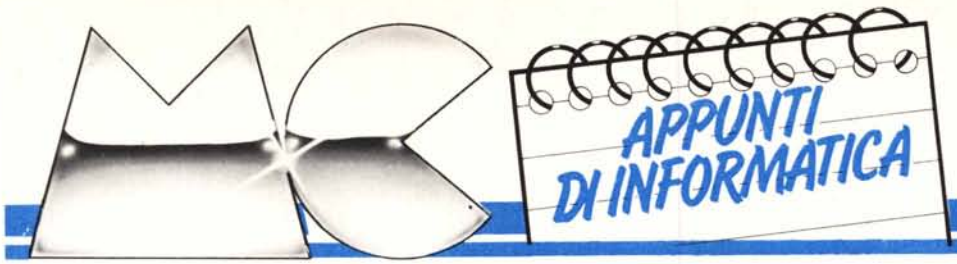

di Andrea de Prisco

Tipi di dato, operazioni, ricorsione

Questa puntata di «Appunti» è interamente dedicata ai moderni linguaggi di programmazione e agli strumenti di programmazione messi a disposizione dell'utente. Spiccano tra questi, oltre ai costrutti condizionali e di iterazione visti lo scorso mese, la possibilità di definire nuovi tipi di dato e nuove operazioni grazie alle quali scrivere oggi un programma risulta essere assai meno laborioso di una volta. Oltre a ciò, un programma scritto con queste tecniche moderne, è facilmente comprensibile anche da chi non ha steso il programma: è il classico caso in cui il programmatore X deve effettuare delle modifiche al programma scritto da Y alcuni anni prima, senza l'appoggio (morale e materiale) dell'autore stesso.

Dato e tipo di dato

Se noi scriviamo 3, abbiamo scritto un numero. Se però andiamo a guardare in una tabella Ascii, alla posizione 51 troviamo il carattere 3, che differisce da una A o una B solo dalla posizione in tabella. Tra l'altro, se noi prendiamo il numero 3 e a questo gli sommiamo il numero 2, otteniamo il numero 5 (sottolineo numero). Questo perché la somma, definita come operazione su numeri, restituisce come risultato un numero.

Riferendoci al BASIC, sappiamo che esiste anche la somma tra due stringhe (più precisamente la concatenazione) e nel caso nostro avremo che «3» + «2» fa «32». In questo caso, infatti, la somma tra stringhe definita su oggetti di questo tipo (eccolo qua) restituisce ancora una stringa.

Detto ciò, che tra l'altro dovrebbe essere arcinoto ai più, vediamo come questo discorso viene esteso nella direzione della definibilità da parte dell'utente.

In un linguaggio di programmazione che si rispetti, deve essere data la possibilità di definire nuovi tipi di dato, a partire da quelli già esistenti. Devono cioè esistere determinati meccanismi atti a descrivere nuovi tipi che nel programma intendiamo usare. Naturalmente i nuovi tipi di dato non devono causarci problemi: devono essere facilmente usabili come i tipi già predefiniti (numeri, stringhe, booleani ecc.).

Tanto per fare subito un esempio, se ci serve il tipo di dato colore, basterà dichiararlo per enumerazione con:

```
type Colore = (rosso, verde, blu, bianco, nero)
```

subito dopo possiamo definire una variabile di questo tipo:

```
var X: Colore
```

alla quale associare uno dei valori dichiarati prima.

Ma non basta. Possiamo definire una matrice che ha per indice i colori e i singoli elementi di tipo intero:

```
var A: array [Colore, Colore] of integer
```

e fare assegnamenti del tipo:

```
A [rosso, blu] = 30
```

o roba simile. Analogamente possiamo usare oggetti di nuovo tipo anche in un costrutto iterativo come il FOR, ad esempio:

```
for X = rosso to nero
```

oppure:

```
for X = bianco downto verde
```

nel primo caso alla variabile X saranno assegnati i valori, iterazione dopo iterazione, rosso, verde, blu, bianco, nero, nel secondo caso dato che downto sta per (si fa per dire) step —1 la variabile X conterrà i valori bianco, blu, verde.

Il record

Un altro costruttore di tipo, generalmente presente in un linguaggio di programmazione, è il record che permette di costruire tipi di dato strutturati. Un esempio abbastanza banale potrebbe essere la dichiarazione di tipo mostrata in figura 1, con la quale intendiamo usare variabili formate da due campi di tipo stringa: nome e cognome. Per quanto riguarda il loro uso, dopo aver dichiarato una variabile di questo tipo

```
var Tizio: = persona
```

per accedere ai vari campi in struttura scriveremo:

```
Tizio.nome: = «giuseppe»
Tizio.Cognome: = «paoletti»
```

analogamente per leggere i relativi campi:

```
C: = Tizio.nome
```

posto che C sia stata dichiarata di tipo stringa.

Un record a sua volta può contenere altri record così come altri oggetti di nuovo tipo purché precedentemente definiti. Un esempio è mostrato in figura 2 dove è stato dichiarato il tipo di dato automobile, record con campi «modello» di tipo stringa, «proprietario» di tipo persona (quindi un record) e «colore» di tipo Colore che abbiamo definito a inizio articolo. Se a questo punto dichiariamo:

```
var MiaAuto: Automobile
```

possiamo aggiornare i vari campi con la sequenza di istruzioni mostrate in figura 3.

Inutile dirvi che possiamo definire array di record così come record con campi di tipo array. Insomma, se un

linguaggio di programmazione è fatto bene non deve limitarci in alcun modo circa scelte di tale fatta. E basta!

Nuove operazioni

Definiti i nuovi tipi di dato, possiamo definire delle operazioni su questi, tramite il meccanismo della procedure e funzioni visto lo scorso mese. Tutto ciò, sempre per rendere la programmazione più chiara e più pulita possibile, nel rispetto (ci ripetiamo) di chi un giorno dovrà eventualmente raccapezzarsi tra le linee dei nostri elaborati.

Se credete che quanto state leggendo siano solo fandonie facciamo un piccolo gioco: immaginiamo di trovare il listato di un programma e voler capire cosa questo faccia. Dando una scorsa veloce vediamo che sono usate due matrici, A e B e che in queste, diciamo, che sono immessi dei numeri compresi tra 0 e 5. Qualche decina di salti a destra e a sinistra, i soliti FOR con indice I o J e degli orribili GOSUB 1000, GOSUB 2000 e GOSUB 3000 ecc. ecc.

Cinque secondi di tempo per capire cosa potrebbe fare questo programma.

Non per sottovalutare qualcuno, ma non credo che sia possibile indovinare. Giriamo pagina e troviamo lo stesso programma scritto in Pascal, da qualche purista della programmazione. Al posto delle matrici A e B troveremo i nomi MioCampo e TuoCampo, ambedue con un indice numerico e l'altro alfabetico compreso (ad esempio) tra A e H. Notiamo poi che le caselle di tali matrici sono di tipo nave e il tipo nave è definito come l'insieme degli oggetti incrociatore, cacciatorpediniere, portaerei, corazzata, acqua. Guardando ancora il listato leggiamo nomi di funzioni come PosizionaNavi, MioColpo, TuoColpo, IntercettaNave e altro.

Cosa fa questo programma?

Scommetto che gioca a battaglia navale.

Senza contare che un programma scritto bene non occupa necessariamente più spazio in memoria di uno scritto male, tanto più che una volta compilato tutti i nomi spariscono e lo spazio occupato riguarda effettivamente la cella x o la cella y indipendentemente se prima della compilazione una variabile si chiamava «P», «Pippo» o «IlMioNomeÈPippo».

Fatta questa piccola dissertazione, andiamo avanti con la nostra scaletta. Dunque è possibile definire nuove operazioni, sui nuovi tipi di dato. Lo scorso mese avevamo già visto qualche esempio di definizione di funzione e procedura sui tipi di dato standard. Se ad esempio ci serve la funzione fattoriale che come è noto è dai naturali ai naturali, possiamo definirla scrivendo le linee mostrate in figura 4. Commentiamola brevemente. La prima linea serve per definire il nome della funzione, nome e tipo dei suoi argomenti (tutto compreso tra le due parentesi), infine il tipo della funzione ossia di che tipo sarà il risultato: nel nostro caso intero. Seguono le dichiarazioni di due variabili locali alla procedura anche queste di tipo intero e l'inizializzazione di K al valore di 1.

Il FOR che segue calcola il fattoriale del numero dato in ingresso che, lo ricordiamo, al momento della chiamata della funzione è associato al nome X, parametro in ingresso di questa.

Infine il valore K che a questo punto contiene il fattoriale di X, è associato alla funzione stessa che nell'espressione nella quale è avvenuta la chiamata restituirà il valore calcolato. Ovvero, se da qualche parte avessimo scritto:

A: = FATTORIALE (3) + 5

il valore 6 (3 fattoriale) nella valutazione dell'espressione a destra dell'assegnamento sarà sostituito alla chiamata FATTORIALE (3) appena tornati da questa.

Detto questo, l'estensione al caso dei tipi di dato definibili dall'utente è banale. Possiamo cioè scrivere funzioni da Colori ad Automobili, da Persone ad Ortaggi o come meglio crediamo. Basta solo usare i tipi nel modo giusto e il gioco è fatto. Facciamo un esempio: immaginiamo di avere un array di 100 Automobili, il tipo mostrato in figura 2. La dichiarazione di tale array sarà data nel seguente modo:

var ListaAuto: array [1..100] of Automobile

vogliamo una funzione, che dato il colore ci restituisca la prima persona che nella nostra lista ha un'auto di quel colore. Per semplicità supponiamo che tale persona (o meglio: tale auto di questo colore) esista sempre ovvero nella nostra lista ci sono auto per tutti i colori.

Una possibile soluzione è mostrata in figura 5: abbiamo chiamato questa funzione PersonaColoreAuto la quale, come detto, riceve in ingresso un parametro di tipo colore e restituisce la persona trovata. Ad esempio una chiamata di tale funzione potrebbe avvenire banalmente così:

Persona1: = PersonaColoreAuto(blu)

che corrisponde ad associare alla variabile Persona1, che precedentemente

Figura 1:

```
type Persona : record
                 nome    : String
                 cognome : String
               end
```

Figura 2:

```
type Automobile : record
                    modello     : String
                    proprietario: Persona
                    colore      : Colore
                  end
```

Figura 3:

```
MiaAuto.modello := "Y10"
MiaAuto.proprietario.nome := "andrea"
MiaAuto.proprietario.cognome := "de prisco"
MiaAuto.colore := blu
```

Figura 4:

```
function Fattoriale (X:integer):integer
   var I,K:integer
   K:=1
   for I=1 to X do K := K*I
   Fattoriale := K
end
```

Figura 5:

```
function PersonaColoreAuto (C:Colore) : Persona
   var I:integer
   I:=1
   while I<=100 and ListaAuto(I).colore <> C do I:=I+1
   PersonaColoreAuto:=ListaAuto(I).proprietario
end
```

Figura 6:

```
function Successore (n:integer):integer
   if n=0 then Successore:=1
      else Successore:=1+Successore(n-1)
```

Figura 7:

```
function Fattoriale (n:integer):integer
   if n=0 then Fattoriale := 1
      else Fattoriale := n*Fattoriale(n-1)
```

deve essere stata dichiarata di tipo persona, il primo proprietario nella nostra lista che possiede un auto di colore blu.

Per quanto riguarda il listato di figura 5 non dovrebbero esserci problemi di comprensione, specialmente una volta chiarito il fatto che se ListaAuto, è un array di automobili, preso I compreso tra 1 e 100, ListaAuto(I) sarà di tipo Automobile (figura 2) quindi per accedere ai vari campi di questo elemento (che è a tutti gli effetti un record) basta scrivere «ListaAuto(I). colore» «ListaAuto(I). modello» oppure «ListaAuto(I).proprietario».

La ricorsione

Per terminare questo ciclo di articoli sulla programmazione a un livello un tantino più alto del Basic, non potevamo non parlare di quell'altro mondo tanto affascinante quanto sconosciuto della ricorsione.

Affascinante per il fatto che permette di risolvere problemi di natura ricorsiva con davvero poche linee di listato, sconosciuto per l'assurdo motivo che nel Basic non è contemplato. L'assurdità, si badi bene, non sta nel fatto che il Basic non ammette ricorsione, ma piuttosto nel fatto che tutto quello che non è specificatamente previsto da questo (chiamiamolo) linguaggio è sconosciuto.

Ovvero se chi avesse pensato al Basic l'avesse fatto con le idee un po' più chiare, il livello di informatizzazione di massa sarebbe ben più alto. Livello nel senso qualitativo. Punto.

Dicevamo che con la ricorsione si trattano i problemi di natura ricorsiva. Un problema di natura ricorsiva è detto tale se la sua soluzione può essere espressa in termini del problema stesso. Esattamente come un gatto che rincorre la sua coda.

Detto in questi termini sembrerebbe una stravaganza matematica insolubile anche se, come vedremo, stiamo tutt'altro che sull'inutile. Facciamo un primo esempio: supponiamo di avere un linguaggio di programmazione che ammette ricorsione, ma per quanto riguarda le addizioni, riesce solo a sommare unità, un numero qualunque di volte. Nella fattispecie non è in grado ad esempio di eseguire 2+3, ma è in grado di eseguire 1+1+1+1+1. Immaginiamo di dover scrivere, con questi mezzi a disposizione, una funzione che dato un numero intero maggiore o uguale a zero restituisce il suo successore. Il problema è di natura ricorsiva in quanto una possibile soluzione potrebbe essere la seguente: «il successore di un numero n si calcola così: se n è uguale a 0 allora il suo successore è 1 altrimenti sarà uguale alla somma di 1 e del successore di n—1». La ricorsione sta proprio nel fatto che nella soluzione si fa nuovamente riferimento al problema stesso, il calcolo del successore (anche se di un numero più piccolo).

Vediamo almeno se funziona, proviamo a calcolare il successore di 2.

Il successore di 2 è uguale a 1 più il successore di 1, il quale è uguale a 1 più il successore di 0 che a sua volta è 1. In tutto 1+1+1, che la nostra macchina è in grado di eseguire e darà come risultato 3. In figura 6 è mostrato il programmino pascal—like corrispondente alla funzione ricorsiva successore. Si noti come sia di fatto la traduzione del procedimento a parole descritto prima e la ricorsione la troviamo nel fatto di vedere dentro alla definizione della funzione uno chiamata alla funzione stessa. Non occorre ricordare che le varie istanze del parametro n, chiamata dopo chiamata (ricorsiva e non) sono tutte diverse: se infatti al primo «giro» n vale 3, al momento della chiamata dopo l'else, dato che passiamo come parametro n—1 pari a 2, al secondo «giro» l'enne dell'if vale 2 e così via, fino a quando (al «giro» giusto) varrà 0.

Facciamo un esempio un tantino più utile: il calcolo del fattoriale di un numero. La versione non ricorsiva l'abbiamo già vista in figura 4: tutti infatti sanno che il fattoriale di n è uguale al prodotto dei primi n numeri (inoltre fattoriale di 0 è posto uguale a 1) e il listato di figura 4 fa appunto questo. Esiste però un'altra definizione di fattoriale tra l'altro anche più corretta:

n! 1 se n0 — n(n—1)! altrimenti

indovinate un po' come si traduce tale algoritmo nel programma ricorsivo corrispondente. Basta tradurre parola per parola, come mostrato in figura 7.

Tolta l'intestazione comune sia al listato di figura 4 che a quello di figura 7 restano 4 linee nel primo caso, 2 nel secondo, pari a un risparmio del 50%. Inoltre, nel primo caso abbiamo dovuto usare due variabili locali che nel secondo caso non servono.

Potremmo continuare col calcolo di un elemento della successione di Fibonacci (vedi riquadro), ricerche in strutture ad albero, ricerche binarie in strutture lineari, problemi di sort (ordinamento), manipolazione di elementi collegati a lista. Tutto diventa enormemente più facile se inquadrato nella giusta ottica ricorsiva. È un vero peccato... MC

Il problema dei conigli

A proposito di relazioni ricorsive, pare che la più antica e famosa di queste (cfr. Fabrizio Luccio, «La struttura degli algoritmi», Boringhieri 1982, pagg. 73 e seguenti) fu posta nel secolo tredicesimo dal matematico pisano Leonardo di Bonaccio da Pisa, più noto forse come Fibonacci. Spicca tra le sue opere la famosa successione di Fibonacci, a suo tempo posta in relazione a un problema ideale di riproduzione di conigli, e tutt'oggi di grande importanza nella matematica discreta mostrando inaspettate relazioni con le altre fondamentali successioni numeriche e con le frazioni continue.

Il problema è posto in questi termini: supponiamo per ipotesi che una coppia di conigli ogni mese produce una nuova coppia di conigli. Dal mese successivo alla loro nascita, diventati adulti, anche i nuovi nati sono in grado di riprodurre. Considerando un periodo pari a un mese per la gestazione della femmina si vuole conoscere a quanti conigli assomma l'allevamento dopo n mesi, partendo al mese 1 con una coppia di conigli neonati.

Al mese 1, come detto, abbiamo una sola coppia. Essendo questa neonata, soltanto al mese 2 questa coppia sarà adulta e sarà in grado di procreare, quindi anche al mese 2 abbiamo una sola coppia. Trascorso un altro mese, la femmina partorisce una nuova coppia e quindi al mese 3 abbiamo in tutto 2 coppie (attenzione: una è neonata). Al mese 4 solo la coppia adulta «sforna» un'altra coppia mentre la coppia più giovane è in grado di riprodurre, totale 3 coppie. Finalmente, al mese 5 nascono due nuove coppie, una dalla coppia più anziana, l'altra dalla coppia giovane ormai adulta. E così via.

In generale, il problema può essere risolto facilmente in questi termini: al mese n abbiamo (ovviamente) tutte le coppie presenti al mese n—1 più le coppie neonate. Le coppie neonate sono pari a tutte le coppie adulte (ovvero con più di due mesi, uno per crescere e uno per la gestazione): nasceranno tante coppie quante ce n'erano al mese n—2. Quindi la soluzione è:

F(n) = F(n-1) + F(n-2)

palesemente ricorsiva. I casi iniziali, li ricordiamo, sono tali che sia al primo mese che al secondo abbiamo una sola coppia quindi la successione ha questa «forma»: 1, 1, 2, 3, 5, 8, 13, 21... ecc.

A questo punto, l'ovvia soluzione al problema tramite funzione ricorsiva Pascal-like:

```
Function Fibonacci(n:integer):integer
   if n<=2 then Fibonacci := 1
      else Fibonacci := Fibonacci(n-1)+Fibonacci(n-2)
```

Provate a scriverla non ricorsiva, così, per assaporare la differenza.