di Sergio Polini

Liste maiuscole e minuscole

Quando vi proposi la prova del Turbo Pascal 4.0 (MC n. 76), accennai molto velocemente ad alcuni programmi di utilità compresi nelle confezioni. Dissi che per illustrare esaurientemente MAKE, TOUCH e GREP non sarebbe bastato un intero articolo. Ora vorrei rimediare: non tanto dilungandomi su tutte le possibilità di quei programmi, ma piuttosto proponendovi di costruire insieme un piccolo MAKE e un TOUCH. Quest'ultimo non presenta particolari problemi; offre anzi l'opportunità di vedere in azione alcune delle funzioni e procedure che compaiono nel ricchissimo repertorio dei nuovi Turbo Pascal. MAKE è invece sicuramente più interessante: ci consentirà di applicare in un contesto completamente diverso alcune delle tecniche viste in QUED, come anche di portare avanti la nostra rassegna delle strutture di dati dinamiche. Mi auguro anche che, se ancora non usate MAKE e TOUCH, vedendo da vicino come sono fatti vi venga la voglia di aggiungerli alla vostra cassetta degli attrezzi: ne vale la pena

Un po' di storia. Negli anni '60 era praticamente impossibile trovare testi dedicati espressamente alle strutture di dati; gli argomenti che stiamo discutendo da circa un anno erano piuttosto oggetto di trattazioni relative ai linguaggi per la manipolazione di liste: il glorioso IPL di Newell, Shaw e Simon (una volta una memoria gestita mediante puntatori veniva chiamata «memoria NSS»), l'ormai scomparso SLIP di Weizenbaum, il fantastico LISP di McCarthy, il sempreverde SNOBOL di Farber, Griswold e Polonsky. Finché, nel 1968, uscì il primo volume di *The Art of Computer Programming* di Donald Knuth, intitolato *Fundamental Algorithms*. Vi si sosteneva in modo inoppugnabile che la gestione di liste non era una sorta di magia riservata a linguaggi specializzati, ma era al contrario alla portata praticamente di ogni linguaggio: Knuth usava addirittura un Assembler, e il nostro Corrado, prima di proporvi alberi e liste in C, si divertiva a maneggiare quella roba in Fortran. Anche noi (ricordate ALLOC, il programma pubblicato ad aprile dello scorso anno?) abbiamo visto come si possono gestire le liste senza ricorrere a variabili dinamiche. È chiaro che linguaggi come il C e il Pascal (o Ada, Modula–2, LISP, Scheme, ecc.) ci rendono la vita molto più facile, ma Knuth ha avuto un grande merito; oggi è a tutti chiaro che si può (anzi: si dovrebbe) parlare di strutture di dati indipendentemente dal linguaggio che si intende adottare, e che non sono poi così vistose le differenze tra i vari linguaggi: vi sono, ad esempio, interpreti Prolog e LISP scritti in C e in Pascal, programmi di intelligenza artificiale scritti in parte in C e in parte in LISP, e così via. Non meravigliatevi quindi se ora, allargando un po' il nostro orizzonte, parleremo non solo di liste più potenti di quelle viste finora, ma anche un po' di LISP.

Scatole e frecce

C'è chi le chiama *liste generalizzate*, e magari ha ragione; preferisco tuttavia

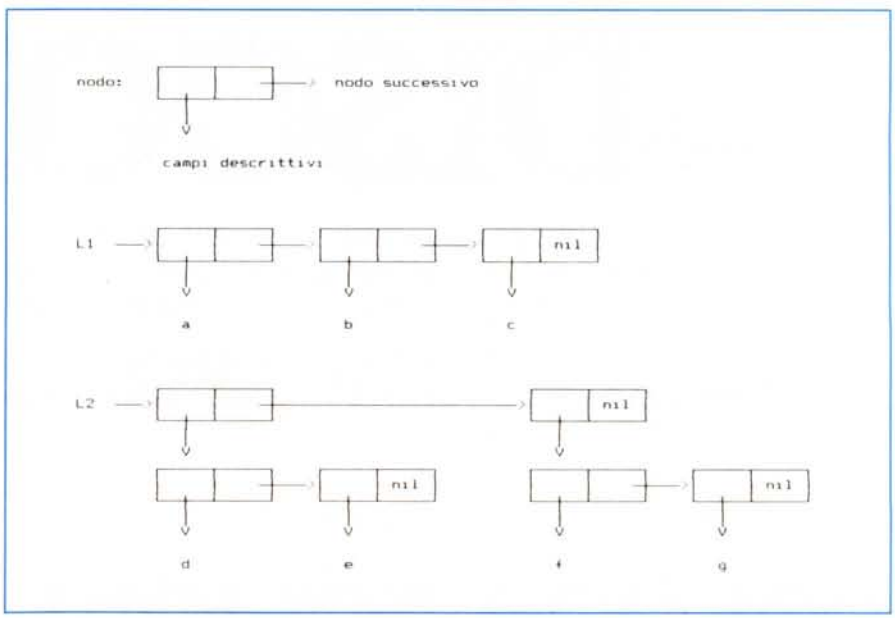

Figura 1 - Nell'ordine: un generico «nodo», la Lista L1 = (a b c), e la Lista L2 = ((d e) (f g)), contenente le subliste (d e) e (f g).

```
a)  type
        NPtr = ^Nodo;
        Nodo = record
             Nome     : string[25];    (*                              *)
             Cognome  : string[25];    (*       campi descrittivi      *)
             Indirizzo: string[40];    (*                              *)
             Next     : NPtr
        end;


b)  type
        NPtr = ^Nodo;
        Nodo = record
             Desc  : pointer;          (* puntatore ai campi descrittivi *)
             Next  : NPtr
        end;
```

Figura 2 - In a) abbiamo un esempio di dichiarazione «normale» nel nodo di una lista: uno o più campi descrittivi accompagnati da un campo «puntatore al nodo successivo». In b) abbiamo una dichiarazione più «generale» dello stesso nodo: un puntatore ai campi descrittivi al posto di questi.

distinguere tra liste maiuscole e minuscole per due motivi: primo perché lo trovo più divertente (de gustibus ...), poi perché, in fondo, è la stessa distinzione che propone Donald Knuth nei suoi *Fundamental Algorithms*. Knuth tratta diffusamente di liste come quelle che abbiamo visto fin qui: lineari e circolari, semplici e doppie, ecc., in generale liste intese come *sequenze ordinate di zero o più elementi*. Avverte però subito che vi è anche un altro tipo di liste, da lui chiamate «capital-Lists» (che traduco con Liste maiuscole, appunto) definite come *sequenze finite di zero o più atomi o Liste*. Apparentemente non ci dà una definizione positiva di atomo, ma solo una negativa: atomo è tutto ciò che non è Lista. Vedremo tuttavia che non si tratta di una definizione vuota; per farlo ci serviremo di un po' di LISP: atomi e Liste infatti (notate la maiuscola!) sono i tipi di dati fondamentali di quel linguaggio.

Nel nono capitolo del loro fondamentale *LISP* (Addison-Wesley, 1984, seconda edizione, basata sul Common LISP), Winston e Horn cercano di illustrare come atomi e Liste possono essere rappresentati nella memoria di un computer; considerano questa come composta di «celle di memoria» il cui contenuto (e quindi il «valore») può essere l'indirizzo di un'altra cella. Sappiamo bene (ne abbiamo parlato diffusamente a marzo e ad aprile dello scorso anno) che una variabile il cui valore sia un indirizzo è una variabile di tipo puntatore. Bene: la rappresentazione fondamentale dei dati in LISP è data da una cella di memoria contenente due puntatori, oppure, ma è la stessa cosa, da due celle adiacenti contenenti ognuna un puntatore.

Ai nostri fini potremo dire che l'elemento di base è un «nodo» con due campi di tipo puntatore, che può essere sia un atomo che una Lista. Il tutto viene illustrato mediante diagrammi «a scatole e frecce», come in figura 1, dove usiamo «nil» con l'abituale significato di «puntatore a nulla».

Abbiamo bisogno di una notazione convenzionale per evitare troppi giri di parole. Indicheremo quindi un atomo con una lettera minuscola, una Lista con una lettera maiuscola, il contenuto di una Lista come una successione di atomi o Liste racchiusa tra parentesi tonde. Sempre in figura 1 trovate la rappresentazione della Lista L1 = (a b c), cioè di una Lista contenente tre atomi, e della Lista L2 = ((d e) (f g)), che invece contiene a sua volta due Liste.

Torniamo ora ai nostri atomi. Siamo abituati a ragionare in termini di nodi definiti mediante record: uno o più campi descrittivi affiancati da un campo di tipo puntatore. Non c'è nulla di male, ma vi prego di considerare per un attimo questa possibilità come un'eccezione, per motivi che vedremo tra breve. Siamo anche abituati (grazie a QUED...) a manipolare Liste con nodi contenenti non solo un puntatore al nodo successivo, ma anche un puntatore al nodo precedente; dobbiamo però considerare questo secondo puntatore come un mero accessorio, come un artificio che ci consente a volte di scrivere programmi più efficienti, come un qualcosa da cui ora possiamo e dobbiamo prescindere. Ragioneremo quindi in termini di nodi contenenti un generico campo *Desc* di tipo *pointer*, intendendo che questo punti ai campi descrittivi del nodo, e un campo *Next*. Potremmo pensare che la rappresentazione «normale» (figura 2a) comprende un implicito campo *Desc* il cui valore sia qualcosa del tipo «proprio qui», una freccia che parta dal nodo per ritornarvi dopo un breve viaggio a 360 gradi. Potremmo anche pensare che, al pari del puntatore al nodo precedente, anche questo è nient'altro che un artificio: comprendendo i campi descrittivi direttamente nella dichiarazione del nodo si evita di accedervi poi mediante il «deriferimento» (che brutta parola!) del puntatore *Desc*: se Pippo è una variabile di tipo nodo, si fa prima a scrivere Pippo.Nome che Pippo.Desc^.Nome. Una notazione più breve che tuttavia ha un prezzo: si rinuncia a qualcosa. Abbiate pazienza: vi chiedo ancora di aspettare un attimo per vedere a cosa si rinuncia. Ora dobbiamo considerare che quei campi descrittivi puntati da *Desc* da qualche parte dovranno pure stare; in Pascal useremmo le procedure *New* o *GetMem* per allocare la memoria necessaria, proprio come faremmo per il nodo (l'abbiamo fatto in QUED: usavamo *New* per allocare il nodo corrispondente ad una riga di testo, *GetMem* per allocare alla memoria necessaria a contenere la riga vera e propria). Dobbiamo fare tuttavia anche lo sforzo di distinguere mentalmente tra due tipi di memoria: una memoria di nodi ed una memoria di simboli, quest'ultima contenente tutti i valori di tutti i nodi-atomi. Lo so che penserete che vi sto chiedendo troppo, ma quando esamineremo il diagramma delle strutture di dati del nostro Mini-Make tutto diventerà più concreto (almeno spero!).

Atomi e Liste condivisi

In LISP si usa la procedura APPEND per concatenare due o più Liste. Se abbiamo M = (a b) e N = (c d), possiamo costruire L = (a b c d) con:

```
(SETQ L (APPEND M N))
```

SETQ è una procedura di assegnazione, sui cui dettagli non ci soffermeremo; diciamo soltanto che serve ad as-

segnare ad L la concatenazione delle Liste M e N.

Potremmo pensare che, per costruire L, basta assegnare al campo *Next* dell'atomo «b» l'indirizzo dell'atomo «c», come in figura 3a. In realtà così facendo avremmo non solo una lista L = (a b c d), ma anche M diventerebbe (a b c d), mentre ovviamente vogliamo che M rimanga quello che era. Per risolvere questo problema, APPEND costruisce in primo luogo una copia di M, assegna ai campi *Desc* della copia gli stessi valori dei corrispondenti campi di M, quindi assegna al campo *Next* dell'atomo «b» della copia l'indirizzo dell'atomo «c» di N (figura 3b). «Assegnare gli stessi valori» vuol dire che il campo *Desc* dell'atomo «a» di M e il campo *Desc* dell'atomo «a» di L, avendo lo stesso valore, puntano alla stessa zona della «memoria dei simboli»; non sono quindi l'uno la copia dell'altro, sono proprio la stessa cosa: il Desc^.Nome del primo nodo di L e il Desc^.Nome del primo nodo di M coincidono.

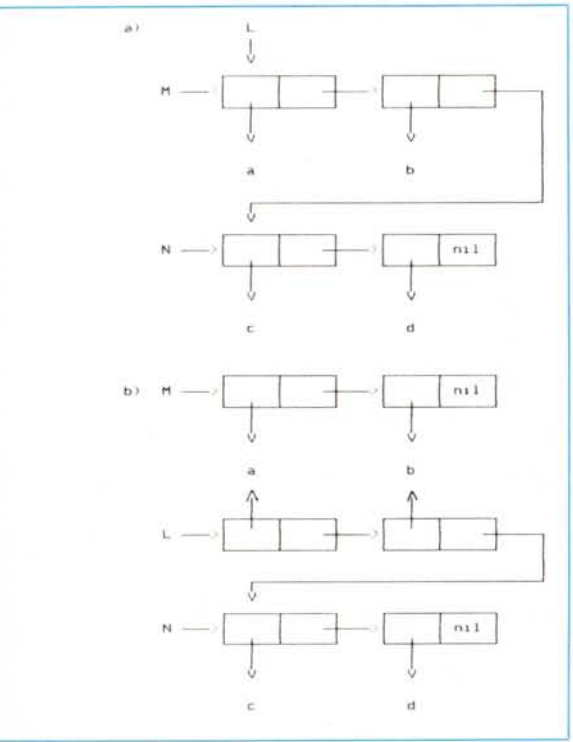

Figura 3 - Date due Liste M = (a b) e N = (c d), vediamo in a) come in LISP non *si costruisce una Lista L = (a b c d), in b) come invece viene costruita: mediante una copia della Lista M i cui atomi hanno lo stesso valore di quelli di M. «Gli stessi» attenzione, non nel senso che ne sono copia fedele, ma nel senso che è identico l'indirizzo assegnato ai rispettivi campi* Desc*; l'atomo «a» di M e l'atomo «a» di L non sono semplicemente uguali: sono proprio identici, usano la stessa area della «memoria dei simboli».*

```
mmake.exe: mmalex.tpu mmsim.tpu mmparser.tpu upstr.obj mmake.pas
     tpc /v mmake

mmalex.tpu: mmalex.pas
     tpc /v mmalex

mmsim.tpu: mmsim.pas
     tpc /v mmsim

mmparser.tpu: mmalex.tpu mmsim.tpu mmparser.pas
     tpc /v mmparser

upstr.obj: upstr.asm
     tasm /zi upstr

print:
     print mmalex.pas mmsim.pas mmparser.pas upstr.asm mmake.pas
```

Figura 4 - Il makefile del nostro MiniMake.

Ora possiamo finalmente cominciare ad esaminare le differenze tra liste minuscole e Liste maiuscole.

Una lista minuscola non è altro che una lista fatta di soli atomi: ogni nodo comprende (un puntatore a) campi descrittivi e un puntatore al nodo successivo. Non solo: ogni atomo ha i suoi propri esclusivi campi descrittivi; usando le convenzioni fin qui adottate, diremo che il campo *Desc* di ogni atomo punta ad una distinta area della «memoria dei simboli», non vi possono cioè essere due atomi con campi *Desc* aventi lo stesso valore.

È proprio questa caratteristica che ci consente quell'artificio che dicevamo prima: dato che ogni *Desc* è diverso da ogni altro, nulla vieta di fare a meno di *Desc* e di inglobare i campi descrittivi nello stesso nodo-atomo, come nella figura 2a e come abbiamo più volte fatto con le nostre liste semplici e doppie, lineari e circolari.

Una Lista maiuscola può invece essere fatta di atomi e Liste. Ciò rende necessario usare il generico campo *Desc*, in quando questo potrà puntare non solo a campi descrittivi contenuti nella «memoria dei simboli», ma anche ad un'altra Lista. In questo senso si deve dire che atomo è tutto ciò che non è Lista: i campi descrittivi di un atomo possono essere di qualsiasi tipo (numeri interi e reali, stringhe e caratteri, array, ecc.) tranne che di tipo Lista. Torniamo alla figura 1: il primo elemento della Lista L1 è un atomo perché i suoi campi puntano il primo alla «memoria dei simboli» e il secondo al nodo successivo *della stessa Lista L1*; il primo elemento della Lista L2 è una Lista perché punta ad un atomo, quindi alla «memoria dei nodi» cioè ad un qualcosa che contiene anche un puntatore (eventualmente nullo: ci sono anche Liste vuote o con un solo atomo) ad un nodo *di un'altra Lista*. La presenza obbligatoria del campo *Desc* si accompagna ad un'altra caratteristica delle Liste maiuscole: poiché posso assegnare quello che voglio ai campi *Desc*, posso creare Liste con atomi uguali (P = (a a b b)), Liste diverse che condividono alcuni nodi (come L e M), addirittura Liste che contengono se stesse!

Se sono riuscito a convincervi che le liste minuscole sono strutture di dati molto potenti, sarete d'accordo con me nel dire che le Liste maiuscole sono addirittura potentissime: non solo le liste minuscole, ma anche tutti i tipi di alberi e grafi possono essere rappresentati mediante Liste maiuscole, ma non viceversa. Questo proprio perché la estrema flessibilità delle Liste ne fa (a mio parere) la struttura di dati più generale. Un'ultima nota. Una Lista è una sequenza di zero o più atomi o Liste; una Lista può contenere altre Liste, perfino se stessa. Le Liste sono cioè strutture di dati *ricorsive*. Questo ha importanti conseguenze. Spesso si propongono algoritmi ricorsivi per calcolare un fattoriale o simili, magari senza precisare subito che si tratta solo di un esempio e che sarebbe folle calcolare i fattoriali in quel modo; la ricorsività è infatti tanto costosa quanto elegante e bisogna evitarla ogni volta che sia possibile. Sarebbe peraltro masochistico rifuggire da algoritmi ricorsivi quando si trattasse di gestire strutture di dati esse stesse ricorsive: per questo motivo il «motore» del nostro MAKE sarà proprio una procedura ricorsiva.

Ordine totale e parziale

Diciamo genericamente che un insieme di elementi è ordinato se è possibile dire quale viene prima di quale altro. Con un po' più di rigore dovremmo dire che un insieme è ordinato se vengono rispettate le seguenti condizioni (l'ultima può sembrare banale o senza senso, ma se mi mettessi anche a discutere di teoria degli insiemi...):

a) se a viene prima di b e b viene prima di c, allora a viene prima di c;
b) se a viene prima di b e b viene prima di a, allora a è uguale a b;
c) nessun elemento viene prima di se stesso.

Non è detto tuttavia che sia sempre possibile confrontare tutti gli elementi

di un insieme; se è possibile si parla allora di insiemi «totalmente ordinati» (detti anche catene). Una lista minuscola è appunto un insieme di questo genere: per ogni coppia di nodi si può stabilire quale viene prima e quale viene dopo, seguendo la «catena» dei puntatori che portano dall'uno all'altro. Un array è un altro esempio: per determinare se a[i] viene prima di a[j] basta confrontare i due indici i e j. Vi sono però alcuni tipi di problemi in cui si ha a che fare con insiemi «parzialmente ordinati», in cui cioè non è sempre possibile il confronto tra due elementi.

MAKE ne è un esempio. Per illustrare come funziona un MAKE vedremo come il nostro MiniMake è stato usato per ... costruire se stesso.

Si parte da un file, detto «makefile», che contiene alcune «regole» aventi il seguente formato:

```
target: [source...]
command
[command]
...
```

Target è in genere, ma non sempre, un file che va aggiornato, *source* è uno degli eventuali file da cui il *target* dipende, *command* è un comando da eseguire per aggiornare *target* se un qualche *source* è cambiato.

Guardate la figura 4, dove è riprodotto il makefile del MiniMake. La prima regola dice che MMAKE.EXE, il nostro programma, dipende dalle unit MMALEX.TPU, MMSIM.TPU e MMPARSER.TPU, dal file UPSTR.OBJ, infine dal suo sorgente MMAKE.PAS. «Dipende da» vuol dire che se uno di questi file è stato modificato (ha data e ora più recenti di MMAKE.EXE), bisogna eseguire il comando «TPC /V MMAKE» per produrre una versione aggiornata del programma (quel «/V» dice al compilatore di produrre codice contenente le informazioni necessarie per poi esaminare il programma con il Turbo Debugger).

Analoghe «dipendenze» vengono stabilite anche per i *source* di MMAKE.EXE, che in tal modo diventano a loro volta *target*: MMALEX.TPU dipende da MMALEX.PAS, UPSTR.OBJ dipende da UPSTR.ASM, e così via.

Poiché anche ciò da cui un *target* dipende può a sua volta dipendere da altri *source*, e questi da altri ancora, la prima cosa che MiniMake deve fare è mettere tutto in ordine: se UPSTR.ASM è stato modificato, bisogna prima aggiornare UPSTR.OBJ e poi MMAKE.EXE. Bisogna cioè ordinare le dipendenze («dipende da» è una relazione del tutto equivalente a quella «viene prima di» da cui abbiamo preso le mosse), tenendo però presente che non tutti i *target* e i *source* sono tra loro confrontabili: non ha senso, ad esempio, dire che MMALEX.TPU dipende da UPSTR.OBJ o viceversa. Non solo. Per costruire una versione aggiornata di MMAKE.EXE posso dare il comando «MMAKE MMAKE.EXE», se volessi solo aggiornare la unit MMPARSER darei «MMAKE MMPARSER.TPU», se volessi stampare tutti i sorgenti userei «MMAKE PRINT». Nel secondo caso si prescinde da tutto ciò che dipende da MMPARSER.TPU (e quindi non verrebbe ricompilato MMAKE.EXE), nel terzo addirittura ci limiteremmo ad una regola il cui *target* non solo non è un file, non solo non ha alcun *source*, ma non ha nessuna relazione di dipendenza con nessun altro *target* o *source* del makefile.

È chiaro, quindi, che non posso limitarmi ad un «normale» sort di tutti i *target* e i *source*, ma devo ordinare un insieme solo parzialmente ordinabile. Si parla in questi casi di «sort topologico» (ne trovate un esempio anche nel solito *Algorithms + Data Structures = Programs* di Wirth).

```
program Touch;
uses Dos;
var
  i: integer;
  Param: string;
  DirInfo: SearchRec;
  Trovato: boolean;
  Anno, Mese, Giorno, GiornoDellaSettimana: word;   (* per GetDate *)
  Ore, Minuti, Secondi, Centesimi: word;            (* per GetTime *)
  DT: DateTime;
  Adesso: longint;
procedure Aggiorna(NomeFile: PathStr);
  var
    f: file;
  begin
    Assign(f, NomeFile);
    Reset(f);
    SetFTime(f, Adesso);
    close(f)
  end;
begin
  Writeln('TOUCH Versione 1.0 - by Sergio Polini (MC1166)');
  if ParamCount = 0 then begin
    Writeln('Uso: touch file [file ...]');
    Halt(1)
  end;
  GetDate(Anno, Mese, Giorno, GiornoDellaSettimana);
  GetTime(Ore, Minuti, Secondi, Centesimi);
  with DT do begin
    Year  := Anno;      Hour  := Ore;
    Month := Mese;      Min   := Minuti;
    Day   := Giorno;    Sec   := Secondi
  end;
  PackTime(DT, Adesso);
  for i := 1 to ParamCount do begin
    Trovato := FALSE;
    Param := ParamStr(i);
    FindFirst(Param, Archive, DirInfo);
    while DosError = 0 do begin
      Trovato := TRUE;
      Aggiorna(DirInfo.Name);
      FindNext(DirInfo)
    end;
    if not Trovato then begin
      Writeln(Param, ' non trovato');
      Halt(1)
    end
  end
end.
```

Figura 5 - La nostra versione di Touch.

TOUCH

Puntata un po' densa o un po' astratta, secondo i gusti. Solo il mese prossimo potremo vedere come tradurre in concreta programmazione i discorsi fin qui fatti. Intanto non me la sento di lasciarvi, dopo tante Liste e tanto LISP, senza un po' di sano Pascal. Vi propongo quindi nella figura 5 il sorgente del nostro TOUCH: solo poche righe, grazie a procedure come *GetDate* e *SetFTime*, *FindFirst* e *FindNext*, alcune tra le molte interessanti novità contenute nella ricchissima libreria dei Turbo Pascal 4.0 e 5.0.

TOUCH non fa altro che... cambiare le carte in tavola. MAKE opera le sue scelte confrontando la data e l'ora di ogni file, TOUCH rende la data e l'ora di uno o più file uguali alla data e ora in cui viene eseguito. Se ad esempio volessi ricompilare tutto il MiniMake, non solo le sue parti non aggiornate (quello che la Borland chiama BUILD), potrei dare i due comandi:

```
TOUCH *.PAS *.ASM
MMAKE MMAKE.EXE
```

Il primo rende recentissimi tutti i sorgenti, il secondo, trovando i file TPU, OBJ e EXE più vecchi, ricompila tutto da capo. Ci vediamo tra un mese. MC

Canadian Connection

per accedere direttamente al software originale americano,a libri, riviste specializzate, a prodotti specifici che in Italia non troverete mai. Microforum*, Toronto (Canada) stabilisce con voi un filo diretto in collaborazione con i migliori Rivenditori italiani.

Le principali
softhouses disponibili:
Aldus, American Small Business,
Aragon, Ashton-Tate,
Autodesk, Bedford, Borland,
Central Point Software,Concentric,
Dac Software, Dca/1-Net Communications,
Digital Research, Fifth Generation, Fox Software,
Fox & Geller, Funk, Hayes, Hercules, Hewlett-Packard,
Hitachi, Ibm, Innovative Software, Javelin, Laser Friendly, Lattice,
Lifetree, Lotus, Micro Database, Microcomputer Specialist, Micrograf,
Microrim, Microsoft, Migent, Mountain, Mouse System, Nantucket, New Nicat,
Olivetti, Osiris, Paul Mace, Pelada, Persoft,
Peter Norton, Pinetree, Prime Solutions, Quadram, Quarterdec, Samna,
Santa Cruz, Soft Logic Solutions, Software Group, Software Publishing,
Software Solutions, Spss, Stsc, Sublogic, Symantec, Toshiba, White Crane,
Word Perfect, Xerox, Xyquest, Zanthe, Z-Soft

Questi alcuni prezzi:
Xerox Ventura Publisher 2.0 1.200.000
Word Perfect 5.0 per IBM 440.000
Word Perfect 5.0 per Mac 390.000
Santa Cruz Lyrix Word Proc.Sys 5.0 860.000
NOVITÀ: PC Tools Deluxe v.5 90.000
Scheda Central Point Software
Copy II PC De Luxe 190.000
Scheda Central Point Software
Copy II PC De Luxe Enhanced 280.000
Back up laser protected.
Modem INCOMM interni da 110.000
Modem INCOMM esterni da 130.000

SOLO PER RIVENDITORI: MC

Inviatemi GRATUITAMENTE il vostro catalogo di software originale

SARANNO ACCOLTE UNICAMENTE LE RICHIESTE CORREDATE DI PARTITA IVA E NUMERO TELEFONICO.

Ditta.....
Via.....
Cap.....Città.....
Prov.....
Tel.....Fax.....
Partita IVA ☐☐☐☐☐☐☐☐☐☐☐
Firma.....

Iva esclusa.
Spedizione postale contrassegno a nostro carico.
Pagamento anticipato sconto 2%.
Sconti per quantità.
Prezzi in Lire Italiane.

Per Informazioni e ordini:

FLOPPY'S MARKET

Piazza del Popolo 5
56029 S.Croce s/Arno (Pisa)
Tel.0571/35124 Fax 0571/32768

*Microforum produce i famosi floppy disk **MITO** e **MICROMITO**