di Raffaello De Masi

Ancora sulle liste; la ricursione

Restiamo un poco ancora a parlare di liste per introdurre un argomento che, fino a qualche anno fa, era dominio incontrastato di linguaggi estremamente specializzati o sofisticati, la ricursione; parleremo poi di altre specifiche proprie delle liste, quale la manipolazione, l'estrazione degli elementi, l'uso di queste nelle strutture, ecc. Si tratta di un argomento fondamentale per la comprensione del Prolog, uno dei tanti che lo rende diverso dai più convenzionali linguaggi, e che rendono questo idioma così utile per la soluzione di problemi non squisitamente numerici. Si tratta di un argomento fondamentale, che, unitamente a quanto già detto nella passata puntata, consentirà al programmatore in Prolog di raggiungere traguardi che ben difficilmente sarebbe agevole superare, anche con un linguaggio dedicato come il Lisp. Il vero vantaggio del Prolog su quest'ultimo è, infatti, la maggiore vicinanza con il pensare umano, che rende la programmazione e la lettura dei listati molto più facile, efficace e rapida

La ricursione

La migliore definizione di ricursione è, secondo me, quella espressa da Dan Shafer in Turbo Prolog Pr. (opera già diverse volte citata) e così riassumibile:

ricursione - vedi ricursione

A parte la forzatura insita nella definizione, è chiaro il principio; la ricursione è una procedura che chiama se stessa per la sua soluzione. È come se si dicesse che le botte sono quelle cose che si prendono quando si hanno le botte, o che l'auto è quell'oggetto che si usa quando si va in auto!

Formalmente la ricursione (ricorsività), in Prolog è una regola che ripetutamente invoca se stessa per eseguire una operazione nell'ambito del programma. L'idea di ricursione è una delle più potenti ed efficienti nella programmazione moderna, e, in Prolog in particolare, è opportuno che il programmatore divenga immediatamente e rapidamente pratico di essa. Per quanto attiene in particolare agli argomenti oggetto di discussione in questa puntata, una procedura ricorsiva scorrerà ripetutamente una lista, modificando ogni volta i dati in essa contenuti in qualche modo. La procedura, in altri termini, è applicata, volta per volta, ai dati modificati fino a giungere ad una soluzione.

Vediamo di seguito un esempio di procedura ricorsiva applicata ad una piccola base di conoscenza. La figura 1 mostra una gerarchia familiare con tredici componenti. Le relazioni di base tra di essi possono essere così riassunte:

```
Domains
  persona = symbol
Predicates
  madre(persona,persona)
  padre(persona,persona)
  nonno(persona,persona)
Clauses
  madre(maria,sara).
  madre(elena,sara).
  madre(biagio,sara).
  madre(donato,maria).
  madre(enrico,maria).
  madre(francesco,elena).
  madre(gina,elena).
  madre(elisa,teresa).
  padre(maria,teodoro).
  padre(elena, teodoro).
  padre(biagio,teodoro).
  padre(donato,antonio).
  padre(enrico,antonio).
  padre(francesco,davide).
  padre(gina,davide).
  padre(elisa,biagio).
  nonno(Discendente,Persona) if
    madre(Discendente,Persona) or
    padre(Discendente,Persona).
```

Che ve ne pare? chiaro, no? anche tenendo conto della figura! È ovvio che i goal destinati all'esplorazione di questo pur piccolo database saranno incentrati su quesiti come «madre(Xxxxx,Yyyyy)» aut similia,; tutto risponde perfettamente ai canoni che alimentano e pilotano un linguaggio di intelligenza artificiale, ma la ricursione dov'è?

Parleremo di ricursione definendo un nuovo predicato-goal, «antenato», per definire qualunque persona che, nella scala delle natività precede qualcun altro. Bisognerà quindi stabilire una regola in base alla quale una persona specifica è un antenato di un'altra.

Certo è che ciò che è chiaro in lingua italiana può essere non chiaro per un calcolatore. Ad esempio, in lingua parlata, un antenato è, come abbiamo detto prima, una persona di una generazione passata; non è questo un modo possibile di definire una regola efficiente in Prolog; un approccio migliore è quello

di definire «un antenato come un padre, o una madre, o un antenato di essi».

Sia ben chiaro quello che abbiamo appena detto, in quanto è di fondamentale importanza per quanto abbiamo enunciato circa la ricursione; in termini semplici dobbiamo definire una regola in cui diciamo a chiare lettere che antenato significa padre o madre; se ciò non avviene occorre ritentare applicando la a priori; invece essa si adatta alla perfezione ad ambienti di conoscenza in continua evoluzione. Tanto per esemplificare su quello già esistente, se Gina si sposasse e avesse figli, una semplice aggiunta di una regola (o di un elemento di conoscenza alla lista) porterebbe ad ampliare l'area di scansione della regola ricorsiva.

Ovviamente avremo tutti capito che

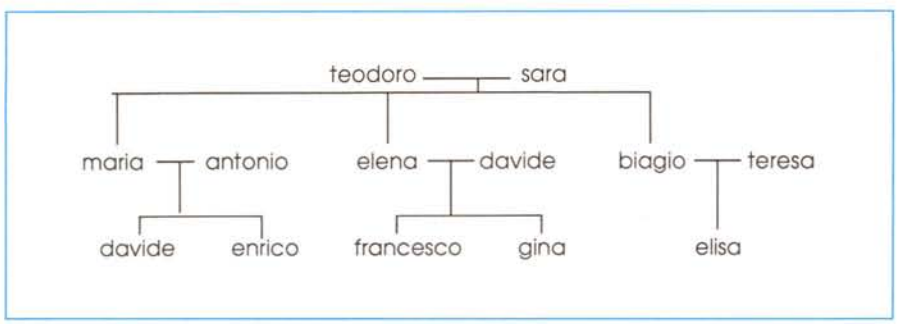

Figura 1 - Una gerarchia familiare utilizzata nell'esempio del testo.

stessa definizione (di antenato) e regola ai termini chè non sono né padre né madre.

Occorre, quindi fornire due condizioni alla definizione di «antenato», per la esatta chiarificazione di questo termine; e qui entra in gioco la ricursione; avremo pertanto due predicati; il primo che stabilisce che è un genitore una persona che vien primo nella scheda gerarchica, il secondo che è genitore «una persona che è genitore di un'altra». Vediamone lo sviluppo sotto il punto di vista degli statement.

La prima regola sarà:

```
antenato(Figlio,Persona) if
  genitore(Figlio,Persona).
```

La seconda legherà questa al concetto di «antenato», nella forma:

```
antenato(Figlio,Persona) if
  genitore(Figlio,Persona) and
  antenato(Figlio,Persona).
```

Se ci pensiamo appena appena un poco, vediamo la chiarezza del costrutto; in altri termini è come se avessimo detto al programma: «Se una persona è diretta figlia di un'altra, quest'ultima è un antenato; se ciò non accade riprovare vedendo se quest'ultima è figlia di un'altra, fino a soddisfare la tesi o a esaurire il DB». Ma la cosa più entusiasmante della ricursione in Prolog è che essa non si limita a situazioni programmatorie in cui le informazioni sono note la seconda definizione della regola appena espressa è ricorsiva. Così basterà battere il goal:

```
Goal: antenato(enrico,Chi).
Chi = maria
Chi = antonio
Chi = maria
Chi = maria
4 Solutions
Goal:
```

Illustriamone un momento il procedimento; al primo passaggio attraverso la base di conoscenza Prolog istanzia Figlio a [enrico]. Poi passa alla regola successiva obbedendo, come al solito, al subgoal «antenato(enrico,Persona)». Prolog tiene sempre conto del punto in cui si trova nella base di conoscenza, e lancia la regola «genitore(Xx,Yy)». Il risultato è un nuovo subgoal, «madre(enrico,Persona)», che ricerca ancora fino a istanziare persona a [maria]. Finalmente ottenuto il primo risultato, restituisce il frutto della ricerca e riprende il processo stavolta per testare la regola «padre(Xx,Yy)» riportando ancora una volta i risultati.

Ma non è finito, visto che con questa regola si è provveduto a risolvere solo le ricorrenze dirette della regola immediata legata a «genitore»; ci si ritrova, invece, con la seconda parte della seconda regola, che impone di nuovo di adottare la stessa procedura; si entra quindi nella base di conoscenza con la variabile «Figlio» istanziata a [enrico] e le altre variabili non istanziate.

Ricomincia di nuovo il ciclo, con le stesse regole e limitazioni precedenti, ma con una variabile già istanziata, che però è del tutto insensibile a variazioni dipendenti dalla mobilità nell'interno del database. È questa la vera marcia in più della ricursione, ed è su questa proprietà che si baseranno le nostre successive esperienze.

Termina così, anche questa volta, la nostra discussione, avendo, speriamo, acquisito un tool di potenza davvero eccezionale. Se si tien conto che il Turbo Basic della Borland (e successivamente il Quick C della Microsoft hanno fatto davvero il loro maggior salto di qualità adottando la ricursione (ne parliamo nel riquadro a fianco), ci renderemo davvero conto di quanto, con una accorta programmazione, sia possibile rendere compatti ed efficienti i programmi da noi scritti. Non a caso Crunch! un eccellente spreadsheet per Macintosh, è stato scritto in Lisp adottando (è il suo campo ideale) la ricursione specie per risolvere quella che forse è la vera croce di questo tipo di applicazioni, i riferimenti circolari. Ma questa è un'altra storia!

Tutti i linguaggi Borland ammettono la ricursione, e traiamo spunto da questo fatto per parlare di questo potente tool programmatorio riportando una considerazione sulla ricorsività utilizzando un linguaggio senz'altro più facile da intuire e da «leggere» del nostro Prolog, il Turbo Basic; anche chi, provenendo dal Basic, si trovasse con il nostro articolo di fronte a una certa difficoltà nel comprendere l'intrinseca potenza di questa struttura potrà leggere quanto diciamo di seguito e poi applicare gli stessi concetti al Prolog, senz'altro, a livello di lettura di un semplice listato, molto più complesso da maneggiare.

TB (e anche Quick Basic) ammette la ricursione, finora relegata nelle nascoste oscurità di certi linguaggi dedicati, come C, Pascal e Forth. Quanto si è detto a proposito di essa! La definizione più spiritosa ricordo di averla letta qualche anno fa su una rivista inglese, dove, come esempio di discorso ricursivo, veniva proposta la frase «La pioggia è quella cosa che cade quando piove». Scherzi a parte, se è merito di Borland di aver cavato dalla nebbia questo potente processo, diamo a Cesare quello che è di Cesare e vediamo in che cosa questa consiste, utilizzando un esempio contenuto nel manuale stesso del TB.

Si abbia il seguente programma, anzi

per essere precisi procedura (o funzione), che adotta una tecnica ricursiva:

```
DEF FNFattoriale#(n%)
 if n% > 1 AND n% <= 170 THEN
  FNFattoriale# = n% * FNFattoriale#(n%-1)
 ELSEIF
 n% = 0 OR n% = 1 THEN
  FNFattoriale# = 1
 ELSE
  FNFattoriale# = -1
 END IF
END DEF
```

A parte la brevità del codice, la struttura si presenta complessa, anche più di quanto sembra. Per renderci conto di come funziona una procedura ricorsiva, vediamo come essa fa a calcolare il fattoriale di 3.

Battiamo in modo immediato l'ordine:

PRINT FNFattoriale#(3)

Il valore 3 viene passato per valore alla funzione. Questa (si segua il listato) controlla se l'argomento è maggiore di 1 e minore od eguale a 170. Superato il controllo si tenta di eseguire l'operazione:

FNFattoriale# = 3 * FNFattoriale#(2)

A questo punto il programma si trova in difficoltà in quanto gli viene chiesto di considerare come valore definito [FNFattoriale#(2)] qualcosa di cui non conosce invece il valore. Allora si sospende momentaneamente l'elaborazione ed il programma si applica alla soluzione di FNFattoriale#(2).

Si ricomincia daccapo, e la funzione FNFattoriale# tenta di passare il valore 2. Superato il controllo del >O e <= 170 ci si ritrova di nuovo di fronte alla chiamata ricorsiva della funzione, che ancora una volta non può essere risolta immediatamente. Anche questa DF viene messa in lista d'attesa e FNFattoriale# viene chiamato per la terza volta, stavolta con valore 1.

Il loop a questo punto si blocca, in quanto c'è la trappola del secondo THEN (n% = 1). FNFattoriale# può terminare la sua corsa all'indietro e richiamare l'ultima messa da parte [FNFattoriale#(2)] per poterla eseguire. Viene così ripercorsa, in senso contrario, la scala delle istruzioni accantonate, fino all'esaurimento dell'argomento della funzione principale. La struttura viene completata ed il risultato (6) stampato sullo schermo.

Le tecniche di ricursione, se correttamente usate, snelliscono molto il lavoro programmatorio, anche se talvolta non è facile decidere se è il caso di fare o no a loro ricorso. C'è da dire, come appare evidente nell'analisi appena eseguita, che occorre dello spazio, da qualche parte (per essere precisi, nello stack) in cui accantonare le operazioni sospese. Ogni linguaggio ha tool a disposizione dell'utente (o automatici) per dimensionare opportunamente la grandezza dello stack alle esigenze del programma stesso. MC