Seconda parte

I tipi strutturati: matrici, files, insiemi

Nella prima parte di questa sommaria esposizione del PASCAL abbiamo gettato le fondamenta del linguaggio: un po' di storia, tanto per capire come si sia arrivati a pensare ad un linguaggio con queste caratteristiche, la struttura base di un programma, e una rapida esposizione delle dichiarazioni più semplici: label, costanti e variabili, fino alle più elementari dichiarazioni di tipo. *Fin qui il PASCAL non presentava caratteristiche particolarmente spettacolari rispetto ai linguaggi più noti — a parte il tipo* scalar *che permette di definire ad uso dell'utente il "dominio" di una variabile.*

Ricordiamoci però che il PASCAL è un linguaggio altamente strutturato, sia nelle istruzioni che nei dati, ragione per cui sarà in questa puntata — dedicata ai tipi strutturati — che la enorme potenza offerta dal linguaggio di Wirth inizierà a mostrarsi in tutta la sua pienezza.

Allacciarsi le cinture, dunque, e non fumare: stavolta si parte sul serio.

La strutturazione di dati

Le variabili — come siamo abituati a considerarle — sono in genere a *un solo valore*, rappresentano cioè *un solo numero*, la cui forma è specificata dal *tipo* della variabile: numero intero per una variabile intera, numero reale per una variabile reale, etc.... Anche nei calcoli matematici fatti a tavolino (se si escludono i numeri complessi che sono composti da una parte reale e una immaginaria e quindi vengono solitamente rappresentati con *due* numeri) non esistono operazioni che si compiano su variabili a più valori.

La necessità di creare variabili che potessero rappresentare più di un numero si fece sentire, nei primi tempi della programmazione, quando si pensò di impostare sul calcolatore operazioni di calcolo matriciale. Pensateci un attimo: la risoluzione di un sistema di grandi dimensioni scritto in forma matriciale tramite il vettore delle incognite, la matrice dei coefficienti e il vettore dei termini noti, richiedeva calcoli lunghi, ripetitivi e noiosi, l'ideale quindi per farli risolvere alla macchina. Ma in questo caso occorreva per forza di cose definire una variabile sola mediante la quale si potessero indirizzare, a controllo di programma, più celle di memoria: creare cioè strutture come quelle di fig. 1 in modo da poter puntare direttamente all'*elemento* del vettore e poter *modificare* da programma questo puntamento; permettere insomma al programma di gestire un meccanismo del tipo V (I) con V = vettore e I variabile.

Si giunse così a definire, nei linguaggi come il FORTRAN, delle variabili di tipo *vettoriale* e *matriciale*, che venivano tradotte dai compilatori in strutture software come quelle della fig. 1a.

Un vettore (fig. 1a) era definito per mezzo di un puntatore ed un indice, mentre per una matrice (fig. 1b) occorrevano un puntatore e due indici: per mezzo di somme e moltiplicazioni si puntava un ben preciso elemento della struttura a seconda del contenuto degli indici.

Variabili definite in questo modo non sono più considerate di tipo intero o reale, ma di tipo *vettoriale* o *matriciale*: insomma, un tipo *strutturato*. Restano da stabilire due cose:

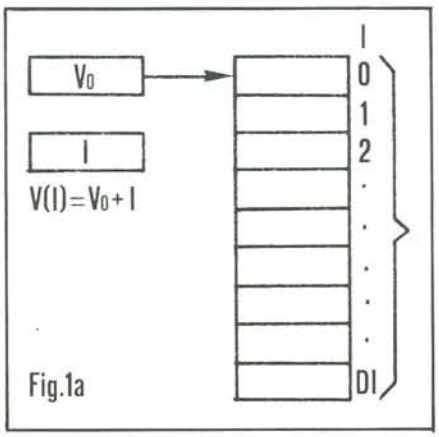


Fig.1a

— le dimensioni della struttura
— il tipo del *singolo elemento* della medesima.

Il FORTRAN (e il BASIC) sono molto vincolanti sotto entrambi questi aspetti:
— le dimensioni di un vettore (o matrice) sono date da un numero intero da 0 a Dk, con Dk determinato all'atto della dichiarazione.

Esempio:
DIM V (20), M (30,40)

Se scriviamo V (I) o M (J,K) sappiamo che I, J e K devono essere variabili intere di valore da 0 a rispettivamente 20, 30 e 40.
— Il tipo del singolo elemento della struttura può soltanto essere uno dei *tipi elementari* (intero, reale, al limite stringa nel caso del BASIC): non può, ad esempio, essere a sua volta un tipo strutturato.
— La matrice (mono o bidimensionale) è l'unico tipo strutturato ammesso.

Il PASCAL supera sotto tutti gli aspetti le prestazioni degli altri linguaggi, soprattutto perché non pone alcun vincolo sul tipo dell'elemento della struttura: possiamo così avere strutture composte da strutture che sono a loro volta formate da strutture e così via, costruendo architetture di dati che possono essere anche molto complesse.

In più il PASCAL offre ben altro che la semplice e rigidissima matrice: i dati possono essere strutturati con diverse modalità, ed è così che la generica dichiarazione di tipo:
type tipo = f (tipi);
citata nella scorsa puntata, acquista tutto il suo significato.

In realtà i tipi strutturati possono essere visti come *funzioni* dei tipi più semplici (o di altri tipi strutturati: è valido anche qui il concetto di "funzione composta").

Vediamo dunque uno per uno questi tipi strutturati, cercando anche di scoprire le migliori possibilità di impiego per ciascuno di essi.

Il tipo array

Il PASCAL non poteva certamente lasciar fuori dal suo insieme di definizioni le strutture a vettore e a matrice, se non altro per compatibilità con i linguaggi preesistenti — oltre che per un'effettiva utilità della struttura in un sacco di casi pratici quali, appunto, i calcoli matriciali.

Però, già che c'era, Wirth ha voluto ampliare le possibilità di questa struttura, generalizzandola e rendendola assolutamente universale.

Il tipo *array* crea dunque una struttura *rigida* formata da elementi di *tipo qualsiasi*, indirizzabili per mezzo di un indice anch'esso di *tipo qualsiasi, purché non strutturato*.

Lo scheletro della definizione è:
type = *array* [tipo 1] *of* tipo 2;
dove "tipo 1" è l'insieme di variabilità dell'indice e "tipo 2" del singolo elemento della struttura: "tipo 1" non deve essere strutturato, mentre per "tipo 2" non c'è limitazione di sorta.

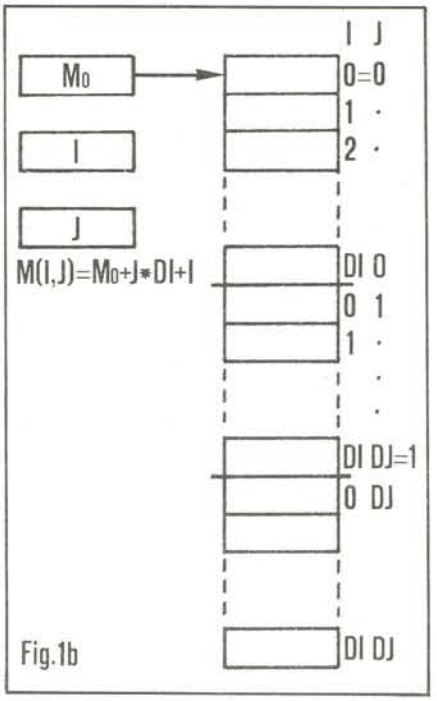


Fig.1b

Così a prima vista l'array potrebbe sembrare soltanto monodimensionale; tuttavia si tenga presente che il tipo "tipo 2" dell'elemento può essere qualsiasi; anche strutturato, e quindi può essere a sua volta un'array:

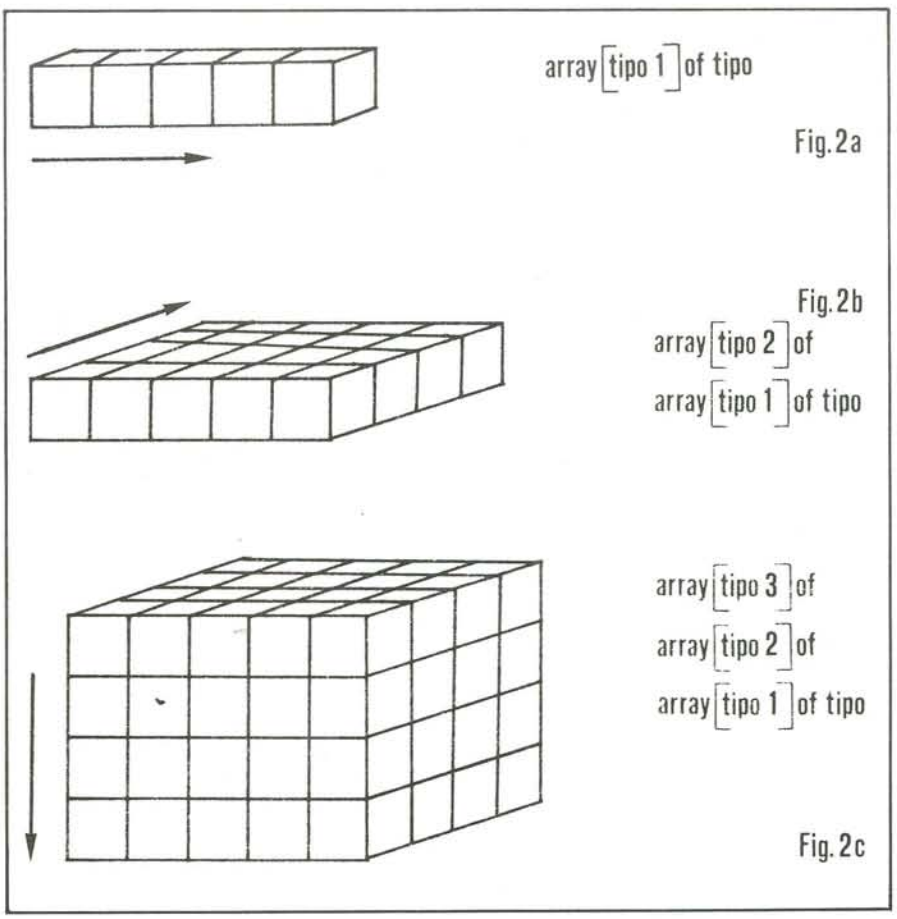
type matrice = *array* [tipo 2] *of*
array [tipo 1] *of* tipo;
come in fig. 2.

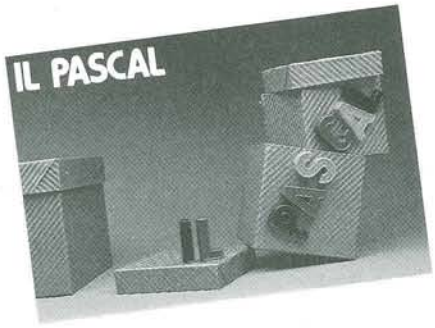
In questo modo, si possono dichiarare matrici a più di due dimensioni; per di più si è pensato, poiché questo caso è abbastanza frequente, di *compattare* le nidificazioni di *array* in una sola con più indici: così la struttura di fig. 2c può essere descritta con la seguente dichiarazione:
type mat3d = *array* [tipo 3, tipo 2, tipo 1] *of* tipo;
ottenendo fra l'altro una scrittura simile a quella dei linguaggi tradizionali.

Le variabili che servono da indici sono in genere di tipo *subrange*, e se in particolare definiscono un sottoinsieme dei numeri interi, definiscono matrici simili a quelle ammesse in FORTRAN e BASIC.

All'istruzione BASIC
DIM MAT% (30,20)
corrisponde in PASCAL
type matrix = *array* [0..30, 0..20] *of* integer;
var MAT: matrix;
o, più brevemente:
var MAT: array [0..30, 0..20] *of* integer;
dove gli indici sono di tipo subrange, da zero alla dimensione massima.

Si noti intanto che non è detto che il sottoinsieme di variabilità dell'indice debba per forza partire da zero: la dichiarazione
type balanced = array [-5..5] *of* real;
crea la struttura di fig. 3, che può essere





puntata anche con indici negativi.

Ma c'è di più: non è detto che l'indice di una matrice debba essere di tipo subrange, e che comunque debba per forza essere di tipo intero. Possiamo così avere matrici indicizzate dai caratteri alfabetici, o da un indice di tipo *scalar* definito dall'utente.

Ad esempio è ammessa la seguente definizione:
var MATCOL: *array* [colore] *of* integer;
che naturalmente si può indirizzare con una costante:
MATCOL [rosso]: =...
o con una variabile di tipo "colore";
var RAINBOW: colore;
MATCOL [RAINBOW]: = ...

Infine, anche le *stringhe* sono definite tramite una dichiarazione di array: esse hanno quindi, all'atto della definizione, una lunghezza massima definita.
type stringa 10 = *packed array* [1..10] *of* char;
definisce una stringa di 10 caratteri: il termine *packed* indica che la matrice è di tipo particolare: e vi possono essere eseguite le normali operazioni sulle stringhe.

Benché, come abbiamo visto, il tipo *array* offra molte possibilità, esso rimane sempre un tipo rigido e adatto più ai calcoli matematici che ad altre applicazioni di tipo più "moderno": per esempio è molto scomodo dover definire a priori la lunghezza dell'array. Una maggiore libertà in questo senso è concessa da un'altra struttura, definita con la dichiarazione di tipo *file*.

Il tipo file

Chiunque abbia lavorato con memorie di massa a disco o a nastro dovrebbe aver presente il concetto di *file* (o *archivio*): una sequenza (a priori illimitata) di componenti dello stesso tipo, che può essere aggiornata soltanto aggiungendone o togliendone elementi dal fondo.

Questa struttura è molto comoda quando occorra eseguire funzioni di *archiviazione*: un dato immagazzinato in un file sarà

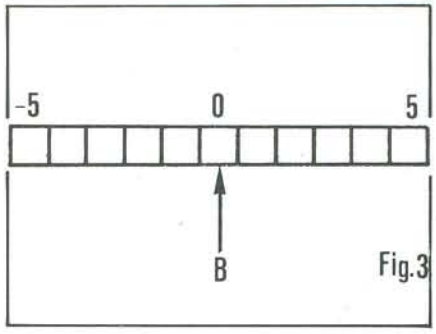


Fig.3

sempre presente (e leggibile) nella struttura, ma non potrà essere cancellato.

La dichiarazione di un file è la seguente:
type tipo file = *file of tipo*;
ove "tipo" può essere qualsiasi; anche strutturato, purché non a sua volta un file.

L'esempio più immediato di struttura a file è il *testo*, definito come *"file of char"*: questa definizione è talmente comune che le si è dato il nome *"text"*: *var* pincopallino: *text;* definisce la variabile "pincopallino" come file di caratteri.

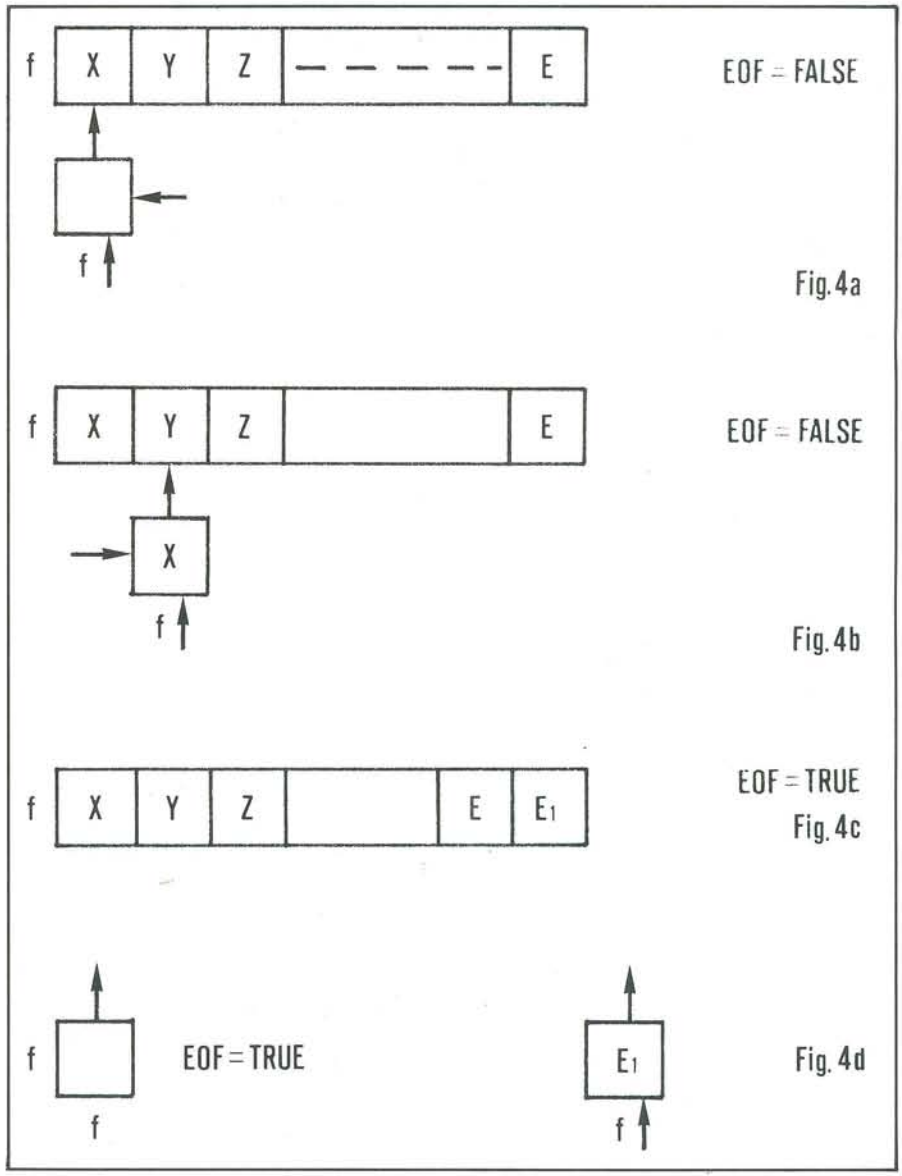
Ogni dichiarazione di variabile di tipo *file* crea automaticamente un'altra variabile del tipo di cui si compone il file: se XXX è il nome del file, questa variabile verrà indicata con XXX↑.

Così, date le definizioni:
var ALFA: *file of* real;
var BETA: *file of array* [1..10] *of* integer;
var GAMMA: *text;*
vengono create non soltanto i *files* ALFA, BETA e GAMMA, ma anche tre variabili ALFA↑, BETA↑ e GAMMA↑ che sono rispettivamente una variabile reale, una matrice di 10 numeri interi e una variabile char.

Queste variabili si chiamano *finestre*, e permettono di accedere ai rispettivi files per leggere ed aggiungere gli elementi.

La fig. 4 mostra le quattro fondamentali operazioni eseguibili sul file:

a - reset (f) posiziona la finestra al *primo elemento* del file.
In questa posizione è accessibile l'elemento X.



b - get (f) legge nella finestra l'elemento corrente e posiziona la finestra al prossimo elemento. Nel nostro esempio, poiché con il *reset* precedente ci eravamo posizionati all'inizio del file, l'operazione genera la lettura del primo elemento X nella finestra, e lo spostamento della finestra stessa al secondo elemento Y.

N.B. 1) L'operazione get non può essere eseguita se la finestra punta in fondo al file (situazione di end-of-file): esiste una funzione logica EOF (f) per mezzo della quale si può controllare in ogni momento se la finestra ha raggiunto la fine del file. Se EOF (f) è vera (true), la finestra f↑ non punta ad alcun elemento.

2) Una get può essere anche usata solo per spostarsi lungo il file; in tal caso il contenuto della finestra non interessa.

c - put (f), al contrario della precedente, è eseguibile soltanto se la finestra si trova in fondo al file, ossia se EOF (f) è vera: questa funzione *appende* il contenuto della finestra al file *come suo ultimo elemento* e si posiziona di nuovo in fondo al file, ossia EOF (f) rimane vera.

d - rewrite (f) azzera completamente il file: non essendoci più alcun elemento, la finestra si trova all'inizio del file con EOF (f) = true.

Il programma I presenta un breve esempio di uso della struttura a file, in cui si suppone che esista un file NUMERI di numeri interi, negativi e positivi. Il programma suddivide i numeri in due files NEGA e POSI a seconda del loro segno.

L'istruzione if.. then.. else è uguale a quella del BASIC, mentre la *repeat* (blocco) *until* (condizione) provoca la ripetizione del blocco finché la condizione non si avvera.

Si noti anche la chiarezza della strutturazione a blocchi *compound* di un programma PASCAL.

Nonostante sia già un bel passo avanti rispetto alle matrici, il file resta una struttura molto rigida, che non soddisfa ancora le esigenze del software moderno: in particolare è scomodo l'ordinamento univoco e rigoroso dei dati di queste strutture.

Un altro tipo di dato strutturato non tiene invece alcun conto dell'ordinamento degli elementi che lo compongono: si tratta infatti dell'*insieme*.

Il tipo set

Il PASCAL è il primo linguaggio capace di lavorare sugli insiemi con le funzioni e le operazioni caratteristiche della relativa teoria. L'esigenza era tanto più sentita quanto più la insiemistica era entrata a far parte delle teorie matematiche moderne: e la trasposizione degli algoritmi elaborati "sulla carta" (molto spesso in termini di insiemi) in programmi eseguibili da un calcolatore richiedeva notevoli equilibrismi per poter tradurre in termini di numeri quelle astruserie.

Valga per tutti la teoria dei grafi, che ha dovuto essere completamente reimpostata a colpi di matrici per cavarne fuori qualcosa di sensato per un computer.

Molto successo ha quindi riscosso il tipo *set*, che definisce un insieme:

type nome = *set of* tipo;

ove il "tipo" è logicamente *non strutturato*.

Qualche esempio:

type lettere = *set of* 'A'.. 'Z';
type numeri = *set of* integer;
type colors = *set of* colore;

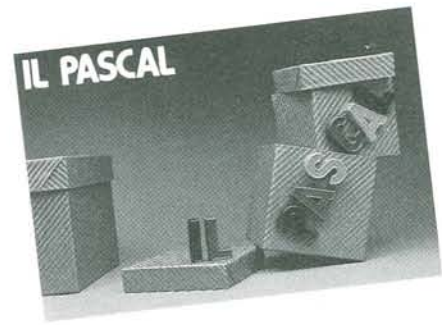
L'*assegnamento* di una variabile di tipo insieme è molto interessante, poiché offre tutta una serie di possibilità. Vediamone qualcuna:

var ALFA: lettere;
ALFA: = [A,I,S]; (1)
ALFA: = [A..D, W..Z]; (2)
ALFA: = [ ]; (3)

La (1) assegna alla variabile ALFA il valore [A,I,S], ossia stabilisce che l'*insieme* ALFA contiene quelle tre lettere dell'alfabeto; la (2) vi assegna le prime quattro e le ultime quattro lettere dell'alfabeto; la (3) vi assegna l'insieme vuoto.

Le *operazioni* sugli insiemi sono quelle di unione, intersezione e differenza, espresse rispettivamente dagli operatori +, * e -: il seguente esempio illustra il funzionamento di queste operazioni:

var ALFA, BETA, GAMMA: lettere;
ALFA: = [A,B,C,D].
BETA: = [C,D,E,F];
GAMMA: = ALFA + BETA; (4)



GAMMA: = ALFA * BETA; (5)

La (4) attribuisce a GAMMA il valore [A..F] (sorpresi? Controllate per credere!), mentre la (5) vi attribuisce il valore [C,D].

Si possono anche eseguire le operazioni di confronto:

if ALFA > = BETA then...

che pone la condizione che BETA sia un sottoinsieme di ALFA.

Programma I

```
type numfile = file of integer;
var NUMER, POSI, NEGA: numfile;
begin reset (NUMERI);
rewrite (POSI); rewrite NEGA;
repeat get (NUMERI);
      if NUMERI↑ < 0 then
        begin NEGA↑: = NUMERI↑;
        put (NEGA)
        end
      else begin POSI↑: = NUMERI↑;
        put (POSI)
        end
until EOF (NUMERI)
end;
```

```
!NUMERI      inizio file
!azzeramento POSI e NEGA
!ciclo di lettura

!elemento negativo: si
!appende a NEGA

!elemento positivo: si
!appende a POSI

!si ripete il ciclo fino ad EOF
```

Programma II

```
var ALFA: set of 'A'.. 'Z';
var CH: char;
ALFA: = [A..Z];
begin repeat read (CH);
if CH in ALFA then write CH
else write ' '
until CH = '.'
end;
```

Infine la funzione logica *in* rappresenta l'*appartenenza* di un elemento ad un insieme:

if 'c' *in* ALFA *then*...

Il programma II esegue la lettura e scrittura di un file di caratteri (le istruzioni *read* e *write* leggono e scrivono un carattere per volta); vengono però scritte solo le lettere: al posto degli altri simboli (numeri, punteggiatura...) vengono scritti degli spazi. Il punto chiude l'operazione.

Conclusione

Non abbiamo ancora concluso l'esame dei tipi strutturati del PASCAL: tuttavia i due che ancora restano da analizzare sono i più importanti, e non consentono un'esposizione affrettata.

Contentiamoci dunque per ora di lavorare su matrici, files e insiemi, e lasciamo al prossimo numero le strutture più complesse ed affascinanti: il *puntatore* con cui si possono costruire liste ed alberi e il *record*, con cui raggrupperemo dati anche diversissimi fra loro in un'unica struttura.

Pietro Hasenmajer