di Sergio Polini

Strutture di dati ricorsive

La volta scorsa abbiamo introdotto alcuni concetti un po' astratti, necessari alla piena comprensione del funzionamento di un programma come MAKE: liste che contengono atomi e altre liste (che abbiamo chiamato «Liste maiuscole»), strutture di dati ricorsive, ordinamento parziale. Il tutto per introdurre una nostra versione «minima» di MAKE (non accetta commenti, regole implicite, macro né direttive; solo regole esplicite. Come al solito il sorgente completo è già disponibile su MC-Link, con il nome MMAKE.ARC)

Qualcuno potrà chiedersi che senso ha «rifare» MAKE. Vi sono due risposte a questa domanda. In primo luogo, come vi dicevo anche la volta scorsa, capire il funzionamento di un programma di utilità può aiutarci ad usarlo nel modo migliore. Soprattutto, però, avremo l'occasione di vedere come vanno affrontati problemi «ricorsivi». Qui abbiamo a che fare con un makefile, ovvero con un file contenente «regole» mediante le quali si esplicitano le dipendenze di alcuni target dai rispettivi source, insieme alle azioni che occorre eseguire per aggiornare un target se i suoi source sono stati modificati. Si tratta di una situazione ricorsiva perché anche i source, a loro volta, possono essere target di altri source (ad esempio: un file EXE dipende da diversi moduli OBJ o TPU, i quali dipendono dai rispettivi sorgenti). Può anche capitare, tuttavia, di dover lavorare sui dati relativi alle parti di alcune macchine: per ogni macchina vi saranno alcuni componenti fondamentali, a loro volta ulteriormente scomponibili, e così via, fino ad arrivare alle parti più elementari. L'aggiornamento di un target e la cosiddetta «esplosione delle parti» (l'elenco dettagliato di tutte le parti di una macchina) sono problemi piuttosto simili.

Un po' di metodo

Appena abbiamo iniziato la scrittura di QUED ci siamo subito posti un problema di metodo: abbiamo visto i limiti di un'adozione acritica del metodo top-down, abbiamo distinto tra i programmi dominati dalla struttura dell'input e quelli in cui predomina la struttura dell'output. Anche in MiniMake, come in QUED, il problema principale è rappresentato dall'input: l'output non sarà altro che un target aggiornato, ma per arrivarci dobbiamo prima venire a capo delle «dipendenze» codificate nel makefile.

A differenza di allora, tuttavia, ora sappiamo come portare a termine l'analisi lessicale e sintattica dell'input (almeno a grandi linee...); questo ci consente di procedere in modo più ordinato, con un piano generale delle operazioni sintetizzato nel corpo principale del programma (figura 1).

Per prima cosa dobbiamo aprire il makefile: questo non sarà altro che un file di testo contenente le regole esplicite per l'aggiornamento di uno o più target (la volta scorsa vi ho proposto un makefile contenente le regole per l'aggiornamento dello stesso programma MiniMake e per la stampa di tutti i suoi sorgenti). È ovviamente essenziale che il makefile esista; MiniMake assume che si chiami proprio MAKEFILE e che si trovi nella directory corrente.

Si procede poi a decodificare le dipendenze e ad organizzarle costruendo una Lista del tipo descritto a marzo: una Lista cioè che può a sua volta contenere altre Liste. Una struttura di dati ricorsiva (se volete darci subito un'occhiata ne trovate un diagramma nella figura 3) è lo strumento più adatto per affrontare un problema per sua natura ricorsivo.

Dobbiamo poi prendere nota della data e dell'ora di tutti i file coinvolti (un target va aggiornato se risulta avere data e/o ora più vecchie di quelle dei suoi source). Si potrebbe pensare di farlo mentre si costruisce la Lista, quindi nell'ambito della procedura Parse; una più attenta riflessione ci porta tutta-

```
begin
  Init;                                  (* Apertura del makefile   *)
  Parse;                                 (* Costruzione della Lista *)
  AssegnaDataOra;                        (* Lettura di data e ora   *)
                                         (*    dei file interessati *)
  if ParamCount > 0 then                 (* Se indicato un target,  *)
    Aggiorna(ParamStr(1))                (*    si aggiorna questo;  *)
  else                                   (* altrimenti si aggiorna  *)
    Aggiorna(PrimoTarget^.Id^.Nome)      (*    il primo target del  *)
end.                                     (*    makefile.            *)
```

Figura 1 - Il corpo principale del programma MiniMake.

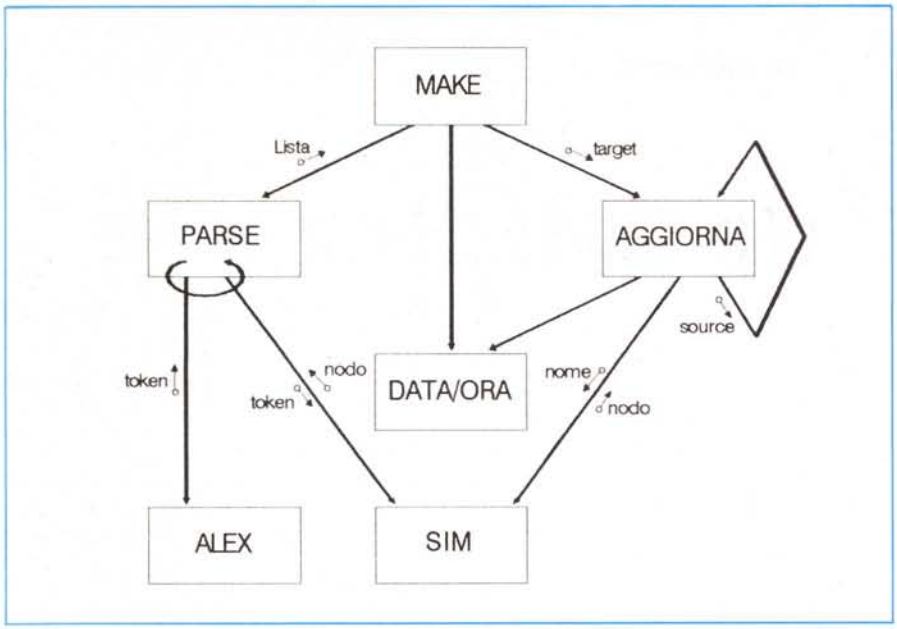

Figura 2 - La struttura del programma.

via a prevedere una distinta procedura AssegnaDataOra. Vedremo subito perché. Concludiamo intanto l'esame del nostro piano d'azione: una procedura Aggiorna si incaricherà di verificare se il target indicato (con un comando del tipo: MMAKE TARGET) è più vecchio dei suoi source; in caso affermativo vengono eseguiti i comandi indicati nel makefile. Se non viene indicato alcun target nella riga comando, viene aggiornato il target della prima regola contenuta nel makefile.

Quello che la figura 1 non consente di apprezzare è che Aggiorna è una procedura ricorsiva; ciò comporta, tra l'altro, che verrà eseguita più volte: la prima su chiamata del corpo principale del programma, le altre su chiamata di se stessa. Ogni volta che viene eseguita può trovare un target da aggiornare, e può quindi eseguire i comandi necessari; questi possono essere di qualsiasi natura, spesso non si tratterà di altro che di una compilazione. Ma una compilazione può rendere «nuovi» alcuni file EXE, OBJ o TPU, generando una situazione diversa da quella sulla cui base Aggiorna aveva lavorato; prima di chiamare nuovamente se stessa, quindi, Aggiorna deve prendere daccapo nota di data e ora dei file. È questo il motivo per cui AssegnaDataOra deve essere una procedura separata.

La struttura del programma

Per giungere ad un'idea più chiara di come dovrà essere il nostro programma, può essere utile un diagramma come quello in figura 2. Si tratta di una rappresentazione ispirata agli insegnamenti del «disegno strutturato» (Yourdon/Constantine, *Structured Design*, Yourdon Press, 1979) e, pur contenendo non poche semplificazioni, consente di mettere a fuoco molti aspetti importanti.

La parte sinistra del diagramma dovrebbe esservi ormai familiare: il programma si avvale di un modulo di analisi sintattica, il quale chiama ripetutamente (come indica la freccia ricurva) il modulo di analisi lessicale, ottenendone uno dopo l'altro i token che questo riconosce nell'input. Abbiamo in input un makefile contenente target, source e comandi, i quali andranno poi ordinati. Come abbiamo visto la volta scorsa, la relazione d'ordine sarà determinata dalla data e ora di tutti i file elencati nel makefile, ma non tutti i file saranno confrontabili (un file EXE viene prima dei suoi OBJ, ma non c'è alcuna relazione tra due differenti OBJ o tra un OBJ e il sorgente di un altro). Dovremo quindi procedere ad un «ordinamento parziale», per il quale ci serviremo di una Lista di Liste. Il modulo di analisi sintattica deve quindi trasformare i token in nodi; per far ciò, prevediamo che si serva di un modulo di gestione di simboli (nome e tipo dei file).

A questo punto il controllo ritorna al corpo principale del programma, che procede all'aggiornamento del target indicato dall'utente (o, in mancanza, di quello di default: il primo nel makefile). La freccia che parte dal modulo di aggiornamento per poi ritornarvi ci indica che lì si esegue una elaborazione ricorsiva: si tratta infatti di verificare se il target è più vecchio dei suoi source (nel qual caso vanno eseguiti i relativi comandi); si tratta però anche di verificare se i vari source sono a loro volta target di altri source, se sono più vecchi di questi, e così via. È questo il motivo per cui quella freccia ritorna al modulo da cui parte «portandosi appresso» un source.

Abbiamo già accennato alla necessità di prendere nota della data e ora di tutti i file sia prima che durante la fase di aggiornamento (necessità evidenziata dalle due frecce che vanno al modulo «DATA/ORA»). Rimane un'ultima considerazione.

Potremmo pensare di passare ad AGGIORNA un puntatore al nodo relativo al target, e di lasciare poi allo stesso AGGIORNA il compito di scorrere la Lista per cercare i suoi source, i source di questi, e così via. In realtà, però, cercare un target è qualcosa che deve fare anche il modulo di analisi sintattica: PARSE parte da un token (il nome di un file) per assicurarsi che non vi siano più regole relative ad uno stesso target; AGGIORNA parte da un source (e anche qui si può usare il nome del file) per verificare se è anche un target. Conviene quindi usare una stessa funzione CercaTarget, alla quale si passerà un nome di file e che ritornerà il puntatore al nodo corrispondente se questo è nella Lista dei target, o nil in caso contrario. Di qui la freccia da AGGIORNA a SIM.

La struttura dei dati

Ogni target può avere un qualsiasi numero di source; se è più vecchio di uno di questi vanno eseguiti alcuni comandi: spesso si tratterà solo di una compilazione, ma in linea di principio non vi devono essere limiti al numero di comandi. Da ogni nodo-target partiranno quindi sia una Lista di source che una lista di comandi (vedremo subito perché quella è «maiuscola» e questa no).

I nodi-target faranno a loro volta parte di una Lista: in questo modo sarà agevole verificare se il source di un target è a sua volta un target. Ciò comporta che uno stesso file può comparire in più Liste. A marzo vi ho proposto il makefile del MiniMake; se lo riguardate un attimo (potreste anche ricostruirlo sulla base della figura 3) vi accorgerete che la unit MMALEX.TPU è source sia di MMAKE.EXE che di MMPARSER.TPU ed è target di MMALEX.PAS: compare quindi sia nella Lista dei target che in due Liste di source.

Ci viene d'aiuto ora una caratteristica delle Liste maiuscole: la possibilità di condividere elementi. Ricordate quei discorsi (un po' astrusi) del mese scorso,

circa un generico campo Desc contenente un puntatore ad una misteriosa «memoria dei simboli»? Bene. I nodi delle tre Liste in cui compare MMALEX.TPU non conterranno un campo NomeFile di tipo stringa, ma un campo di tipo «puntatore alla memoria dei simboli». Dato che il Pascal non ci offre nulla del genere, ce la dobbiamo/possiamo costruire come ci pare. Sceglieremo una normalissima lista lineare (invece di un array), per il semplice motivo che non possiamo stabilire a priori il numero massimo dei suoi elementi. I nodi relativi a MMALEX.TPU conterranno ognuno un puntatore allo stesso nodo di questa lista, divenendo così «uguali» pur rimanendo distinti: distinti perché presenti in tre Liste diverse, uguali perché se cambio un qualcosa nel nodo della «memoria dei simboli» a cui tutti e tre puntano è come se avessi cambiato tutti e tre i nodi.

In termini pratici, questa fantomatica «memoria dei simboli» (la zona ombreggiata nella figura 3) ci consente un'efficiente registrazione della data e ora di tutti i file su cui devo lavorare, in quanto ognuno vi compare una sola volta. Se dovessi scorrere sia la Lista dei target che tutte le Liste dei source mi troverei ad effettuare quella rilevazione più volte per alcuni file (tre volte per MMALEX.TPU). Potreste anche pensare a più file di dati, ognuno contenente un campo CodiceCliente, affiancati da un file Clienti nel quale siano registrate le informazioni anagrafiche corispondenti ad ogni codice: non sarebbero altro che una diversa rappresentazione i primi delle nostre Liste, l'altro della nostra memoria dei simboli. Nulla di misterioso.

Le liste dei comandi non richiedono comunque tanta sofisticazione: i comandi non hanno data e ora, ed è ben probabile che ogni target abbia i suoi propri comandi, diversi da quelli necessari ad aggiornare gli altri. Anche le liste dei comandi potranno quindi essere normalissime liste lineari.

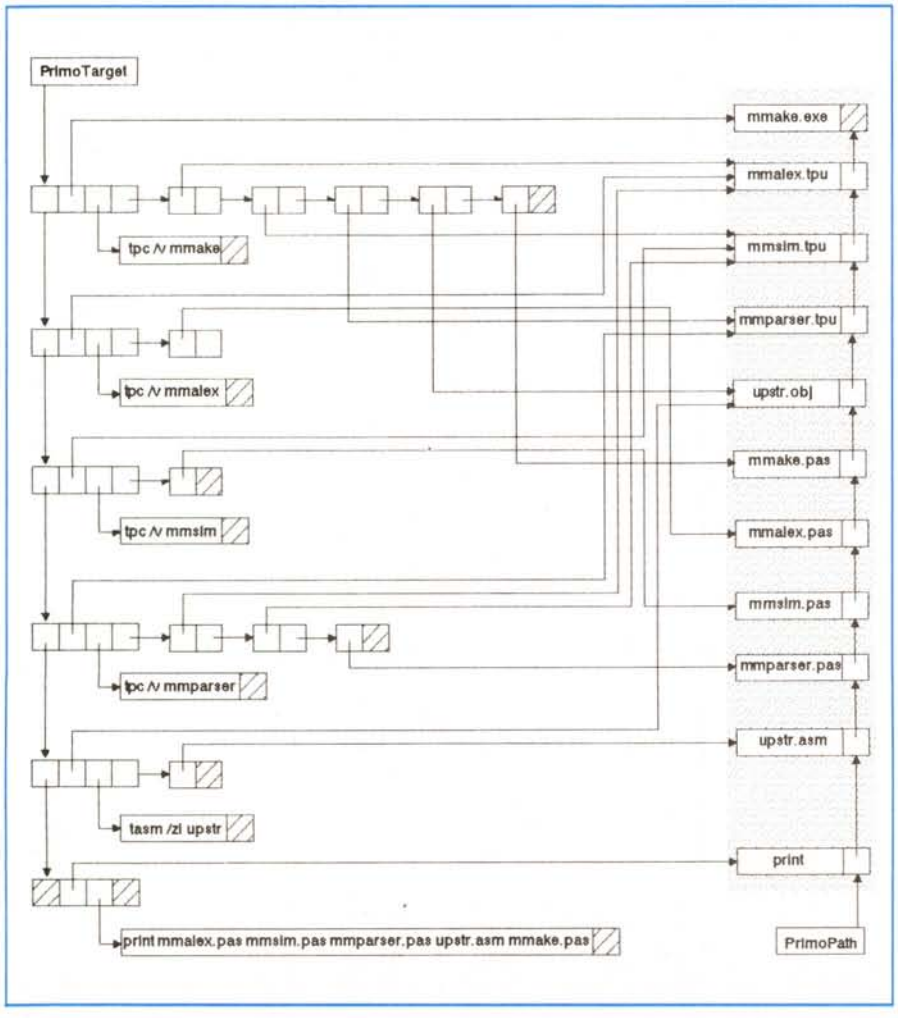

Figura 3 - La struttura dei dati del nostro MiniMake.

Compilazione condizionale

A questo punto dovrebbe risultarvi chiaro il sorgente della figura 4: si tratta appunto di MMSIM.PAS, la unit che si incarica di costruire tutte le nostre liste: quella dei pathname (la «memoria dei simboli») e quelle dei target, dei source e dei comandi.

Per quanto abbiamo detto sopra, non vengono trattate tutte nello stesso modo. La funzione CercaPath, ad esempio, ritorna il puntatore al nodo contenente pathname, data e ora di ogni file, creandolo se non lo trova; CercaTarget, al contrario, ritorna nil se non trova il suo argomento (per consentire al modulo di analisi sintattica di verificare che non vi siano più regole relative ad uno stesso target) ed è quindi affiancata da una funzione NuovoTarget.

Può forse valere la pena di sottolineare che varia anche l'aggiunta di nuovi nodi: CercaPath li aggiunge all'inizio della lista dei pathname (il primo nodo trattato sarà quindi l'ultimo della lista); NuovoTarget invece, come anche AddSource e AddCommand, inserisce i nuovi nodi sempre in fondo. Ciò è necessario soprattutto per AddCommand, in quanto i comandi relativi ad un target vanno eseguiti nello stesso ordine in cui sono elencati nel makefile. Per il resto ho voluto solo proporvi esempi delle due diverse tecniche.

Vorrei tuttavia porre l'accento sulla particolare struttura del sorgente: grazie alle direttive di compilazione condizionale (una novità introdotta con il Turbo Pascal 4.0), è possibile usare lo stesso sorgente sia come unit che come programma a sé stante. Attraverso il menu Options/Compiler/Conditional defines è possibile «definire» un simbolo («Main» nel nostro caso) e servirsene per orientare la compilazione. Alcune parti del sorgente sono racchiuse in un $IFDEF ... $ELSE ... $ENDIF; le parti tra $IFDEF e $ELSE vengono compilate se «Main» è stato definito, quelle tra $ELSE e $ENDIF solo nel caso opposto.

Ciò consente di verificare immediatamente il corretto funzionamento della unit: definendo «Main» prima di compilare si ottiene un breve programma di test; ricompilando poi senza quel «Main» si prepara invece il file TPU che ci servirà per compilare il MiniMake completo.

Adotteremo lo stesso accorgimento anche per i moduli di analisi lessicale e sintattica, che vedremo nelle prossime due puntate.

```
{$IFDEF Main}
program MMSim;
{$ELSE}
unit MMSim;

interface
{$ENDIF}
uses Dos;
type
  TPtr          = ^TargetRec;
  SPtr          = ^SourceRec;
  CPtr          = ^CommandRec;
  PPtr          = ^PathName;
  PathName      = record        (* nodo della lista dei pathname, *)
    Nome        : PathStr;      (*   sia target che source *)
    DataOra     : longint;
    Visite      : word;
    Next        : PPtr
  end;
  TargetRec     = record        (* nodo della lista dei target *)
    Id          : PPtr;
    SourceList  : SPtr;
    CmdList     : CPtr;
    Next        : TPtr
  end;
  SourceRec     = record        (* nodo della lista dei source *)
    Id          : PPtr;         (*   per un target *)
    Next        : SPtr
  end;
  CommandRec    = record        (* nodo della lista dei command *)
    Comando     : PathStr;      (*   per un target *)
    Argomenti   : ComStr;
    Next        : CPtr
  end;
var
  PrimoTarget   : TPtr;
  PrimoPath     : PPtr;
{$IFDEF Main}
var
  NomeFile      : PathStr;
  Comando       : ComStr;
  NFPtr         : TPtr;
{$ELSE}
function CercaPath(PathName: PathStr): PPtr;
function NuovoTarget(PathName: PathStr): TPtr;
function CercaTarget(PathName: PathStr): TPtr;
procedure AddSource(Target: TPtr; Nome: PathStr);
procedure AddCommand(Target: TPtr; Comando: ComStr);

implementation
{$ENDIF}
var
  UltimoTarget : TPtr;

function CercaPath(PathName: PathStr): PPtr;
var
  p: PPtr;
begin
  p := PrimoPath;
  while (p <> nil) and (p^.Nome <> PathName) do
    p := p^.Next;
  if p = nil then begin
    p := PrimoPath;
    New(PrimoPath);
    with PrimoPath^ do begin
      Nome    := PathName;
      DataOra := 0;
      Visite  := 0;
      Next    := p
    end;
    CercaPath := PrimoPath
  end
  else                          (* PathName gia' in lista *)
    CercaPath := p
end;

function NuovoTarget(PathName: PathStr): TPtr;
var
  t: TPtr;
begin
  New(t);
  if PrimoTarget = nil then
    PrimoTarget := t
  else
    UltimoTarget^.Next := t;
  with t^ do begin
    Id         := CercaPath(PathName);
    SourceList := nil;
    CmdList    := nil;
    Next       := nil
  end;
  UltimoTarget := t;
  NuovoTarget  := t
end;

function CercaTarget(PathName: PathStr): TPtr;
var
  t: TPtr;
begin
  t := PrimoTarget;
  while (t <> nil) and (t^.Id^.Nome <> PathName) do
    t := t^.Next;
  CercaTarget := t
end;

procedure AddSource(Target: TPtr; Nome: PathStr);
var
  p, q: SPtr;
begin
  New(q);
  q^.Id := CercaPath(Nome);
  q^.Next := nil;
  p := Target^.SourceList;
  if p = nil then
    Target^.SourceList := q
  else begin
    while p^.Next <> nil do
      p := p^.Next;
    p^.Next := q
  end
end;

procedure AddCommand(Target: TPtr; Comando: ComStr);
var
  p, q: CPtr;
  i   : integer;
begin
  New(q);
  i := 1;
  while (i <= length(Comando)) and not (Comando[i] in [#9, ' ']) do
    Inc(i);
  q^.Comando   := Copy(Comando, 1, i-1);
  q^.Argomenti := Copy(Comando, i+1, SizeOf(ComStr));
  q^.Next := nil;
  p := Target^.CmdList;
  if p = nil then
    Target^.CmdList := q
  else begin
    while p^.Next <> nil do
      p := p^.Next;
    p^.Next := q
  end
end;

{$IFDEF Main}
procedure Print;
var
  t: TPtr;
  s: SPtr;
  c: CPtr;
begin
  t := PrimoTarget;
  while t <> nil do begin
    Writeln('Target: ', t^.Id^.Nome);
    Writeln('SourceList:');
    s := t^.SourceList;
    while s <> nil do begin
      Write(' ', s^.Id^.Nome);
      s := s^.Next
    end;


    Writeln;
    Writeln('CommandList:');
    c := t^.CmdList;
    while c <> nil do begin
      Writeln(#9, c^.Cmd);
      c := c^.Next
    end;
    Writeln;
    t := t^.Next
  end
end;

begin
  PrimoPath := nil;
  PrimoTarget := nil;
  Write('Target: '); Readln(NomeFile);
  repeat
    if CercaTarget(NomeFile) = nil then begin
      NFPtr := NuovoTarget(NomeFile);
      Write('Source: '); Readln(NomeFile);
      repeat
        AddSource(NFPtr, NomeFile);
        Write('Source: '); Readln(NomeFile)
      until NomeFile = '';
      Write('Command: '); Readln(Comando);
      repeat
        AddCommand(NFPtr, Comando);
        Write('Command: '); Readln(Comando)
      until Comando = ''
    end
    else
      Writeln('...gia'' definito');
    Write('Target: '); Readln(NomeFile)
  until NomeFile = '';
  Print;
  Write('Premi <Enter> ...');
  Readln
{$ELSE}
begin
  PrimoPath := nil;
  PrimoTarget := nil
{$ENDIF}
end.
```

Figura 4 - Il sorgente della unit MMSIM.PAS.