di Sergio Polini

Procedure ricorsive per strutture di dati ricorsive

Eccoci finalmente al cuore del nostro MiniMake, al suo motore ricorsivo. Abbiamo visto l'altra volta che le procedure ricorsive possono essere di grande utilità: spesso aiutano a pervenire nel modo più rapido alla descrizione in pseudocodifica dei nostri algoritmi. Allora ci siamo soffermati su semplici esempi tratti dalla matematica, tutti esempi che ammettevano una facile soluzione iterativa. Ci importava soprattutto affrontare un tema delicato con gradualità. Cercheremo ora di muovere i primi passi, con la stessa gradualità, su un terreno forse di più concreto interesse: algoritmi ricorsivi come il modo più naturale di trattare strutture di dati ricorsive

Ricordate la definizione di lista lineare data il mese scorso? Una lista lineare è o una lista nulla o un atomo (un nodo) seguito da una lista. Se devo costruire una lista, posso partire da una lista nulla, cioè un **nil**, aggiungendo poi un nodo il cui puntatore Next valga nil. Ottengo così una lista; se aggiungo un atomo il cui Next punti al primo, cioè alla lista appena costruita, ottengo un'altra lista. E così via.

Sappiamo già che potremmo definire le liste lineari anche in altro modo, come sequenze finite e ordinate di nodi legati tra loro dai rispettivi puntatori. Sappiamo anche (ne abbiamo parlato a gennaio) che le «sequenze» possono essere manipolate mediante un normale ciclo **while**, senza bisogno di ricorsività. Partire tuttavia direttamente da situazioni in cui l'approccio iterativo sarebbe innaturale o acrobatico, da proble-

```
program ProcedureRicorsive1;

type
  NPtr = ^Nodo;
  Nodo = record
    Ch   : char;
    Next: NPtr
  end;

var
  Lista: NPtr;

procedure PreparaLista;
var
  i : integer;
  NP: NPtr;
begin
  Lista := nil;
  Randomize;
  for i := Random(26) downto 0 do begin
    New(NP);
    NP^.Ch   := Char(Ord('a')+i);
    NP^.Next := Lista;
    Lista    := NP
  end
end;

function NumeroNodi(var NP: NPtr): integer;
begin (* v. figura 2 *) end;

procedure WriteLista(NP: NPtr);
begin (* v. figura 3 *) end;

begin
  PreparaLista;
  Writeln('La lista contiene ', NumeroNodi(Lista), ' nodi.');
  Write('Eccoli: ');
  WriteLista(Lista)
end.
```

Figura 1 - Il corpo principale del nostro primo programmino di manipolazione ricorsiva di liste lineari, preceduto dalla dichiarazione di tipi e variabili, nonché da una procedura per la costruzione della lista su cui opereremo.

mi che ammettono una soluzione relativamente agevole solo se ricorsiva, potrebbe risultare un po' arduo. Metteremo quindi per un attimo da parte i cicli **while**, in modo da poter partire da strutture di dati semplici. Poi complicheremo un po' le cose.

Un caso semplice

Cominceremo da un programma che, data una lista lineare, ne conti e poi ne visualizzi i nodi. In figura 1 trovate innanzitutto la rituale dichiarazione di tipi e variabili, poi una procedura che provvede alla costruzione di una lista lineare; questa comprenderà un numero casuale di nodi, da 1 a 26, ognuno dei quali conterrà un campo di tipo carattere e l'immancabile puntatore Next al nodo successivo. Ad ogni esecuzione del programma verrà prodotta una lista con un diverso numero di nodi, il primo rappresenterà una «a», il secondo una «b», e così via fino al numero totale di nodi.

Notate che la preparazione della lista segue esattamente le stesse tappe che ci vengono suggerite dalla definizione ricorsiva: si parte da **nil**, la lista nulla, e via via si aggiungono «davanti» a questa nodi il cui campo Next punta ogni volta alla lista fino a quel momento costruita.

Vediamo poi il corpo principale del programma, che non fa altro che preparare la lista, contarne i nodi, visualizzarli uno di seguito all'altro. In fondo si tratta di liste che rappresentano stringhe!

Per prima cosa si tratta di contare. Ragionando sulla definizione ricorsiva, otteniamo subito un algoritmo, naturalmente anch'esso ricorsivo: se la lista è nulla, il numero dei nodi è zero; altrimenti il numero dei nodi è uno più il numero dei nodi della lista che segue il primo nodo. La figura 2 mostra la facile implementazione in Pascal del tutto.

Quanto a vedere sul video quali sono questi nodi, basta considerare che, una volta visualizzato il primo, quello che rimane... non è altro che una lista da visualizzare (figura 3).

```
function NumeroNodi(NP: NPtr): integer;
begin
  if NP = nil then
    NumeroNodi := 0
  else
    NumeroNodi := 1 + NumeroNodi(NP^.Next)
end;
```

Figura 2 - Il conto del numero di nodi in una lista lineare.

Immagino che giudicherete tutto ciò un po' banale e un po' contorto al tempo stesso: «ma non si fa prima con un ciclo **while**?». Probabilmente la risposta è sì, soprattutto se ricordiamo che la ricorsività ha un costo. Tuttavia vorrei farvi notare che, se una soluzione iterativa è a portata di mano, è solo perché è immediata la conversione delle nostre definizioni ricorsive in definizioni tail-ricorsive. Ricorderete che queste sono in genere caratterizzate dalla presenza di una variabile in cui si «accumula» progressivamente il risultato dell'elaborazione; nel caso della funzione NumeroNodi si tratterebbe di aggiungere una variabile NumNodi o simili, nel caso di WriteLista la soluzione è ancora più semplice: è lo stesso mio schermo che funge da accumulatore, in quanto vi compaiono via via i vari nodi proprio nell'ordine in cui voglio vederli.

Se però insistete, la figura 4 vi mostra una procedura nella quale l'approccio ricorsivo appare sicuramente più naturale: rovesciare una lista lineare, percorrere i suoi nodi dall'ultimo al primo, non è così facile con un processo iterativo. Tanto che, in casi del genere, si preferisce aggiungere ad ogni nodo anche un puntatore al nodo precedente, proprio per potersi muovere direttamente dall'ultimo al primo senza troppe acrobazie.

```
procedure WriteLista(NP: NPtr);
begin
  if NP = nil then begin
    Writeln;
    Exit
  end
  else begin
    Write(NP^.Ch);
    WriteLista(NP^.Next)
  end
end;
```

Figura 3 - Visualizzazione dei nodi di una lista lineare.

```
procedure WriteListaInvertita(NP: NPtr);
begin
  if NP = nil then Exit;
  if NP^.Next = nil then begin
    Write(NP^.Ch);
    Exit
  end
  else
    WriteListaInvertita(NP^.Next);
  Write(NP^.Ch)
end;
```

Figura 4 - Un problema meno banale: la visualizzazione di una lista lineare in ordine inverso, dall'ultimo nodo al primo.

Una lista un po' più articolata

Rimane il fatto che con le liste lineari è tutto troppo semplice. Proveremo quindi a complicare un po' le cose, aggiungendo ad ogni nodo della nostra lista, oltre al puntatore al nodo successivo, un puntatore ad un'altra lista. La figura 5 vi propone la dichiarazione di tipi e variabili, il corpo principale del programma e una procedura che prepara la nostra lista. Vi prego di notare che PreparaLista *non* è una procedura ricorsiva.

In figura 6 vi sono invece le due procedure ricorsive per la visualizzazione della lista e per il conto del numero dei suoi nodi: se la lista è vuota non ho nulla da fare; altrimenti scrivo o «conto» il primo nodo e poi tratto nello stesso modo sia la sublista che parte dal nodo che la lista principale «al netto» di questo. Anche qui non sarebbe stato affatto difficile o innaturale il ricorso ad un tranquillo ciclo **while**. Ma non saremmo andati molto lontano. Mi spiego. Come avevamo visto la volta scorsa, una definizione ricorsiva ha la caratteristica di poter descrivere insiemi potenzialmente infiniti con un numero finito di asserzioni (bastano i cinque assiomi di Peano per descrivere l'intero insieme dei numeri naturali). Entro certi limiti ciò si verifica anche con l'iterazione, ma solo entro certi limiti. Per tornare al nostro esempio, notate che in PreparaLista vi sono due cicli: uno per costruire i nodi della lista principale, un altro per costruire le subliste attaccate ai nodi di questa. Se volessi andare oltre, se volessi che anche dai nodi delle subliste partissero altre liste, dovrei aggiungere un altro ciclo. Provate a sostituire alla procedura PreparaLista della figura 5 quella della figura 7, non cambiate niente altro, lanciate il programma. Verificherete che (nonostante un po' di confusione su uno schermo, ahimè, bidimensionale) il programma funziona perfettamente senza bisogno di alcuna modifica alle due procedure ricorsive.

Morale: se conosco già l'esatta configurazione di una struttura di dati ricorsiva, posso anche permettermi di limitarmi a procedure iterative. Se invece conosco solo la definizione della struttura, se non so quanto profondamente si sviluppa, l'approccio ricorsivo è, se non l'unico possibile, almeno sicuramente quello più comodo.

```
program ProcedureRicorsive2;

type
  NPtr = ^Nodo;
  Nodo = record
    Ch        : char;
    Next      : NPtr;      (* puntatore al nodo successivo *)
    SubLista  : NPtr       (* puntatore ad un'altra lista  *)
  end;

var
  Lista: Nptr;

procedure PreparaLista;
var
  i, j       : integer;
  NP, SP, Sub: NPtr;
begin
  Randomize;
  Lista := nil;
  for i := Random(26) downto 0 do begin
    New(NP);
    NP^.Ch       := Char(Ord('A')+i);
    NP^.Next     := Lista;
    Sub          := nil;
    for j := Random(26) downto 0 do begin
      New(SP);
      SP^.Ch       := Char(Ord('a')+j);
      SP^.Next     := Sub;
      SP^.SubLista := nil;
      Sub          := SP
    end;
    NP^.SubLista := Sub;
    Lista := NP
  end
end;

procedure WriteLista(NP: NPtr);
begin (* v. figura 6 *) end;

function NumeroNodi(NP: NPtr): integer;
begin (* v. figura 6 *) end;

begin
  PreparaLista;
  Writeln;
  WriteLista(Lista);
  Writeln('Totale: ',NumeroNodi(Lista),' nodi.')
end.
```

Figura 5 - Un problema un po' più complicato del precedente: ogni nodo della lista punta, oltre che al nodo successivo, anche ad un'altra lista.

L'aggiornamento dei target

Con il MiniMake succede proprio questo: so che vi sarà una lista di target, da ogni nodo della quale partirà una lista di source. Non so quanti saranno i target né quanti saranno i source; soprattutto non so se e quali source compariranno in più di una delle subliste attaccate ai target, non so se e quali source saranno a loro volta target. Anche peggio del programma appena visto. Mi rassegno quindi ad una soluzione ricorsiva e comincio a tracciarne le linee fondamentali.

Rammentiamo che il programma lavora su un makefile contenente «regole» così fatte:

target: source [source ...]
<almeno uno spazio o tab> comando
[<almeno uno spazio o tab> comando]
[...]

Si parte da un target. Se si lancia il programma senza indicarne uno nella riga comando, si partirà dal target della prima regola del makefile, altrimenti da quello indicato. Il nostro obiettivo è confrontarne la data/ora con quella dei suoi source: se uno di questi è più recente, vengono eseguiti i comandi. Di norma questi aggiorneranno il target «ricostruendolo» a partire dai suoi source: il target sarà un file EXE o TPU, i source

```
procedure WriteLista(NP: NPtr);
begin
  if NP = nil then begin
    Writeln;
    Exit
  end
  else begin
    Write(NP^.Ch);
    if NP^.SubLista <> nil then begin
      Write(': ');
      WriteLista(NP^.SubLista)
    end;
    WriteLista(NP^.Next)
  end
end;

function NumeroNodi(NP: NPtr): integer;
begin
  if NP = nil then begin
    NumeroNodi := 0;
    Exit
  end
  else
    NumeroNodi := 1 + NumeroNodi(NP^.SubLista) + NumeroNodi(NP^.Next)
end;
```

Figura 6 - Le procedure ricorsive per la visualizzazione della lista generata dal programma di figura 5 e per il calcolo del numero dei suoi nodi.

saranno i suoi sorgenti e/o altri file TPU o OBJ. Non è però necessario che il target sia un file: potrebbe essere un nome qualsiasi, non dipendente da alcun source, utilizzato quindi solo come una sorta di abbreviazione di uno o più comandi. Se il target è un file ed esiste, si tiene conto della data e dell'ora registrate dal DOS nella directory, altrimenti gli si assegna una data convenzionale, azzerando il campo DataOra del corrispondente nodo nella lista dei pathname; lo si rende così sicuramente più vecchio degli eventuali source (procedura AssegnaDataOra in figura 8).

La prima cosa che dobbiamo fare è quindi cercare il nostro target sia nella lista dei target che in quella dei pathname; in quest'ultima per leggerne data e ora, nell'altra per verificare che si tratti davvero di un target (altrimenti non abbiamo ovviamente bisogno di aggiornarlo). Se il target non è in nessuna delle due liste si ha una situazione di errore: non è un file, ma non vi è alcuna regola che ci dica come costruirlo. Se è nella lista dei target, si tratta di scorrere la lista dei suoi source per il confronto dei rispettivi campi DataOra.

Può tuttavia verificarsi che un source sia a sua volta un target, che cioè vada aggiornato prima di procedere all'aggiornamento del suo target. Solo dopo aver aggiornato tutti i source da cui dipende posso eseguire i comandi che aggiornano il target. Se un source può a sua volta essere un target, nulla vieta che anche i suoi source siano a loro volta dei target, ecc. Tipica situazione ricorsiva.

Proviamo quindi a tracciare le linee del nostro algoritmo di aggiornamento come in figura 9. Si tratta solo di una prima approssimazione, in quanto ancora non abbiamo tenuto conto di un'ulteriore possibile complicazione: non solo un source può essere anche un target, ma tra i suoi source vi potrebbe essere finito anche quello che era il suo target. Questa volta non si tratterebbe di altro che di un errore puro e semplice: il programma deve essere in grado di riconoscere tali situazioni per rifiutarle (anche se si possono immaginare casi in cui quelli che ora appaiono come circoli viziosi sarebbero perfettamente legittimi; non in un make, comunque).

È questo il motivo per cui nei nodi della lista dei pathname vi è anche un campo Visite: mi serve per contare quante volte visito (passo per) uno stesso file. Il campo viene inizializzato a zero quando il nodo cui appartiene viene allocato in CercaPath (in MMSIM.PAS), diventa 1 non appena si rileva che il pathname è quello di un target, diventa 2 dopo che il target è stato aggiornato (se necessario). Ogni volta che si esamina un source si osserva il suo campo Visite: se è zero è la prima volta che ci passiamo, dobbiamo quindi vedere se è a sua volta un target per aggiornarlo se necessario; se è 2 vuol dire che è stato già aggiornato; se però è 1 vuol dire che si trova nello stesso stato del suo target, quindi che c'è circolarità.

Altra cosa di cui non si tiene conto nella figura 8 è il caso del target senza source, cioè di un target i cui comandi vanno eseguiti comunqe (di questo genere è il «print» del makefile del nostro MiniMake, visto a marzo: dando il comando «mmake print» si ottiene la

Figura 7 - Una procedura per preparare una lista (primo loop) dai cui nodi partono altre liste (secondo loop), dai cui nodi partono altre liste (terzo loop). Può essere inserita nel programma di figura 5 senza bisogno di modificare le procedure ricorsive della figura 6.

```
procedure PreparaLista;
var
  i, j, k    : integer;
  NP, SP, SSP, Sub: NPtr;
begin
  Randomize;
  Lista := nil;
  for i := 5 downto 0 do begin
    New(NP);
    NP^.Ch       := Char(Ord('A')+i);
    NP^.Next     := Lista;
    Sub          := nil;
    for j := 9 downto 0 do begin
      New(SP);
      SP^.Ch       := Char(Ord('a')+j);
      SP^.Next     := Sub;
      SP^.SubLista := nil;
      if (i = 0) and (j = 0) then
        for k := 4 downto 0 do begin
          New(SSP);
          SSP^.Ch       := Char(Ord('0')+k);
          SSP^.Next     := SP^.SubLista;
          SSP^.SubLista := nil;
          SP^.SubLista  := SSP
        end;
      Sub          := SP
    end;
    NP^.SubLista := Sub;
    Lista := NP
  end
end;
```

```
procedure AssegnaDataOra;
var
  f: file;
  p: PPtr;
begin
  p := PrimoPath;
  while p <> nil do begin
    Assign(f, p^.Nome);
    {$I-} Reset(f); {$I+}
    if IOResult <> 0 then
      p^.DataOra := 0
    else begin
      GetFTime(f, p^.DataOra);
      Close(f)
    end;
    p := p^.Next
  end
end;
```

Figura 8 - La procedura AssegnaDataOra scorre la lista dei pathname per assegnare ad ogni suo nodo la data e l'ora registrate nella directory. Se Nome non corrisponde ad alcun file esistente, assegna zero al campo DataOra.

```
procedure Aggiorna(Nome) =
  cerca Nome sia nella lista dei path che in quella dei target
  se non esiste e non e' un target -> errore
  se e' un target
    scorri la lista dei suoi source
    se uno di questi e' a sua volta un target
      Aggiorna(source)
    se il source e' piu' recente
      esegui il comando
      leggi di nuovo data e ora dei file interessati
```

Figura 9 - Una prima approssimazione, in pseudocodice, dell'algoritmo di aggiornamento del target.

stampa di tutti i sorgenti). Possiamo venirne a capo limitandoci a settare una variabile booleana se un target è più vecchio dei suoi source, ed eseguire poi i comandi se quella variabile vale TRUE oppure se la lista dei source è vuota (si fa così nella procedura Aggiorna della figura 10, per la quale mi sono ispirato al breve make proposto da A. V. Aho, B. W. Kernighan e P. J. Weinberger nel loro *The AWK Programming Language*, Addison-Wesley, 1988).

```
procedure Aggiorna(Nome: PathStr);
var
  TP     : TPtr;
  PP     : PPtr;
  SP     : SPtr;
  CP     : CPtr;
  Vecchio: boolean;
begin
  PP := CercaPath(Nome);
  TP := CercaTarget(Nome);
  if (PP^.DataOra = 0) and (TP = nil) then begin
    Writeln(Nome, ' non esiste e non so come farlo!');
    Halt(1)
  end;
  if TP <> nil then begin
    Vecchio := FALSE;
    TP^.Id^.Visite := 1;
    SP := TP^.SourceList;
    while SP <> nil do begin
      if SP^.Id^.Visite = 0 then
        Aggiorna(SP^.Id^.Nome)
      else if SP^.Id^.Visite = 1 then begin
        Writeln(Nome,' e ', SP^.Id^.Nome,' si definiscono l''un l''altro!');
        Halt(1)
      end;
      if TP^.Id^.DataOra <= SP^.Id^.DataOra then
        Vecchio := TRUE;
      SP := SP^.Next
    end;
    TP^.Id^.Visite := 2;
    if Vecchio or (TP^.SourceList = nil) then begin
      CP := TP^.CmdList;
      while CP <> nil do begin
        Esegui(TP^.Id^.Nome, CP^.Comando, CP^.Argomenti);
        CP := CP^.Next
      end;
      AssegnaDataOra;
      TP^.Id^.DataOra := $7FFFFFFF
    end
  end
end;
```

Figura 10 - La procedura Aggiorna nella versione definitiva. Dopo l'aggiornamento si assegna ad ogni target una data tale che nessun file possa risultare più recente (con $7FFFFFFF).

Conclusioni

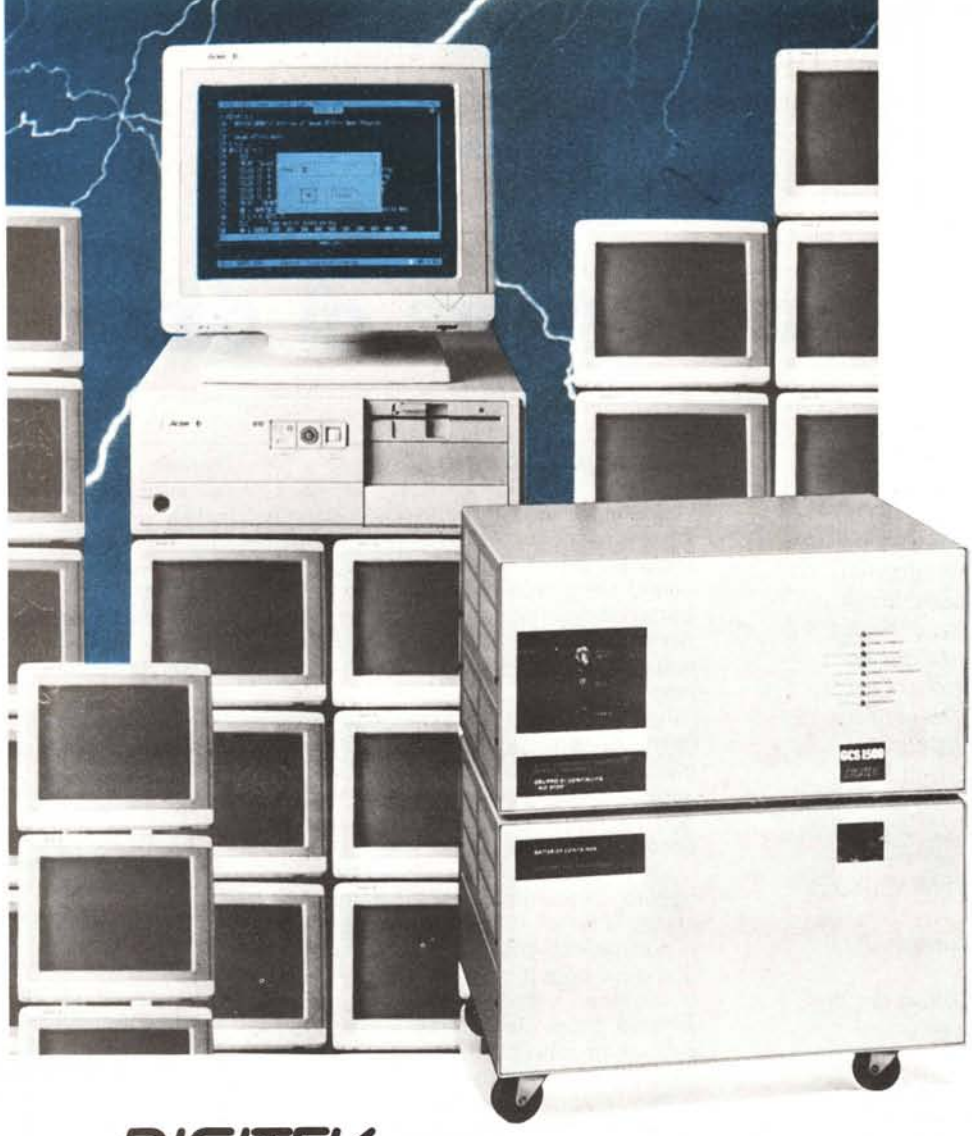
C'è chi sostiene che è sempre possibile convertire una procedura ricorsiva, anche non tail-ricorsiva, in una equivalente procedura iterativa (si tratta in sostanza di creare una variabile che faccia le veci dello stack, o magari tre variabili: uno stack per i parametri, uno per le variabili locali ed uno per gli indirizzi a cui devono tornare le singole chiamate ricorsive. Il metodo è illustrato nel quarto capitolo dei *Fundamentals of Data Structures in Pascal*, di E. Horowitz e S. Sahni, Pitman, 1984). C'è chi concorda, ma solo per precisare subito dopo che la conversione è tutt'altro che banale e che quindi si giustifica solo se si tratta di evitare un collo di bottiglia, se si tratta di rendere più efficiente proprio quella parte di un programma che ne compromette l'efficienza complessiva. C'è anche chi sostiene che in alcuni casi la conversione è semplicemente impossibile.

Non è necessario schierarsi con gli uni o con gli altri. Mi pare più utile considerare che in alcune situazioni l'approccio ricorsivo è estremamente comodo, a condizione, naturalmente, che non ci si proponga come un'inafferrabile magia.

Al riguardo potrei anche dire, per la mia personale esperienza di qualche anno fa, che cercare di eliminare a tutti i costi la ricorsività vuol dire rallentare l'apprendimento di tecniche con le quali prima o poi ci si deve confrontare (ad esempio quando si lavora con strutture di dati ad albero, o quando si passa a linguaggi come il LISP o il PROLOG) e che quindi la conversione da procedura ricorsiva a procedura iterativa va cercata solo in casi estremi.

In fondo anche il MiniMake non è poi così inefficiente: è addirittura più veloce del MAKE della Borland (ma solo perché è più semplice).

Spero di aver aiutato almeno qualcuno di voi a rompere il ghiaccio. Il mese prossimo cambieremo pagina per esaminare l'ultima parte del programma: l'uso della procedura Exec.



DIGITEK UNA PROTEZIONE SULLA QUALE PUOI CONTARE.

Gruppi di continuità NON-STOP e a RELÉ.

I black-out e le microinterruzioni dell'energia elettrica, oltre a danneggiare le Vs. apparecchiature, provocano variazioni o cancellazioni dei dati inseriti nel Vs. computer; a volte il danno rappresenta il lavoro dell'intera giornata.

Per eliminare questi costosissimi inconvenienti la DIGITEK propone due GRUPPI DI CONTINUITÀ:
- GRUPPI DI CONTINUITÀ "NON STOP" che, alimentando direttamente le apparecchiature attraverso le batterie, separano totalmente il carico dalle fluttuazioni ed instabilità della rete elettrica.
- GRUPPI DI CONTINUITÀ "A RELÈ" che intervengono, in caso di black-out o abbassamento della tensione sotto i 200V, in tempo utile per non creare problemi.

In caso di black-out, il gruppo, oltre a garantire il salvataggio dei dati, permette il proseguimento del lavoro, dandoVi una autonomia fino a 2 ore.

I gruppi di continuità della serie non-stop:

GCS	450	pot. nom.	450 VA
GCS	700	pot. nom.	700 VA
GCS	1000	pot. nom.	1000 VA
GCS	1500	pot. nom.	1500 VA
GCS	2400	pot. nom.	2400 VA

I gruppi di continuità della serie a relè:

GR	2428	pot. nom.	450 VA
GR	1000	pot. nom.	1200 VA
FS	4000	pot. nom.	4000 VA

DIGITEK

VIA VALLI, 28 - 42011 BAGNOLO IN PIANO (RE)
Tel. 0522/951523 r.a. - Telex 530156 - fax 0522/951526 G3

Desidero ricevere materiale illustrativo riguardante i Gruppi di continuità.
Cognome e Nome ..... Ditta .....
Via ..... Cap ..... Città ..... M.C.