di Sergio Polini

Procedure e processi iterativi e ricorsivi

La volta scorsa ci siamo soffermati sul modulo di analisi lessicale del MiniMake. Visto che vi erano alcune piccole ma sostanziali differenze rispetto a quello di QUED, ne abbiamo approfittato per approfondire un po' l'argomento anche dal punto di vista teorico.
Il modulo di analisi sintattica non presenta invece particolari novità, è anzi anche più semplice di quello di QUED; non andremo quindi oltre la pubblicazione del listato. Inizieremo in compenso un discorso che ci introdurrà non solo all'ultima parte del programma, ma più in generale ad una delle tecniche più affascinanti e più potenti della programmazione

Tutti sappiamo cosa si intende per algoritmo, ma non sarà male un piccolo ripasso. Ricorderemo soprattutto che un algoritmo non è semplicemente una «ricetta», non è solo un insieme di regole che definiscono la sequenza di operazioni necessaria per risolvere un particolare problema. Occorre qualcos'altro.

Tanto per cominciare, quelle operazioni devono essere definite in modo chiaro e non ambiguo, ed altrettanto deve valere per la loro sequenza. Non è scontato cosa voglia dire «chiaro» o «non ambiguo»; potremmo tuttavia dire che non basta una semplice descrizione discorsiva, occorre una descrizione che sia univocamente traducibile in un dato linguaggio di programmazione (non basta che la capisca un umano, la deve capire anche quello stupido di computer). È poi necessario che, accanto a zero o più input, vi sia almeno un output. Un algoritmo deve infine rispettare alcune condizioni di finitezza: le diverse operazioni devono poter essere rappresentate ed eseguite in uno spazio e in un tempo finiti, il loro numero deve essere finito, ognuna deve essere eseguita solo un numero finito di volte. Tanto per dare un esempio, il breve programmino nella figura 1 non è un algoritmo: poiché la condizione di uscita dal ciclo non può diventare vera, l'istruzione al suo interno verrebbe eseguita un numero infinito di volte.

È proprio questo requisito di finitezza l'aspetto su cui volevo richiamare la vostra attenzione.

```
program LoopInfinito;
begin
  repeat
    Writeln('Ciao!')
  until TRUE = FALSE
end.
```

Figura 1. Un esempio di non-algoritmo.

Oggetti e azioni ricorsive

Un oggetto viene detto ricorsivo se è definito in termini di se stesso. A prima vista potrebbe sembrare che vi sia una stretta parentela tra la ricorsività e la tautologia. In realtà invece le definizioni ricorsive si rivelano estremamente potenti; basti pensare all'esteso uso che se ne fa in matematica, fin dalla definizione dei numeri naturali, dove si stabilisce, tra l'altro, che il successore di un numero naturale è a sua volta un numero naturale. La potenza della ricorsività sta proprio qui, nella possibilità di definire con un numero finito di parole un insieme infinito, di scrivere un programma con un numero finito di istruzioni in grado di compiere un numero infinito di operazioni. Un tale programma, però, non rappresenterebbe un algoritmo. Se i matematici sembra quasi non possano vivere senza perdersi nell'infinito, per noi programmatori vale proprio il contrario.

Pensiamo alla definizione di una normale lista lineare: può essere «nulla», o può essere un atomo seguito da una lista. Un atomo seguito da una lista nulla è solo un atomo, un atomo seguito da una lista contenente un atomo seguito da una lista nulla è una lista con due atomi, e così via. Possiamo vedere facilmente che quello strano oggetto che chiamiamo «lista nulla», apparentemente di nessun interesse, serve proprio a delimitare le liste non-nulle: se dicessimo che una lista è un atomo seguito da una lista avremmo una definizione non solo ricorsiva, ma anche «infinita» e quindi non implementabile in un algoritmo. Se aggiungiamo la lista nulla possiamo avere programmi in grado di manipolare liste, alla sola condizione di delimitarle con un puntatore a nil, alla lista nulla.

Analoga la situazione per una procedura ricorsiva. Questa non è altro che una procedura che chiama se stessa, come tale anch'essa potenzialmente infinita. Per essere concretamente utiliz-

zabile deve quindi contenere una condizione d'uscita che diventi prima o poi vera.

Diamo un'occhiata alla figura 2. In a) vediamo il caso più generale: la procedura P comprende, tra altri insiemi S e T di istruzioni, anche una chiamata a se stessa. In b) chiariamo che tale chiamata non è incondizionata, ma sottoposta alla falsità di una condizione d'uscita. In c) aggiungiamo che deve essere previsto un meccanismo per cui quella condizione ad ogni chiamata si avvicini sempre di più alla sua verità. Quando ciò si verificherà, si potrà finalmente uscire da P.

Procedure e processi

Una distinzione fondamentale è quella tra ciò che effettivamente avviene quando un programma viene eseguito (processo) e la descrizione che noi ne diamo (procedura). Per quanto riguarda la ricorsività, si tratta di una distinzione importante soprattutto in linguaggi come il Lisp e lo Scheme, per i quali esistono interpreti e compilatori in grado di tradurre automaticamente una certa classe di definizioni ricorsive in processi iterativi. Per chi programmi in Pascal è invece più utile distinguere tra la descrizione discorsiva di una procedura (la cosiddetta pseudocodifica) e il processo risultante, in modo da poter scegliere poi la traduzione più efficace di quella descrizione in Pascal vero e proprio.

Può infatti capitare che si proponga naturalmente una descrizione ricorsiva di un certo algoritmo, ma che si possano poi rilevare alcune caratteristiche che ne rendono facile e conveniente una implementazione iterativa.

La cosiddetta «tesi di Church» ci dice che ogni procedura che possa essere implementata da un programma può essere espressa in forma ricorsiva; da ciò segue, per inciso, che in linea di principio i programmi non hanno bisogno né di assegnazioni né di GOTO. Se vi sembra esagerato, guardate nella figura 3 come si possono descrivere formalmente la somma e il prodotto di due numeri e come se ne possa derivare una codifica in Pascal (una nota: le assegnazioni contenute nella implementazione in Pascal delle due operazioni non violano la tesi di Church, ma sono solo la conseguenza della mancanza di una istruzione return. Invece di "if n = 0 then Prod := 0", sarebbe infatti possibile scrivere "if n = 0 then return 0").

```
a)  P      -->   ((S); P; (T))
b)  P      -->   ((S); if C then exit; P; (T))
c)  P(n)   -->   ((S); if n = 0 then exit; P(n-1); (T))
```

Figura 2. Schemi di procedure ricorsive.

```
prod( 4, 3 )
somma( 4, prod( 4, 2 ) )
somma( 4, somma( 4, prod( 4, 1 ) ) )
somma( 4, somma( 4, somma( 4, prod( 4, 0 ) ) ) )
somma( 4, somma( 4, somma( 4, 0 ) ) )
somma( 4, somma( 4, 4 ) )
somma( 4, 8 )
12
```

Figura 4. Il processo ricorsivo per il calcolo di un prodotto

```
somma(x, y) =
    somma(x, 0) = x
    somma(succ(x), pred(y))

prod(x, y) =
    prod(x, 0) = 0
    prod(x, y) = somma(x, prod(x, pred(y)))

program SumProd;
var
  x, y: integer;
function Somma(a, b: integer): integer;
begin
  if b = 0 then
    Somma := a
  else
    Somma := Somma(a+1, b-1)
end;
function Prod(m, n: integer): integer;
begin
  if n = 0 then
    Prod := 0
  else
    Prod := Somma(m, Prod(m, n-1))
end;
begin
  Write('Due numeri: '); Readln(x, y);
  Writeln('Somma     : ', Somma(x, y));
  Writeln('Prodotto  : ', Prod(x, y))
end.
```

Figura 3. Calcolo ricorsivo della somma e del prodotto.

```
prod(x, y) = prod_iter(m, n, 0)

prod_iter(m, n, contatore) =
     prod_iter(m, n, accumulatore, n) = accumulatore
     prod_iter(m, n, accumulatore, contatore) =
          prod_iter(m, n, somma(accumulatore, m), succ(contatore))

program ProdIter;
var
  x, y: integer;
function Prod_Iter(m, n, Accumulatore, Contatore: integer): integer;
begin
  while Contatore <= n do begin
    Accumulatore := Accumulatore + m;
    Inc(Contatore)
  end;
  Prod_Iter := Accumulatore
end;
function Prod(m, n: integer): integer;
begin
  Prod := Prod_Iter(m, n, 0, 1)
end;
begin
  Write('Due numeri: '); Readln(x, y);
  Writeln('Prodotto  : ', Prod(x, y))
end.
```

Figura 5. Una definizione tail-ricorsiva del prodotto.

Figura 6. Pseudocodifica del calcolo del massimo comun divisore.

```
MCD(a, b), con a > b =
    se b è zero
      allora il risultato è a
    altrimenti
      il risultato è MCD(b, resto di a diviso b)
```

```
function MCD(a, b: integer): integer;
var
  resto: integer;
begin
  while b > 0 do begin
    resto := a mod b;
    a := b;
    b := resto
  end;
  MCD := a
end;
```

Figura 7. Implementazione iterativa del calcolo del MCD.

Fuori dei formalismi, la figura ci dice che la somma di due numeri è la somma del successore del primo e del predecessore del secondo, che il prodotto di due numeri è uguale alla somma del primo e del prodotto del primo per il predecessore del secondo. Come abbiamo imparato alle elementari. Nella figura 4 vediamo come apparirebbe un processo ricorsivo incaricato di moltiplicare 4 per 3. Mediante una espansione virtualmente infinita, il secondo argomento di prod viene via via decrementato mentre si tiene memoria delle somme lasciate in sospeso; grazie alla condizione d'uscita (il risultato di una moltiplicazione per zero è zero), l'espansione termina quando si arriva a prod(4, 0). A questo punto si ha una contrazione, durante la quale vengono eseguite tutte le somme.

Abbiamo qui le due caratteristiche fondamentali di un processo ricorsivo. Sappiamo bene che, quando scriviamo una istruzione del tipo Sqrt(Abs(n)), viene prima chiamata la funzione Abs e poi Sqrt; non si chiama Sqrt con un argomento ancora da valutare. Analogamente, in ognuna delle prime quattro righe non si può sommare prima di aver «espanso» prod. Quando poi inizia la contrazione, ho solo una serie di somme che vengono anch'esse eseguite partendo dalla più «interna». È questa la prima caratteristica: l'espansione ha prodotto una serie di operazioni differite.

Qui ritroviamo la potenza delle definizioni ricorsive: posso indicare in sole due righe (la definizione di prodotto nella figura 3) un insieme anche molto grande di operazioni, non infinito solo grazie alla presenza di una condizione di uscita; posso scrivere 4*5 invece di 4+4+4+4+4. L'altra caratteristica emerge da un confronto tra la definizione di prodotto nella figura 3 e il processo risultante come illustrato nella figura 4: qui abbiamo un maggior numero di righe, un maggior numero di caratteri. Non si tratta solo di considerazioni di natura tipografica: quello che nella figura 4 c'è in più deve trovare un suo posto nella memoria del computer; dalla definizione sembrerebbe sufficiente la sola memoria occupata dalle due variabili, ma in realtà perché quelle operazioni differite possano essere eseguite c'è bisogno di una memoria ausiliaria. In concreto, perché un processo ricorsivo possa svilupparsi occorre disporre di spazio sufficiente nello stack; è per questo che il Turbo Pascal mette a disposizione la direttiva M, con la quale possiamo ampliare la dimensione dello stack se le nostre chiamate ricorsive non entrano nei 16K di default.

Definizioni tail-ricorsive

Questo aggiuntivo bisogno di memoria consiglia anche di evitare quando possibile i processi ricorsivi. Posso eseguire un loop un numero grande quanto mi pare di volte, ma il numero di chiamate ricorsive è limitato dalla memoria disponibile come stack. Se riesco ad annullare il bisogno di memoria ausiliaria posso tuttavia trasformare una definizione ricorsiva in un processo iterativo; in genere si tratta solo di introdurre variabili ausiliarie.

Guardate ad esempio la figura 5. Viene proposta una definizione di prod nella quale non si fa altro che invocare una seconda procedura prodi_ter. Questa, pur essendo indubbiamente ricorsiva, descrive in realtà un processo iterativo, grazie alla presenza di due variabili ausiliare: Contatore si incarica di tenere il conto del numero di esecuzioni necessarie (in un processo ricorsivo il «conto» è tenuto nello stack: si finisce quando la fase di contrazione ci riporta al punto da cui era partita l'espansione dell'operazione di partenza in una serie di operazioni differite); in Accumulatore si accumulano progressivamente i risultati parziali, rendendo così inutile il ricorso ad una memoria ausiliaria (allo stack). Il successivo programmino in Pascal ci dimostra che il tutto può essere agevolmente realizzato mediante un processo iterativo: un normalissimo ciclo while.

Non è sempre facilissimo distinguere tra definizioni ricorsive che implicano processi ricorsivi e definizioni ricorsive che implicano processi iterativi. Quasi sempre, tuttavia, basta osservare la condizione d'uscita. Nella definizione di prod data nella figura 3, la condizione concerne un caso limite che non ha diretta attinenza con il risultato che ci attendiamo nel caso generale, intendendo per questo la moltiplicazione di due numeri diversi da zero: viene detto che, se il secondo argomento è zero, otteniamo subito zero senza bisogno di una ulteriore chiamata di prod (e può quindi terminare la fase di espansione). Nella figura 5, invece, la condizione di uscita ci offre proprio il risultato che attendiamo: se il Contatore è diventato uguale ad n, allora il risultato è contentuto in Accumulatore. Il risultato è cioè già calcolato «in coda» alla fase di espansione, la quale quindi, dal momento che non faceva altro che preparare le operazioni che avrebbero dovuto essere eseguite nella fase di contrazione, non è più necessaria. Non ho bisogno di eseguire operazioni differite per ottenere un risultato che ho già. Se proprio volessi pensare ad un processo ricorsivo, questo potrebbe arrestarsi appena giunto «in coda» alla fase di espansione; per questo motivo si parla di situazioni tail-ricorsive.

Supponiamo ora di dover tracciare un algoritmo. Nei casi banali non ho altro da fare che mettermi alla tastiera e scrivere in Pascal. Altre volte, tuttavia, è utile passare attraverso una fase di pseudocodifica: esporre in un misto di Pascal e normale italiano quello che il computer dovrà fare (prima che lo capisca lui, lo devo capire io).

Proviamo a calcolare il massimo comun divisore di due numeri. Ricorderete che, dati due numeri, il loro MCD è uguale al massimo comun divisore di altri due numeri che siano il primo il minore dei due e il secondo il resto

```
{$IFDEF Main}
program MMParser;
uses Dos, MMAlex, MMSim;
{$ELSE}
unit MMParser;

interface
uses Dos, MMAlex, MMSim;
{$ENDIF}
type
  CodiciErrore = (NoFile, IniDip, Dup, Sep, Sint, NoCmd, NoRiga, IniCmd);

{$IFDEF Main}
var
  TP: TPtr;
  SP: SPtr;
  CP: CPtr;
  DT: DateTime;
{$ELSE}
procedure Init;
procedure Parse;

implementation
{$ENDIF}
var
  Target: TPtr;

procedure Errore(Codice: CodiciErrore);
begin
  write('ERRORE');
  if Codice = NoFile then
    writeln(': Non trovato makefile')
  else begin
    write(' alla riga ', NumRiga, ': ');
    case Codice of
      IniDip: writeln('Target non valido');
      Dup   : writeln('Target gia'' definito');
      Sep   : writeln('Mancano i due punti dopo il target');
      Sint  : writeln('Source non valido');
      NoCmd : writeln('Mancano i comandi per costruire il target');
      NoRiga: writeln('Regole non separate da una riga vuota');
      IniCmd: writeln('Comando non preceduto da spazio o tab')
    end
  end;
  Halt(1)
end;
procedure Init;
begin
  Assign(Input, 'MAKEFILE');
  {$I-} Reset(Input); {$I+}
  if IOResult <> 0 then Errore(NoFile)
end;

procedure Parse;
var
  Token: char;
begin
  repeat
    ReadNextRiga;
    if FineFile then Exit;
    Token := NextToken;
    if Token <> NOMEFILE then Errore(IniDip);
    if CercaTarget(Nome) <> nil then Errore(Dup);
    Target := NuovoTarget(Nome);
    Token := NextToken;
    if Token <> DUEPUNTI then Errore(Sep);
    Token := NextToken;
    while Token = NOMEFILE do begin
      AddSource(Target, Nome);
      Token := NextToken
    end;
    if Token <> FINERIGA then Errore(Sint);
    ReadNextRiga;
    if FineFile then Errore(NoCmd);
    Token := NextToken;
    if Token = FINERIGA then Errore(NoCmd);
    while Token <> FINERIGA do begin
      if Token = NOMEFILE then Errore(NoRiga)
      else if Token <> SPAZIO then Errore(IniCmd);
      Token := NextToken;
      if Token <> RIGACMD then Errore(NoCmd)
      else begin
        AddCommand(Target, Comando);
        ReadNextRiga;
        if FineFile then Exit;
        Token := NextToken
      end
    end
  until false (* si esce per FineFile o per Errore *)
end;
{$IFDEF Main}
begin
  Init;
  Parse;
  TP := PrimoTarget;
  while TP <> nil do begin
    Write(TP^.Id^.Nome, ' dipende da:');
    SP := TP^.SourceList;
    while SP <> nil do begin
      Write(' ', SP^.Id^.Nome);
      SP := SP^.Next
    end;
    Writeln;
    Writeln('Comandi da eseguire:');
    CP := TP^.CmdList;
    while CP <> nil do begin
      Writeln(#9,CP^.Comando,' ',CP^.Argomenti);
      CP := CP^.Next
    end;
    Writeln;
    TP := TP^.Next
  end
{$ENDIF}
end.
```

Figura 8. Il modulo di analisi lessicale del MiniMake.

della divisione del più grande per il più piccolo: il MCD di 206 e 40 è uguale al MCD di 40 e 6, che è uguale al MCD di 6 e 4, e così via fino al MCD di 2 e 0, che è ovviamente 2. Il processo termina appunto quando il secondo numero è zero, nel qual caso il risultato è dato dal primo.

Prima di metterci alla tastiera proviamo a scrivere (figura 6). Sembrerebbe una definizione ricorsiva bella e buona, ma notiamo appunto che quando diventa vera la condizione d'uscita il risultato è gia bell'e pronto. Ciò vuol dire che la definizione è in realtà tail-ricorsiva, e che una soluzione iterativa è a portata di mano (figura 7).

Conclusioni

Il grande pregio delle definizioni ricorsive è presto detto: moltissime volte si tratta della via più rapida alla descrizione in pseudocodifica di algoritmi non banali. Come vedremo la volta prossima, una descrizione ricorsiva è praticamente obbligata quando si tratta di intervenire su strutture di dati esse stesse ricorsive.

Il limite dei processi ricorsivi è però quello di richiedere una memoria ausiliaria, di essere grandi consumatori di una risorsa scarsa (quello stack che sotto DOS è limitato a 64K). La regola è quindi di evitare i processi ricorsivi ogni volta che ciò appaia ragionevolmente possibile, ed il primo passo in questa direzione consiste nel cercare di pervenire a definizioni tail-ricorsive.

Questo vale come introduzione generale ad un argomento su cui mi auguro di poter tornare. Per ora l'appuntamento è per il mese prossimo, avvertendovi subito che... ci accontenteremo del processo ricorsivo suggerito dalla struttura dei dati del MiniMake. MC