di Raffaello De Masi

La manipolazione delle informazioni

Cammina cammina stiamo giungendo al termine delle nostre trattazioni. Era però giocoforza essere un poco più chiari su una delle strutture più importanti del linguaggio, i database e la relativa manipolazione. Trattandosi di un idioma il cui costrutto si basa soprattutto sulla manipolazione di informazioni non poteva essere che così. Diamo perciò una rapida occhiata agli operatori che permettono di usare nella maniera più efficiente il database stesso

La prima cosa da fare, nella creazione e nella manipolazione di un database, è, mi si perdoni la boutade, crearlo: proprio per rendere più chiaro il tutto immaginiamo di creare una base di dati relativa ad un gruppo di impiegati. Poiché, come abbiamo visto la volta scorsa, conviene, data la potenzialità del linguaggio, adottare tecniche relazionali per il database, converrà dividere le informazioni relative in due tabelle, che hanno, ovviamente, due dichiarazioni diverse, ad esempio è possibile crearne una che comprende nome, numero e salario; un'altra invece conterrà le funzioni dell'impiegato e sarà anch'esso conservato in ordine di numero, per consentire un facile riferimento alle tecniche relazionali. È da notare che dichiarazioni di database somigliano, in maniera molto vicina, a predicati e fatti, essendo organizzati, come essi, in forma di liste di argomenti.

Una base di dati viene dichiarata attraverso il predicato [database]; successivamente i predicati dichiarati possono essere usati esattamente allo stesso modo di quelli visti alcune puntate fa, ma ad essi è consentita anche la possibilità di inserire e rimuovere fatti dalla base di conoscenza purché, ovviamente, essi siano del formato ammissibile dalla base stessa.

Grande utilità nella creazione di una base di dati ha il dominio [dbasedom] (che il linguaggio vede come un identificatore invisibile). Ogni volta che si definisce un database in Turbo Prolog, con la struttura propria di un database, il linguaggio crea un nuovo dominio, invisibile, riferito, appunto a dbasedom. Sebbene non sia stato mai dichiarato, questo dominio esiste, esattamente come uno esplicitamente dichiarato; un esempio di dichiarazione di dbasedom potrebbe essere:

```
domains
dbasedom = impiegati (nome, numero, salario); funzioni (numero, impiego)
```

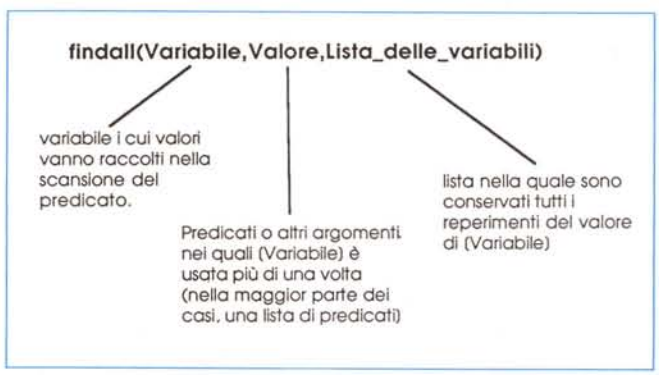
La dichiarazione così eseguita rende possibile usare i domini compresi tra parentesi, come se fossero stati effettivamente dichiarati nel programma.

Usare il database è non molto complicato (visto che le maggiori funzioni di relazionalità vengono svolte direttamente dal linguaggio); vediamo come vanno usate le due tecniche, rispettivamente di inserimento di dati dalla struttura e di richiamo dalla stessa di informazioni desiderate. Uno dei tool più importanti nella manipolazione dei database è il predicato [asserta], e il suo complementare [assertz]. Questi predicati servono ad aggiungere nuove conoscenze, fatti, alla base di conoscenza creata con Turbo Prolog.

Il loro costrutto è molto semplice, visto che manipolano un solo argomento, che deve essere organizzato secondo il pattern definito al momento della dichiarazione di costruzione del database.

La differenza tra i due predicati e tra le rispettive funzioni è abbastanza netta; [asserta] pone i dati da esso manipolati all'inizio della lista del database, [assertz] alla fine (tanto per agganciare il tutto a una regola mnemonica, la a e la z finali consentono di ricordare immediatamente, con analogia all'alfabeto, la posizione in cui viene sistemata l'informazione).

Passiamo adesso a definire un nuovo predicato, [findall]. Questo predicato esegue una ricerca totale attraverso una intera collezione di fatti in una base di conoscenza, e, ogni volta che la ricerca ha successo, il programma pone un



determinato valore di variabile in una lista, per un successivo confronto e uso. Questo predicato, del tutto nuovo, utilizza tre argomenti, aventi le funzioni illustrate in figura.

Abbiamo detto come fare a inserire dati e valori in un database; vediamo ora come è possibile toglierli; quanto diremo è valido sia per database residenti su file sia per strutture di dati in memoria. Alla bisogna assolve il predicato [retract] che, nella forma

```
retract (lista _ nome,numero _)
```

permette, nel nostro caso di cancellare l'informazione relativa al salario (si noti l'uso della variabile dummy [_]). Ovviamente il predicato [retract] non serve solo a cancellare definitivamente fatti e dati; un uso più logico è quello di richiamare valori per correggerli e successivamente reinserirli. L'uso più logico è quello di manipolare un database basato su disco da cui prelevare e su cui depositare dati.

Ultimo argomento di cui tratteremo oggi, argomento che riveste una grande importanza che per forza di cose dovremo prolungare alla prossima puntata, prevedibilmente l'ultima o la penultima; il controllo del flusso del programma.

Questo argomento richiede un trattamento completamente nuovo e un approccio del tutto diverso dagli altri linguaggi procedurali a causa della differente natura specifica del linguaggio Prolog. Possiamo senz'altro dire che le tecniche di controllo Prolog dipendono dal corretto uso e gestione del procedimento di backtracking, specifico ed unico di questo idioma.

A causa del ruolo chiave giocato dalla tecnica di backtracking nella programmazione, è opportuno avere ben chiaro in mente il principio della diretta interconnessione esistente tra backtracking e gestione delle informazioni; la base di lancio del BT è rappresentata dal fatto che, nella maggior parte dei casi il programma è chiamato a giudicare e leggere più di una regola o di un fatto, relativo alle operazioni che sta seguendo. Nella gestione e nell'esame dei fatti Prolog usa dei marker per verificare se e come le regole siano state verificate o no. Il principio è quello della ricerca per ramificazioni successive. Quando un ramo è completamente verificato, il monitor di flusso ritorna indietro fino all'ultima ramificazione e prosegue la sua ricerca in altra direzione. La tecnica di BT è il mezzo più efficace e facile per utilizzare in modo non procedurale il linguaggio; il processo di avanzare per soluzioni successive ai goal e di cambiare direzione è del tutto automatico e trasparente all'utente. Senza scendere in esemplificazioni, sempre piuttosto complesse, possiamo solo dire che la stessa tecnica di analisi, eseguita con un altro linguaggio, imporrebbe l'adozione di codici molto più lunghi, più complicati e meno leggibili.

In effetti la tecnica di BT, ridotta all'osso, si basa su sue assunti; quello di forzare un particolare goal per verificarne eventuali errori, e quello di interrompere un goal o una regola prima che un processo passi completamente attraverso il BT. Come si sa fin dall'inizio, i mattoni di base dell'analisi eseguita da programmi redatti in Prolog sono i goal. La semplicità del procedere è basato sul fatto che ogni goal proposto può essere coronato o meno da successo. Ma cosa significa avere o meno successo? Un goal ha successo quando tutte le condizioni che esso impone sono soddisfatte, vale a dire che risponde alle esigenze ritrovate nella base di conoscenza o desumibili da essa attraverso schemi logici. Il punto chiave da tener presente nella gestione del BT è che quando un goal viene soddisfatto, questo (ed eventualmente quelli che da esso dipendono) raggiunge il suo scopo e il programma si ferma. Esistono goal che sono sempre soddisfatti; spesso questo dipende dai predicati che questi goal manipolano; un esempio è [nl] che non richiede argomenti, o [text], che può avere uno o più argomenti. Questi goal hanno, quindi successo qualunque sia il contenuto della base di conoscenza nel cui ambiente stiamo lavorando.

Altri goal possono essere soddisfatti una volta sola; è il caso di [write]. In questo caso non c'è backtracking in quanto l'esaudimento del goal termina l'operazione a tutti gli effetti.

Al contrario, nella maggior parte dei casi, esistono implicitamente le possibilità che un goal non sia soddisfatto o fallisca. Questo può avvenire a causa di una non accurata organizzazione delle regole, o ancora perché il sistema non ha abbastanza informazioni per soddisfarlo. In altre parole, un goal può fallire perché è costruito in modo cattivo o perché il sistema non capisce.

Qualunque sia la situazione per cui si verifica un errore o un mancamento del risultato, l'elaborazione non si ferma ma, specifico del linguaggio, si instaura un adeguato procedimento di backtracking. Questa operazione, del tutto automatica, può essere d'altro canto provocata direttamente dal programmatore (o anche dall'utente) invocando il predicato [fail]. Il significato e l'uso del predicato sono ovvi, esso forza Prolog ad eseguire un BT forzato per cercare di soddisfare un precedente goal.

Ci sono fondamentalmente due occasioni per usare il predicato [fail]; la prima, in combinazione con il predicato [cut], consente la creazione di potenti meccanismi di controllo; la seconda consente di forzare il backtracking quando si desidera leggere tutto il database per ricavare tutte le possibili risposte a un goal. Prolog 86 ed altri dialetti del linguaggio includono un altro predicato, assente in Turbo Prolog, [retractall], che può essere simulato nel nostro con un opportuno uso di [retract] e di [fail]. Purtroppo [retract] è un goal che non può essere risoddisfatto, vale a dire che con esso TurboProlog cancella solo il primo reperimento del fatto ricercato.

Infine qualche parola sul predicato [cut]. Si tratta di un operatore che esegue il provvedimento inverso di [fail], vale a dire che ferma le operazioni di backtracking. Il significato e l'uso non sono difficili da comprendere, ma diventa difficile capire quando sia il caso di usarlo. Cercheremo di spiegarci con un esempio. Un uso appropriato del [cut] lo si ha quando Prolog ha trovato la regola corretta da applicare e quindi si desidera evitare che il flusso di programma si disperda in operazioni di backtracking inutili. L'altro uso più diffuso è quello di adottare un [cut] per fermare il BT quando, sapendo che a un goal esiste una sola soluzione, questa soluzione è stata già trovata dal programma.

Usare cut e fail in combinazione permette di maneggiare situazioni molto complesse dove un numero anche elevato di possibili scelte possono applicarsi a una determinata situazione. [cut] in questo caso ha la funzione soprattutto di evitare lunghe e noiose peregrinazioni del flusso del programma in database piuttosto estesi, quando la soluzione è stata trovata (un esempio può essere la gestione dei conti correnti di una banca, dove è prevedibile che la combinazione nome-numero di conto sia una e una sola per ogni correntista).

Anche stavolta abbiamo terminato; ci resta solo da discutere delle tecniche di debug e di quelle di compilazione; a risentirci la prossima volta. MC