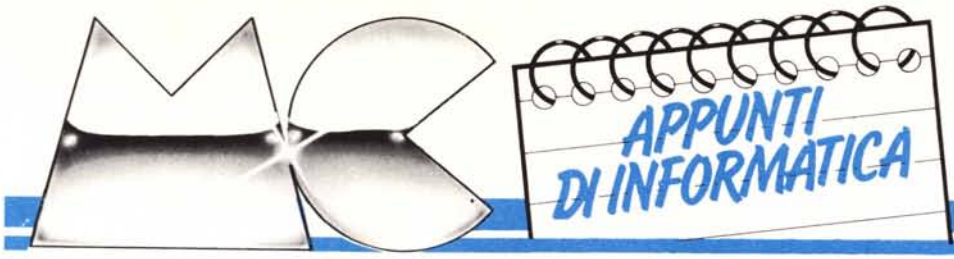

di Andrea de Prisco

Linguaggi, istruzioni, parametri

Dopo essere violentemente precipitati nei più infimi bassifondi di un calcolatore (sino al livello di microprogrammazione) questo mese risaliremo verso gli alti livelli dei moderni linguaggi di programmazione. Tratteremo circa i meccanismi di programmazione offerti da tali linguaggi, tra cui la strutturazione a blocchi, le dichiarazioni, le procedure e le funzioni.

Cenni storici

Come è noto, un tempo esistevano solo le macchine nude e crude, e queste potevano essere istruite soltanto a colpi di «volgare» linguaggio macchina.

Il primo sforzo per aiutare il povero programmatore a non innervosirsi troppo a furia di numeri esadecimali, fu di inventare l'assembler e il macroassembler, col quale era possibile parlare al calcolatore con un linguaggio appena un po' più civile: era perlomeno fatto di parole mnemoniche, con la possibilità di definirsi anche nuove istruzioni a partire da quelle già esistenti (le Macro).

Subito dopo però si sentì l'esigenza di mezzi più potenti per quel che riguarda le applicazioni scientifiche e in particolare i calcoli più complessi delle somme e moltiplicazioni disponibili a livello di CPU.

Servivano nuovi strumenti per trattare facilmente le equazioni, i sistemi e le funzioni matematiche in generale: nacque così il FORTRAN il cui nome sta per FORmula TRANslator che in linguaggio made in Italy vuol dire (per l'appunto) traduttore di formule.

Apparve in questo modo, il primo compilatore della storia: una sorta di programmone che traduceva un linguaggio da un livello più alto ad uno più basso: nel caso del fortran, da fortran a codice eseguibile dalla CPU.

Quasi parallelamente agli ingegneri che protendevano verso le soluzioni meccanizzate dei loro problemi matematici, gli «archivisti» parteggiavano per un linguaggio di programmazione più consono alla archiviazione e l'elaborazione automatica dei dati. Per loro nacque il Cobol: COmmon Busiiners Oriented Language.

Siamo ovviamente ancora agli albori dell'informatica: nonostante gli sforzi compiuti, programmare sia in fortran che in cobol qualcosa di non specificatamente previsto dai due linguaggi risultava tanto difficile quanto poteva esserlo per i calcoli il linguaggio macchina. E da quel momento un po' tutti nel mondo si sbizzarrirono a fare linguaggi di programmazione.

Nacquero linguaggi per trattare agevolmente le stringhe alfanumeriche (Snobol), liste o in generale oggetti non troppo numerici (LISP), processi e comportamenti di oggetti reali (SIMULA) e altro.

L'occhio con cui si guardava la nascita di un nuovo linguaggio di programmazione era comunque la massima comprensibilità anche da chi non avesse scritto il programma. Niente meccanismi per fare trucchetti strani, ma solo strumenti tutti puliti per una programmazione semplice e ordinata. Sparì il concetto di sottoprogramma per fare posto alle procedure e alle funzioni: niente gosub e return ma invocazione tramite il nome della stessa e nient'altro. Dei goto manco a parlarne: sono brutti semanticamente e soprattutto un programma zeppo di goto può solo far disperare chi cerca il bug nel proprio elaborato. Grazie a nuovi costrutti di programmazione come l'IF-THEN-ELSE e il WHILE-DO è stato dimostrato che se ne può fare comodamente a meno.

Algol-like

Il primo linguaggio di programmazione che sfruttò i nuovi costrutti antigoto è stato l'algol 60. Si badi bene

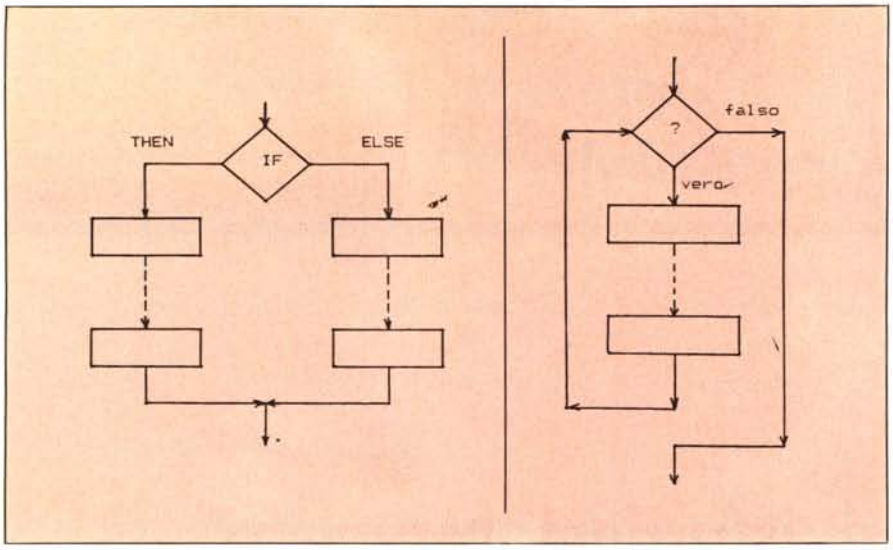

Figura 1 - Diagramma a blocchi dell'IF-THEN-ELSE. - Figura. 2 - Diagramma a blocchi del WHILE-DO.

che per i fissati, il goto era pur sempre disponibile: semplicemente ne era altamente sconsigliato l'uso, anche perché poteva creare non pochi problemi il saltare da un punto all'altro dato che la programmazione algol è strutturata.

A partire da questo, nacquero in seguito altri linguaggi che si ispiravano all'algol 60: tutta la sua dinastia, comprendente algol 68, algo w, pascal, ada e altri, è stata così chiamata algol-like (tipo -algol).

Algol sta per ALGOritmic Language, e il suo nome sta a sottolineare il fatto che il modo di programmare corrisponde praticamente ad una definizione precisa dell'algoritmo che stiamo implementando sul computer.

Un programma Algol-like è strutturato a blocchi (vedremo meglio tra poco) ed ogni blocco è diviso in due parti: la parte dichiarazioni e la parte comandi. Il programma stesso, per intero, è un blocco: altri blocchi saranno in esso nidificati (stile bamboline russe) per descrivere l'algoritmo (o scrivere il programma che è la stessa cosa).

Nella zona dichiarazioni bisogna indicare tutte le variabili usate nel blocco e per ognuna di queste il tipo (intero, reale, stringa, array o altro) e eventualmente i parametri che occorrono, ad esempio il numero di elementi se è un array. Le dichiarazioni, oltre a servire per allocare a tempo di compilazione o a tempo di esecuzione lo spazio necessario alle variabili, permettono durante la stesura di un programma di avere sempre sottomano tutta la lista delle variabili già usate in modo da non usare in modi diversi lo stesso oggetto. Sempreché questo è ciò che desideriamo: infatti la strutturazione a blocchi, unita alla possibilità di fare le dichiarazioni in ogni blocco, permette anche il comportamento opposto: in due punti del programma, lo stesso nome di variabile denota due oggetti diversi.

È arrivato il momento di fare qualche esempio. Dicevamo che un programma algol-like è un blocco formato da due parti: dichiarazioni e comandi:

```
Begin
lista dichiarazioni
lista comandi
End
```

Le due parole chiave begin e end delimitano l'inizio e la fine del blocco. Prendiamo ora l'istruzione IF-THEN: la sua sintassi è:

IF <condizione> THEN <comando o blocco>

Ecco un punto dove possiamo usare un blocco più interno del blocco principale. Quasi tutti i comandi sono fatti in questo modo: se c'è da far fare qualcosa a più di una istruzione, basta racchiuderle tra begin e end in modo da creare un nuovo blocco (si noti che per i blocchi più interni, se non si usano nuove variabili o nuove occorrenze di variabili già esistenti, non sono necessarie le dichiarazioni). L'if di cui sopra, se la condizione è verificata, esegue il comando o i comandi contenuti nel blocco che segue il then; se la condizione non è verificata, non ha effetti (come in Basic).

Ora vedremo la prima delle istruzioni anti goto. Il caso dovrebbe essere ovvio: a seconda di una condizione dobbiamo eseguire un insieme di comandi o un altro, come mostrato nel diagramma a blocchi di figura 1. In un linguaggio di programmazione vecchia maniera, ciò si realizza con almeno un salto, ad esempio in Basic avremo:

```
10 IF A>0 THEN PRINT A: X=X+3: GOTO30
20 B=A/2: A=A+1
30 .....
```

lo stesso programmino in algol-like si scrive:

```
IF A>0 THEN BEGIN
            PRINT A
            X:=X+3
           END
      ELSE BEGIN
            B:=A/2
            A:=A+1
           END
```

appare evidente come nel secondo caso, una volta chiarita la convenzione che begin ed end delimitano un insieme di operazioni da compiere, il programmino algol-like non è altro che la descrizione a parole (sebbene in inglese) del procedimento che volevamo descrivere (libera traduzione: se A è maggiore di zero allora stampa A e ad x associagli il valore di x + 3, altrimenti... ecc. ecc.).

Analogamente per il caso in cui dobbiamo eseguire un insieme di istruzioni finché è vera una condizione (figura 2). Continuiamo con gli esempi basic:

```
10 IF A<0 THEN 50
20 PRINT A
30 A=A-1
40 GOTO 10
50 .....
```

in algol-like scriveremmo un più pulito e più consono al vero significato:

```
WHILE A>0 DO BEGIN
              PRINT A
              A:=A-1
             END
```

Esiste però anche il caso contrario in cui la condizione è posta dopo le istruzioni e si desidera ripeterle fino a quando una condizione non si verifica (figura 3). In basic:

```
10 PRINT A
20 A=A+1
30 IF A<0 THEN 10
```

in algol-like diventa:

```
REPEAT
 PRINT A
 A:=A+1
UNTIL A>0
```

più pieno di significato di così si muore.

Infine, vorremmo mostrarvi come si implementa il caso in cui, a seconda del valore di una certa espressione bisogna eseguire un determinato pezzo di programma (figura 4). A esempio, in basic la situazione potrebbe essere:

```
10 IF A=0 THEN PRINT "OK": GOTO 50
20 IF A=3 THEN PRINT A: A=A-1: GOTO50
30 IF A=5 THEN A=A-2: GOTO 50
40 IF A=2 THEN A=A+5
50 .....
```

scritto in un linguaggio algol-like diventa:

```
CASE A OF
 0: PRINT "OK"
 3: BEGIN
      PRINT A
      A:=A-1
    END
 5: A=A-2
 2: A=A+5
```

Si noti che per il caso 3, dovendo eseguire due comandi è stato necessario racchiuderli in un blocco begin-end.

I blocchi

Finora abbiamo visto e usato i blocchi solo per racchiudere più istruzioni da eseguire al verificarsi di evento. Dicevamo, però, che un blocco è formato anche da una opzionale parte dichiarazioni tramite la quale possiamo definirci nuove variabili locali a quel blocco. Ciò significa essenzialmente due cose: primo al termine del blocco (una volta cioè incontrato il corrispondente end) tutte le variabili li dentro dichiarate vengono deallocate, secondo è possibile creare una nuova istanza di una variabile che non ha nulla a che spartire (tranne il fatto di avere lo stesso nome) con la corrisondente creata in un blocco più esterno. Facciamo un esempio:

```
BEGIN
  VAR X:INTERO
      Y:INTERO
  X:=0
  Y:=100
  WHILE X<>Y DO
  BEGIN
    X:=X+1
    Y:=Y-1
    IF X=10 THEN BEGIN
                   VAR X:INTERO
                   X:=25
                   WHILE X>0 DO BEGIN
                                  Y=Y-1
                                  X=X-1
                                END
                 END
  END
  PRINT "HO FINITO ";X;Y
END
```

All'inizio ci sono le due dichiarazioni per X e Y di tipo intero. Segue la loro inizializzazione rispettivamente a 0 e a 100. Incontriamo a questo punto un comando while che fa ciclare il blocco seguente fino a quando X e Y non diventano uguali. Nel blocco del while, dopo aver incrementato di 1 la X e di egual misura decrementato la Y, se X ha raggiunto il valore 10 si passa al blocco del THEN. Qui troviamo una dichiarazione di nuova istanza per X che viene inizializzata a 25. È importante notare che dentro a tale blocco la X di fuori non è accessibile mentre lo è la Y che non è stata dichiarata nuovamente. La X esterna, che non è scomparsa, è solo disattivata, ritornerà in vita (e col suo valore 10) una volta usciti dal blocco del THEN. Il resto del programma è ovvio.

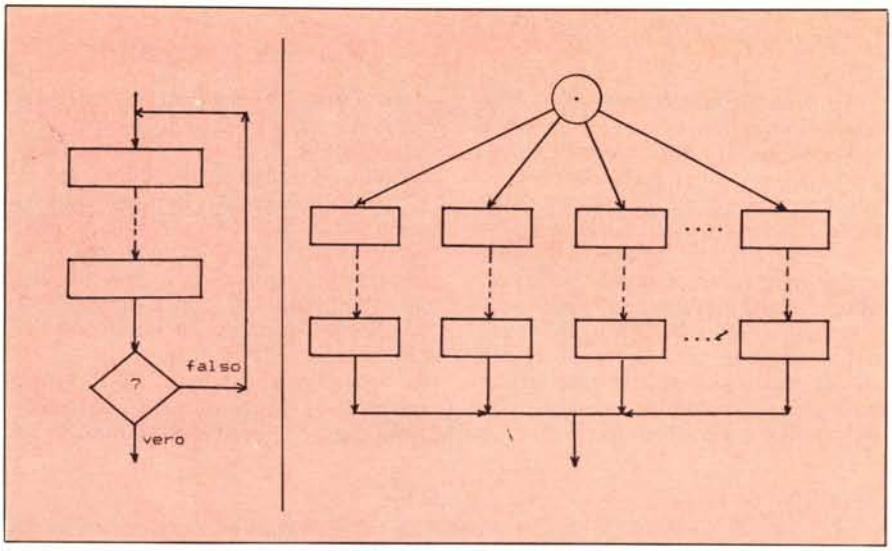

Figura. 3 - Diagramma a blocchi del REPEAT-UNTIL. - Figura 4 - Diagramma a blocchi del CASE-OF.

A questo punto qualcuno si chiederà: «Serve tutto questo?». Sì, come era prevedibile. Noi abbiamo fatto l'esempio di una variabile: se manipolavamo matrici di 10000 elementi la differenza sarebbe stata più sensibile. Immaginiamo che, per un qualsiasi motivo in un punto di un programma ci servono delle nuove matrici per effettuare delle operazioni locali a un preciso momento dell'algoritmo. Ipotizziamo ancora che tale necessità non sempre si presenta, ma è funzione dei dati di ingresso. Dichiarare le matrici ausiliarie nel momento in cui ci servono significa essenzialmente che queste saranno allocate solo se necessario e lo spazio da loro occupato ci verrà restituito alla loro deallocazione automatica all'uscita dal blocco.

Procedure e funzioni

Un altro dei meccanismi di programmazione offerto dai linguaggi algol-like è la definizione delle procedure e delle funzioni. Sono assimilabili a meccanismi di estensione del linguaggio in quanto una procedura può essere vista come un nuovo comando così come per le funzioni definibili. Sia le une che le altre, una volta definite, vengono usate come normali statement e funzioni, come se queste fossero proprie del linguaggio.

Se ad esempio abbiamo la necessità di un comando, che presi due argomenti ne calcola la somma e stampa il risultato, possiamo definirci la seguente procedura:

```
PROCEDURE PIPPO (X,Y:INTERO)
BEGIN
  VAR SOMMA:INTERO
  SOMMA:=X+Y
  PRINT SOMMA
END
```

e da questo momento possiamo usare PIPPO a nostro piacimento, ad esempio PIPPO (4,9), PIPPO (28*A,J/2) o similmente.

X e Y della procedura sono detti parametri formali e effettivamente stanno lì pro forma. Nel senso che servono solo per associare i parametri di ingresso (detti attuali) a qualcosa (i nomi X e Y) che saranno usati all'interno della procedura. Quindi tanto questi, quanto la variabile SOMMA dichiarata all'interno della procedura, una volta terminata l'esecuzione di questa, non esisteranno più. Esistono cioè solo nell'ambiente locale della procedura, che dal canto suo si comporta come un blocco essendo costituita da un blocco.

Analogamente possiamo definirci una funzione che potremo comodamente usare nelle nostre espressioni, come se fosse una funzione predefinita ossia offerta direttamente dal linguaggio. A differenza delle procedure, le funzioni restituiscono un valore e quindi bisogna dichiarare oltre al tipo dei parametri anche il tipo del risultato. Facciamo un esempio: immaginiamo che il nostro linguaggio non disponga della funzione tangente di un angolo, ma solo delle funzioni seno e coseno. Come è noto, la tangente di un angolo è uguale al seno diviso il coseno dell'angolo in questione. Noi,dovendo usare spesso la tangente non vogliamo ricorrere a scrivere ogni volta seno su coseno. Definiamo la nostra funzione così:

```
FUNCTION TAN (X:REAL) : REAL
BEGIN
  TAN:=SIN(X)/COS(X)
END
```

e potremo usare TAN dove e come vogliamo, naturalmente nel giusto contesto: come espressione che restituisce un valore: es. A: = TAN(30), A: = TAN(30) + SIN (45) e così via.

Routine e coroutine

Tutti sanno cos'è una subroutine, come si chiama e come, da questa, si torna al programma chiamante. La prima operazione, generalmente gosub o call ha come parametro il nome o l'indirizzo dove saltare. La seconda operazione, return o similare, si usa generalmente senza specificare altro. Naturalmente le subroutine possono essere tra loro nidificate, nel senso che una subroutine, allo stesso modo del programma principale può a sua volta invocare altre subroutine e via dicendo.

In figura A è mostrato un programma che con CALL «A» invoca la routine A, la quale a sua volta con CALL «B» invoca B. Incontrato il return di B il controllo passa nuovamente a A per poi passare al programma principale dopo il return di questa. In ognuno di questi quattro momenti, indicati con i numeri 1-4 sempre in figura A, anche se non direttamente visibile, è in gioco un altro oggetto, di primaria importanza per il buon funzionamento del meccanismo dei sottoprogramma: lo Stack dei punti di ritorno.

Come funziona una struttura a Stack, ne abbiamo già ampiamente parlato altre volte, quindi passiamo oltre. In tale stack, sono posti come dice il suo nome i punti di ritorno delle subroutine: dove il controllo deve tornare una volta incontrato il return. Tale punto sarà ovviamente l'istruzione seguente il CALL attivante. In figura B è mostrato lo stack dei punti di ritorno nei 4 momenti salienti di cui sopra: in 1 è posto sullo stack 1001 che è l'indirizzo successivo al CALL «A» del programma principale; in 2 succede esattamente la stessa cosa per il sottoprogramma A che chiama B; in 3, B esegue il return e quindi dallo stack si preleva il 2001 che serve per tornare ad A. Analogamente in 4 per tornare al programma principale.

In figura C è mostrato il funzionamento delle coroutine. Queste, in numero sempre maggiore o uguale a 2 hanno un funzionamento globalmente (le coroutine tutte insieme) simile ad una subroutine ma prese singolarmente sono una cosa ben diversa. Globalmente simili vuol dire che il programma principale alla stessa stregua di un sottoprogramma esegue un CALL per far partire la prima coroutine. Allo stesso modo, il primo return che si incontra fa tornare il controllo al programma principale. Le coroutine invece, prese singolarmente si invocano l'un l'altra tramite l'istruzione RESUME che ha la particolarità (a differenza della CALL) di far ripartire la coroutine dal punto in cui precedentemente s'era fermata inseguito a un suo RESUME. Per convenzione, una coroutine che non è mai partita, se «riesumata» inizia dalla prima istruzione.

Tornando alla figura C, la sequenza degli eventi mostrati, in ordine cronologico è la seguente: il programma principale invoca A; A, dopo qualche sua istruzione riesuma B (che parte dall'inizio); dopo un po' B riesuma A che riprende dal resume B precedente. E così via fino al RETURN di B che usando anch'esso lo stack dei punti di ritorno fa tornare al programma principale.

Più semplicemente dei sottoprogrammi, per l'implementazione è sufficiente associare ad ogni coroutine una cella di memoria, inizializzata all'indirizzo di partenza della coroutine stessa, nel quale è salvato di volta in volta il proprio indirizzo di riesumazione. Tutto qui.

MK

Figura A: Chiamate nidificate di subroutine

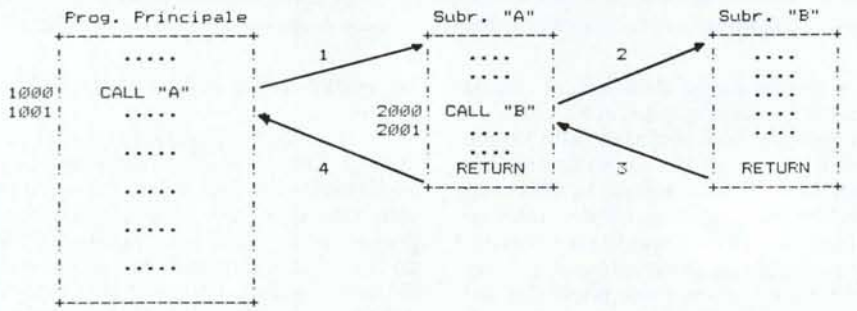

Figura B: Stack dei punti di ritorno in seguito alle operazioni a-b-c-d

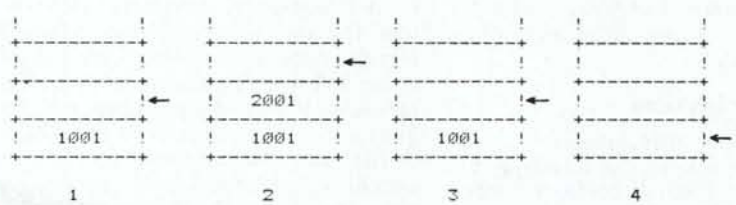

Figura C: Chiamata, attivazione e riattivazione di coroutine