristica, della strutturazione, e può benissimo esistere un programma, ad esempio una successione sequenziale di operazioni, senza nemmeno una GOTO, eppure non strutturato.

Vediamo invece in che cosa consiste la programmazione strutturata dal punto di vista delle istruzioni di controllo del PASCAL. Essa si basa su un uso intensivo dell'istruzione *compound*, che abbiamo analizzato nella prima puntata, e nell'apertura e chiusura di parentesi nel corso del programma.

Il confronto con la programmazione a spaghetti si può vedere molto bene nell'esempio di fig. 1, dove si presenta la struttura di un programma con un'istruzione di controllo condizionale, che causa l'esecuzione delle parti di programma chiamate A e B a seconda che la variabile X sia rispettivamente uguale o diversa da zero. In entrambi i casi il programma riprende con una parte comune C.

La fig. 1a) mostra la soluzione del problema in FORTRAN: sono stati evidenziati i due "spaghetti" usati per visualizzare il flusso logico del programma, perché la suddivisione non è evidente a colpo d'occhio: figurarsi poi se le cose si complicano un po'.

La fig. 1b) mostra invece la soluzione in PASCAL: il blocco A è contenuto in un'istruzione *compound*, e costituisce una parentesi nel programma, evidenziata anche da un'opportuna spaziatura; analogamente accade con il blocco B. Il risultato finale è la perfetta leggibilità del flusso: "se X è uguale a zero esegui A, altrimenti esegui B; successivamente esegui C".

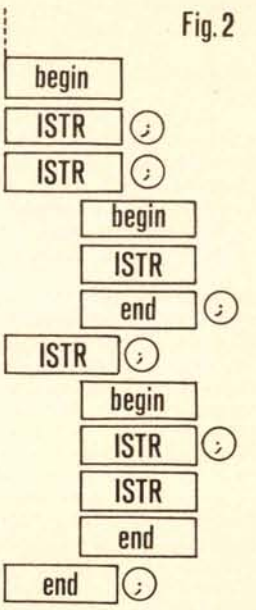
Questo è vero sia dal punto di vista del programmatore, che ha una più chiara visione di quanto sta facendo, sia dal punto di vista del linguaggio in sé: abbiamo già detto che la *compound* è un'unica istruzione, anche se ne può racchiudere parecchie al suo interno; ed il compilatore tratta i blocchi A e B come se fossero istruzioni singole.

Come corollario si può notare che usando la struttura di fig. 1b) abbiamo effettivamente eliminato tutte le istruzioni di salto e conseguentemente gli "spaghetti": è per questo motivo che il prof. Wirth, pur includendo la istruzione GOTO nel linguaggio, ne sconsiglia l'uso.

Tirando le somme, il PASCAL si distingue dagli altri linguaggi per la diversa struttura delle sue istruzioni di controllo, più che per quella di assegnamento, e sarà quindi su di esse che porremo maggiormente l'accento.

Iniziamo dunque la rassegna con l'istruzione

Fig. 2



di assegnamento, utile soprattutto per definire le diverse espressioni aritmetiche e logiche.

L'istruzione di assegnamento

Negli esempi citati nelle scorse puntate abbiamo già incontrato l'istruzione di assegnamento, per poter dare un valore alle variabili che volta per volta si definivano per illustrarne i tipi. L'istruzione, se si ricorda, aveva la seguente struttura:

variabile: = espressione

Possiamo ora precisare che "variabile" deve essere una variabile di tipo semplice: se è di tipo strutturato dovrà essere corredata degli opportuni accessori per renderla semplice.

Un esempio può chiarire il concetto:

```
var PIPPO : record A, B : char;
            I, J : integer;
            ARR : array [1..10] of integer
            end
```

L'istruzione di assegnamento:

PIPPO := .....

non è corretta, perché PIPPO è una variabile strutturata.

È invece corretta la seguente istruzione:

PIPPO.I := 1

perché se PIPPO è un record, PIPPO.I è una variabile intera, e quindi di tipo semplice. Infine

PIPPO.ARR = 1

è errata, perché il campo ARR è a sua volta strutturato ad array: per accedervi bisogna scrivere l'indice:

PIPPO.ARR[1] = 1

Una seconda precisazione può essere che il tipo (semplice) risultante dall'espressione a secondo membro deve essere uguale al tipo (semplice) della variabile che si assegna: è infatti scorretta l'istruzione

PIPPO.I = 'a'

perché PIPPO.I è una variabile intera, e l'espressione al secondo membro è di tipo char.

Come si vede, l'istruzione di assegnamento PASCAL non differisce di molto dall'istruzione di assegnamento degli altri linguaggi: semmai può essere più o meno vasta la gamma di operatori che compaiono nell'espressione.

Le operazioni che si possono eseguire sulle variabili sono:

— operazioni aritmetiche: le quattro fonda-

mentali, l'elevamento a potenza e la divisione in modulo; quest'ultima operazione rappresenta il resto di una divisione, e si indica con l'operatore *mod*: ad esempio 5 *mod* 3 dà come risultato 2, e 10 *mod* 5 dà zero.

Si può eseguire la divisione intera (senza resto) tra numeri reali usando l'operatore *div*: 5.2 *div* 2.0 dà come risultato 2.0.

— operazioni logiche: le tre fondamentali (*and*, *or* e *not*)

— operazioni di relazione: uguale, diverso, maggiore etc. con la stessa scrittura del BASIC. Si noti che queste operazioni possono essere eseguite anche su espressioni logiche, ottenendo ad esempio l'or esclusivo ((A=B) <> (C=D)) o l'implicazione ((A=B) <= (C=D)). Infatti la prima espressione è vera se (A=B) è vera e (C=D) è falsa, oppure viceversa: se entrambe sono vere o entrambe sono false, l'espressione è falsa.

Si noti inoltre che il segno uguale non può creare ambiguità con l'operatore di assegnamento, che è ":=".

Infine si fa presente che esiste una precedenza implicita nelle operazioni, analogamente ai linguaggi più comuni, che può essere modificata mediante l'uso di parentesi.

L'istruzione compound

Abbiamo già visto la struttura dell'istruzione *compound*: praticamente è una copia in miniatura del programma, poiché può contenere dichiarazioni di costante, tipo e variabile che hanno senso soltanto all'interno della *compound* stessa. Nella maggior parte dei casi, però, la *compound* viene usata per strutturare i programmi, e più precisamente per poter scrivere più di una istruzione quando la grammatica ne prescriverebbe una sola. L'esempio di fig. 1b) è indicativo: l'istruzione *if... then... else...* prevede una sola istruzione in entrambe le alternative: se ve ne occorre più di una, si costruisce una *compound*.

Particolare attenzione merita il punto e virgola, il cui uso può apparire a prima vista oscuro: vi sarete già accorti che negli esempi ogni tanto compare e ogni tanto no. Chiariamo la faccenda una volta per tutte precisando che il punto e virgola ha il compito di *separare* le istruzioni all'interno dei blocchi *compound*. La fig. 2) mostra il funzionamento di questo meccanismo: dopo il *begin* e prima dell'*end* non ci sarà mai un punto e virgola, che sarà invece presente ogni volta che due istruzioni appaiono contigue: poiché anche la *compound* è un'istruzione, ecco che il punto e virgola potrà comparire prima del *begin* (anche se non ha molto senso) e soprattutto dopo l'*end*.

Un primo uso dell'istruzione *compound* risiede nelle istruzioni di controllo dei cicli, che nel PASCAL sono più di una e rispondono a particolari requisiti. Analizziamole in dettaglio.

Le istruzioni ripetitive

In FORTRAN e in BASIC abbiamo una istruzione, chiamata rispettivamente DO e FOR, per il controllo dei cicli: come è noto essa serve per eseguire più volte una sequenza di istruzioni racchiusa fra l'istruzione di controllo e un terminatore (in FORTRAN un label, in BASIC l'istruzione NEXT). Il grande svantaggio di queste istruzioni è che il numero di ripetizioni è fissato a priori: al momento di definire il ciclo si pongono i valori iniziale e finale di una variabile di controllo che modifica il suo valore ad ogni passo del ciclo. Questo può essere comodo in certi casi, soprattutto quando si ha a che fare con vettori e matrici; tuttavia anche in questi casi può essere utile un ciclo di lunghezza *variabile*, non prefissata. In FORTRAN e BASIC l'ostacolo viene aggirato ponendo un'istruzione condizionale all'interno del ciclo dimensionato su una lunghezza ritenuta massima: se ad un certo punto la condizione è soddisfatta si esegue un *salto* fuori dal ciclo, senza terminarlo. La fig. 3) illustra questo modo di procedere: come si vede, ricompare la famigerata programmazione a spaghetti.

Il PASCAL prevede l'istruzione di ciclo a lunghezza fissa, che prende il nome di *for* come nel BASIC, ma vi affianca altre due istruzioni che controllano un ciclo a lunghezza variabile, chiamate *while* e *repeat*.

Iniziamo con la prima, che è la più semplice: la sua struttura è:

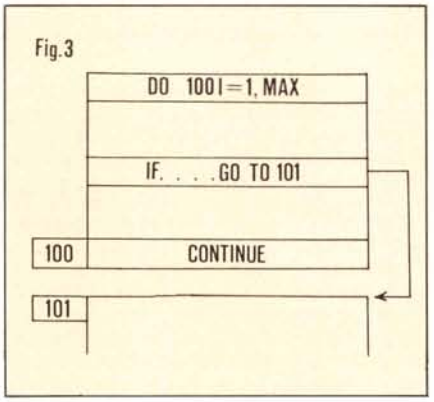
for variabile := inizio *to* fine *do* istruzione

oppure

for variabile := fine *downto* inizio *do* istruzione

Si noti innanzitutto che rispetto all'analoga istruzione BASIC manca il *passo*: la variabile di controllo potrà incrementare il suo valore soltanto di +1 (nel primo caso) o di -1 (nel secondo caso). Si è infatti constatato che i cicli a lunghezza fissa con passo maggiore di 1 sono molto poco usate; se si vuole eseguire un'istruzione di questo tipo, ci sono i cicli a lunghezza variabile.

Un'altra differenza rispetto al BASIC è l'assenza di un *terminatore* del ciclo: ma non ve n'è bisogno, poiché l'istruzione da eseguire è sempre una e una sola, e non vi possono quindi essere ambiguità. A questo punto è chiaro il significato dell'istruzione *compound*: un ciclo di più di un'istruzione sarà racchiuso fra un begin e una end, e sarà quindi visto a tutti gli effetti come un'istruzione sola.



In questo modo non abbiamo soltanto semplificato il lavoro del compilatore, che non deve più tenersi a mente quanto è lungo il ciclo, ma abbiamo scritto il programma in un modo molto più chiaro: se vediamo una *compound* come una coppia di parentesi, possiamo interpretare una *for* in questo modo:

"per I che va da 1 a N esegui (ciclo)"

con l'informazione aggiuntiva:

"(ciclo) = (...;...;...;...)"

Questo è il sugo, l'essenza principale della programmazione strutturata.

Naturalmente, ragionando in questo modo, diventano molto più comprensibili i cicli nidificati: se fra le istruzioni del ciclo c'è un'altra for, si apre un'altra parentesi e si ripete il discorso.

Come esempio scriviamo un programma che esegue la moltiplicazione di due matrici A e B e scrive il risultato in una terza matrice C. N1, N2 e N3 sono costanti e determinano le dimensioni delle matrici.

```
const N1 = ...; N2 = ...; N3 = ...;
var A : array [1..N1,1..N2] of integer;
    B : array [1..N2,1..N3] of integer;
    B : array [1..N1,1..N3] of integer;
    I,J,K : integer;
for I = 1 to N1 do
    for J = 1 to N3 do
      begin
      C[I,J] = 0;
      for K = 1 to N2 do
      [C1,J] = C[I,J] + A[I,K] * B[K,J]
      end;
```

Il programma, dopo la parte dichiarativa, è formato da tre istruzioni for poste una dentro l'altra: la prima esegue un ciclo di una sola istruzione, cioè la seconda *for*; questa invece esegue due istruzioni, ossia l'azzeramento dell'elemento C[I,J] e la terza *for*; quest'ultima infine esegue, come la prima, una sola istruzione che compie la somma iterativa degli elementi di A e B nell'elemento di C preventivamente azzerato.

È chiaro che in FORTRAN o in BASIC una scrittura di questo tipo sarebbe stata molto meno visibile.

Capita invece spesso che un ciclo non abbia una lunghezza prefissata, ad esempio quando occorre scorrere una lista finché non si trova un certo elemento, oppure nell'esecuzione di un'operazione iterativa finché non si verifica una particolare condizione. Come già detto, i linguaggi tradizionali superano quest'ostacolo nel modo illustrato in fig. 3), mentre il PASCAL possiede apposite istruzioni.

L'istruzione *while* esegue un ciclo *finché* una specificata condizione rimane *vera*: non appena la condizione diventa falsa, il ciclo viene interrotto.

La struttura è la seguente:

while condizione *do* istruzione

Come per la for, l'istruzione del ciclo è unica e può essere una *compound*. La condizione deve avere un valore *logico*, ossia "vero" o "falso", e il ciclo viene eseguito finché essa rimane vera: se inizialmente la condizione è già falsa, il ciclo non viene eseguito per nulla.

In questo modo si elimina il salto condizionato di fig. 3 e si salva la strutturazione a blocchi *compound*, compattando il programma e semplificando la sua lettura.

Un esempio di uso efficiente dell'istruzione *while* può essere l'analisi di una struttura di dati a lunghezza variabile. Supponiamo di costruire una lista di numeri interi e di volerli sommare tutti: non sapendo a priori quanto è lunga la lista, un'istruzione *while* è utilissima.

Abbiamo visto nella scorsa puntata come si definisce e si costruisce una lista, e non mi ci voglio soffermare: la fig. 4 illustra la struttura in questione, e la sua definizione PASCAL è riportata nell'esempio.

```
type LISP = ↑LIST;
     LIST = record NUM: integer;
            NEXT: LISP
            end;
var LISTA, PTR: LISP;
    SUM: integer;
(costruzione della lista, vedi fig. 4)
SUM: = 0;
PTR: = LISTA;
while PTR <> 0 do
      begin
      SUM: = SUM + PTR. NUM;
      PTR: = PTR. NEXT
      end;
```

L'operazione si svolge in questo modo: il puntatore PTR si posiziona in capo alla lista, e l'elemento puntato viene sommato all'accumulatore SUM; l'operazione si ripete finché PTR non arriva in fondo alla lista, cosa che accade quando esso assume il valore zero.

L'istruzione while si può quindi interpretare così:

"somma l'elemento e spostati avanti finché PTR è diverso da zero"

Il bello dell'istruzione *while* è che le cose sono già sistemate anche nel malaugurato caso che la lista fosse vuota: in questo caso infatti LISTA è a zero (ossia non punta da nessuna parte), e la condizione della *while* è falsa fin dall'inizio. Il ciclo non viene eseguito neppure una volta e in SUM si trova il valore iniziale, cioè zero.

L'istruzione *repeat*, invece, garantisce l'esecuzione del ciclo da essa controllato almeno una volta, ed ha una struttura leggermente diversa, proprio per evidenziare questo fatto. Vediamola:

repeat istruzione; istruzione;
... *until* condizione

Intanto la *repeat* controlla più di un'istruzione: il PASCAL prevede anche il caso di una programmazione non strutturata. Questo deriva dal fatto che quest'istruzione ha la condizione *in fondo* al ciclo, proprio per evidenziare il fatto che almeno una volta lo esegue, e quindi è presente un *terminatore* del tipo NEXT che rende inutile l'artificio della *compound*.

Il ciclo, al contrario di quanto avviene nella *while* è eseguito finché la condizione rimane *falsa*: quando essa diventa vera, l'esecuzione prosegue.

L'istruzione *repeat* è utilissima quando il ciclo inizializza delle variabili al primo passaggio, e non si può correre il rischio di lasciare qualcosa appeso per aria; altrimenti la sua struttura e il suo impiego non sono molto diversi dalla *while* e in genere si usa l'una o l'altra a seconda che sia più comodo scrivere la condizione in forma "vera" o "falsa".

Come si vede, le istruzioni di controllo dei cicli sono molto più elaborate di quelle dei linguaggi più comuni, e soprattutto permettono di strutturare a blocchi il programma. Ma la potenza del PASCAL riserva un'altra sorpresa nel campo delle istruzioni condizionali, ultimo paragrafo di questa puntata.

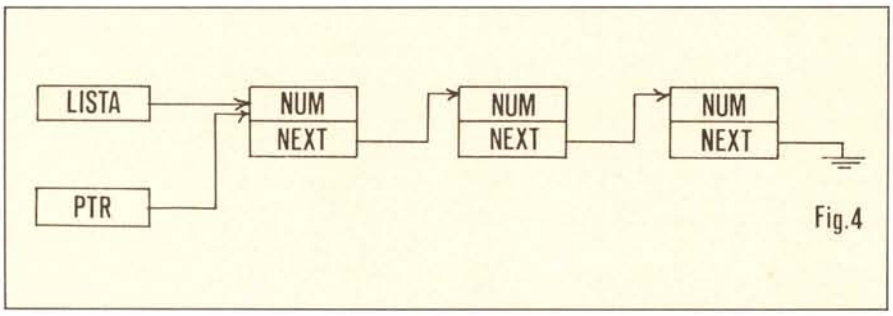


Fig.4

Le istruzioni condizionali

Ogni linguaggio ad alto livello possiede almeno un'istruzione *condizionale*: esse sono infatti le istruzioni di controllo per eccellenza, e tutte le altre (ad esempio quelle che governano i cicli) non sono altro che istruzioni condizionali scritte in maniera particolare.

L'importanza del concetto di *percorso alternativo* in un programma non ha bisogno di essere sottolineata: la possibilità di eseguire diverse parti di programma a seconda del verificarsi o no di una determinata condizione è l'anima stessa della programmazione. Senza di essa non avrebbero senso tutte le tecniche del software, e i programmi si ridurrebbero a brutali sequenze di istruzioni eseguite una dopo l'altra dall'inizio alla fine.

L'istruzione condizionale è dunque presente in tutti i linguaggi, anche negli assemblatori, dove prende il nome di "salto condizionato": esistono istruzioni del tipo: "se l'accumulatore è uguale (o diverso, o maggiore etc.) a zero (o alla tal cella di memoria, etc.) allora salta al tale indirizzo, altrimenti procedi in sequenza." In questo modo il programmatore ha a disposizione un percorso alternativo, e nel corso del programma il controllo potrà seguire l'una o l'altra strada a seconda dello stato in cui si trova in quel momento la variabile che condiziona il passaggio. Il salto condizionato è come uno scambio ferroviario che può essere controllato dal macchinista del treno.

Il FORTRAN, in quanto primo linguaggio ad alto livello, ha mantenuto questa struttura abbastanza rigida del salto condizionato: la sua istruzione condizionale è una IF che offre come alternativa una sola istruzione, altrimenti si procede in sequenza. Va bene che l'istruzione di cui sopra è di solito una GOTO, ma questa struttura di controllo è comunque abbastanza povera.

Con il BASIC andiamo già meglio: l'istruzione IF offre opzionalmente anche la seconda alternativa (IF ... THEN ... ELSE ...), e su questa struttura è costruita l'istruzione *if* del PASCAL. Naturalmente poiché il PASCAL è strutturato a blocchi, è stato fatto il solito trucco della *compound*: cioè il formato dell'istruzione condizionale è il seguente:

if condizione *then* istruzione
[*else* istruzione]

Anche qui, come per la *for* e la *while*, l'istruzione deve essere unica, e l'ostacolo viene superato per mezzo di una *compound*. Penso che l'istruzione condizionale sia il caso più evidente in cui questa strutturazione a blocchi chiarisce la scrittura e il flusso logico del programma: l'esempio di fig. 1 parla chiaro.

Supponiamo però che si voglia scrivere un'istruzione condizionale con *più di due alternative*, ad esempio per eseguire diverse parti del programma a seconda del valore di una variabile: se la variabile vale zero si voglia eseguire una certa routine, se vale uno un'altra, se vale due un'altra ancora, e così via. Il FORTRAN e il BASIC offrono un'istruzione di salto calcolato, che in BASIC ha la struttura "ON variabile GOTO lista di labels", e che naturalmente non esce dalla programmazione a spaghetti: il PASCAL offre invece un'istruzione anche non numerica, visto che il PASCAL le prevede.

Abbiamo già incontrato la *case* la volta scorsa, nella definizione della parte variabile del record: ora vediamo con maggior precisione la sua struttura:

```
case variabile of
v1, v2, ... :
v3, ...     : istruzione;
end
```

Innanzitutto la variabile può essere di qualsiasi tipo *non strutturato* tranne il tipo reale (per un'oggettiva difficoltà nel "centrare" il valore esatto), e l'istruzione deve essere una sola (vale il solito trucco della compound). Le espressioni v1, v2, etc. sono i valori che la variabile può assumere nel corso del programma: per ognuno di queesti valori viene eseguita l'istruzione corrispondente, e soltanto essa.

Un esempio dell'uso dell'istruzione *case* può essere la determinazione del numero di giorni di ogni mese dell'anno. Si voglia infatti costruire un vettore di numeri interi che contenga per ogni mese il numero dei suoi giorni. Terremo conto anche degli anni bisestili e del calendario gregoriano, che salta il bisestile negli anni secolari tranne quelli le cui prime cifre sono divisibili per 4.

```
type MESE = (jan, feb, mar, apr, may, jun, jul,
aug, sep, oct, nov, dec);
var MONTH: MESE;
    NGIOR, YEAR: integer;
    MAXG: array [MESE] of integer;
for MONTH: jan to dec do

begin
case MONTH of
nov, apr, jun, sep: NGIOR: = 30;
jan, mar, may, jul,
aug, oct, dec: NGIOR: = 31;
feb: begin
NGIOR: = 28;
if (YEAR mod 4) = 0 then
if not ((YEAR mod 100) = 0) and
((YEAR div 100) mod 4) = 0) then
NGIOR : = NGIOR + 1

end
end; (case)
MAXG(MONTH): = NGIOR
end; (for)
```

Il programma fissa tramite un'istruzione *for* (da gennaio a dicembre) il numero di giorni del mese nell'array MAXG tramite un'istruzione case. Seguendo la filastrocca "30 giorni ha novembre ...", la prima alternativa mette a posto i mesi con 30 giorni; la seconda quelli con 31; mentre per febbraio le cose sono più difficili a causa dell'anno bisestile: si è usata l'operazione *mod* per stabilire se un anno è divisibile per 4, per 100 o per 400.

In questo ultimo esempio abbiamo mischiato alcune delle istruzioni viste in questa puntata, costruendo un programma strutturato e molto leggibile: il PASCAL aiuta una volta di più a chiarire le idee a chi affronta i misteri della programmazione. Nella prossima puntata affronteremo un altro importantissimo capitolo di questo linguaggio, ossia la definizione e la struttura di funzioni e sottoprogrammi; anch'essi sono utilissimi nel campo della programmazione strutturata, ed occupano anzi un posto di particolare rilievo. Concluderemo infine con qualche accenno alle istruzioni di ingresso e uscita (tipicamente le *read* e *write*), su cui però Wirth non si è sbilanciato molto, lasciando l'iniziativa a chi deve implementare il PASCAL su un particolare sistema operativo.

Pietro Hasenmajer

Nessuno vi dà più potenza di calcolo allo stesso prezzo.

Lit. 269.000 + IVA*

TI-59 è una delle più versatili calcolatrici programmabili che si possano trovare ad un prezzo contenuto (Lit. 269.000 + IVA*).

A differenza di altre calcolatrici programmabili, la TI-59 non richiede la conoscenza di uno speciale linguaggio.

Vi evita la noia dei calcoli ripetitivi, richiedendo un minor numero di impostazioni sulla tastiera e rendendo la soluzione più facile e veloce.

È dotata di un piccolo vano, pronto ad accogliere uno dei 14 "moduli" (Solid State Software™) disponibili, ciascuno dei quali contiene ben 5000 passi di programma pre-registrati. Potrete così scegliere il programma più idoneo per la soluzione dei vostri problemi di progettazione, di fatturazione, di valutazione dei costi, di gestione del budget, ecc., sicuri di utilizzare programmi maneggevoli e affidabili, sperimentati con successo da molti anni.

La sua memoria contiene fino a 100 registri e 960 passi di programmi. Ma non è tutto. Con la TI-59 potrete anche redigere programmi vostri e conservarli registrati su schede magnetiche. Oppure comprare uno dei 16 manuali di programmi (di statistica, dinamica dei fluidi, ecc.) pronti da impostare sulla calcolatrice.

Se poi non avete intenzione di registrare su schede magnetiche, ma vi basta avere una memoria "costante" (Constant Memory™) che conserva gelosamente le vostre informazioni anche a calcolatrice spenta, nella gamma Texas Instruments troverete la TI-58C, la cui memoria contiene fino a 60 registri o 480 passi di programmi ad un prezzo ancor più sorprendente (Lit. 159.000 + IVA*).

Entrambe queste calcolatrici sono in grado di farvi risparmiare tempo, sono portatili e facili da usare.
Completate con l'accessorio PC-100C per la stampa alfa-numerica, vi permettono la trascrizione delle operazioni eseguite e dei risultati (anche sotto forma di grafici).

Perciò, se volete acquistare una calcolatrice programmabile veramente potente, versatile ed aggiornata, progettata e costruita da un'azienda leader nel mondo dell'elettronica, scegliete una TI-59 o una TI-58C della Texas Instruments.

* Prezzo suggerito al pubblico.
TM: marchio registrato Texas Instruments

Il circuito integrato, il microcomputer e il microprocessore sono invenzioni Texas Instruments.

Vi aiutiamo a fare meglio.

TEXAS INSTRUMENTS
SEMICONDUTTORI ITALIA S.p.A.