di Raffaello De Masi

Ancora sulle liste
L'uso dei predicati

Adesso che siamo padroni della ricursione e delle sue tecniche, siamo pronti a metterla in azione per sperimentare, in particolare, questa sofisticata particolarità di questo strano linguaggio nella manipolazione delle stesse liste, così come avevamo pensato di fare la volta scorsa; il primo passo da fare è quello di definire predicati capaci di lavorare con le liste stesse

C'è da fare a questo punto una piccola precisazione, visto che mai come qui il Turbo Prolog differisce dai più convenzionali Prolog di altri produttori. Questo in quanto bisogna considerare che Turbo Prolog richiede come vedemmo nelle prime puntate, la definizione dei tipi di [dominio] per ogni elemento, anche se abbiamo già visto occasionalmente che è possibile eseguire «forzature» destinate a bypassare l'estenuante trafila definizione-istanziazione-uso della variabile. Inoltre, proprio a causa della particolarità degli elementi [tipo], non è possibile semplicemente assegnare una lista ad un valore di variabile (ancorché in forma di array-matrice), e poi immediatamente lavorare con tale lista stessa. Dobbiamo invece esplicitamente definire i contenuti della lista mentre siamo al lavoro con lo stesso programma, nella finestra «Dialog» del linguaggio, direttamente e senza la fase intermedia della sequela delle definizioni.

Questo è indubbiamente un vantaggio se si tien conto che Prolog aborrisce le pastoie tipiche del Pascal e suoi derivati (sotto questo punto di vista il Nostro è, volendo, altrettanto confusionario del Basic o del Fortran), ma può divenire una trappola per chi appena appena dimentica la dualità dell'uso delle liste e la sua disponibilità a essere definita. Con queste premesse bene in mente vedremo in questa puntata la tecnica dell'uso dei predicati nella definizione e manipolazione delle liste.

L'uso dei predicati nella manipolazione delle liste

Quasi dappertutto nella programmazione di applicazioni Turbo Prolog occorre presto o tardi eseguire una scansione di una lista per cercare, nella base di conoscenza, un termine particolare. Ad esempio, avendo una lista di numeri di identificazione di impiegati come questa:

impiegati (333, 444, 555, 666, 77, 888, 999, 99)

può essere utile possedere un predicato capace di ricercare attraverso una base di conoscenza per trovare se un numero di identificazione di un impiegato (ad esempio 999) è un numero presente nella lista [...] definita. La cosa può essere agevolmente risolta attraverso un semplice uso di una struttura ricorsiva, dal significativo titolo [membro]. L'intera struttura di ricerca, rappresentata comunque da un solo rigo di programma è esemplificabile come segue:

```
Goal:
membro (999, [333, 444, 555, 666, 77, 888, 999, 99]).
True
1 Solution
Goal:
```

Il predicato [membro] appena definito richiede due argomenti; il primo è rappresentato dall'elemento cercato nella lista, il secondo è la lista stessa (e non può essere rappresentato da una sola variabile).

Ricordate quanto abbiamo detto qualche puntata fa circa la struttura di una lista (testa, coda, ecc.)? È giunto il momento di mettere a frutto quanto vedemmo allora. Poiché la lista è appunto vista come una testa e una coda, esistono due possibilità che l'elemento sia ritenuto appartenente alla lista definita; che cioè sia rappresentato dalla testa o dalla coda, appunto. Ma, e qui entra in gioco la ricursione, la coda è essa stessa una lista; per eseguire una corretta ricerca occorrerebbe quindi rieseguire una scansione della lista al completo. La cosa più semplice è quella di definire una funzione ricorsiva, in Turbo Prolog, del tipo:

```
membro (Oggetto, [oggetto|_]).
membro (Oggetto, [_|Coda])      if
membro (Oggetto, Coda).
```

La prima regola indica che se Oggetto è il primo termine della lista, allora esso fa parte della lista stessa, è quello che noi cerchiamo, e l'elaborazione può sospendersi. La seconda clausola usa una notazione tipica delle liste abbondantemente condite di ricursione dicendo, in linguaggio parlato: «Oggetto fa parte della lista se esso fa parte della lista». La cosa non è stupida quanto sembra se si tiene conto che innanzi tutto si esamina la testa della lista, successivamente si esamina la istanziazione (inizializzazione, ricordate?) della variabile Coda, che, se riconosciuta come una lista, viene sottoposta a trattamento d'esame, non solo, ma testata per accertare se è composta da zero o più elementi.

Senza ricursione il predicato Membro non potrebbe essere definito in quanto non è detto che si conosca effettivamente quanto sia lunga la lista stessa.

Un curiosità; l'uso del predicato Membro può essere in pratica limitato solo dalle regole contenute nella base di conoscenza. Così esso può essere adeguatamente utilizzato in una regola in modo che possa essere altrettanto utile se la lista è molto lunga oppure il suo contenuto e la sua estensione è sconosciuto al lancio del programma (è il caso di liste depositate su file di dati che vengono caricati dal programma stesso e che possono essere aggiornate durante le successive sedute di programma). Un esempio dell'uso di Membro, su una lista di grandezza e caratteristiche sconosciute è:

```
domains
  alista = integer
predicates
  risposta
  membro(integer,alista)
clauses
  risposta    if
      readint(Numero) and
      membro(Numero[1,2,3,4,5,6,7])    and
      write(Numero),nl
  membro(X,[X|_]).
  membro(X[_|Y]) if
      membro(X,Y).
```

Lo scopo del programma è abbastanza chiaro (alcune istruzioni di I/O, ancora non note sono peraltro abbastanza intuibili). Esso funziona in questo modo: al lancio, alla chiamata Goal (si ricordi,

ancora una volta, che esso può essere definito anche durante la stesura stessa del programma) battere la parola «risposta» e premere RETURN. Il cursore inizierà a lampeggiare nella finestra di Dialogo (a causa della richiesta di «readint», che chiede di istanziare da tastiera il valore di Numero); battendo un numero intero, il programma risponderà affermativamente se esso è contenuto nella lista [1, 2, 3, 4, 5, 6, 7].

La cosa appare di estrema utilità, come dicevamo, nella scansione di basi di dati piuttosto complesse e articolate; ricerche efficienti e pilotate possono essere eseguite adottando sofisticate combinazioni di operatori logici; questo è di grande utilità ad esempio, in sistemi esperti dove occorre ricercare fattori concomitanti per definire, ad esempio, una diagnosi di una malattia; tanto per non autoincensarsi, ho utilizzato una semplice struttura come quelle presentate nell'esempio, ma arricchita e articolata dalla combinazione oculata di operatori logici, per mettere a punto una sofisticata ricerca dei fattori inquinanti nelle acque di alcuni pozzi perforati nella provincia di Avellino.

L'estrazione di un elemento da una lista

Come abbiamo detto fino alla noia, una lista è formata da una testa (Head) e da una coda (Tail). La particolare notazione per tali sottodivisioni, da parte di Turbo Prolog è mostrata nel programma successivo. Definiremo un dominio che crea una lista contenente le lettere a, b, c, d. Una semplice tecnica di estrazione potrebbe essere rappresentata da:

```
domains
  lista = char
predicates
  test (lista)
clauses
  test (['a', 'b', 'c', 'd']).
```

Un esempio di estrazione di parti di lista sarebbe:

```
Goal: test ([Testa|Coda])
Testa = a, Coda = [b, c, d]
1 Solution
Goal:
```

Che cosa succede? Semplice, abbiamo indicato al sistema di definire la Testa e la Coda e di assegnare queste parti alle variabili aventi lo stesso nome; dimenticavamo di dire che il segno [|], che su alcuni monitor è rappresentato da un [:] con i simboli allungati, indica al sistema che intendiamo lavorare sulle due parti diverse della lista stessa, in altri termini che desideriamo accedere alla Testa e alla Coda (si consideri in tal senso anche l'analogo segno presente nell'esempio precedente).

Per chiamare solo la Testa, faremo uso di una variabile anonima (la ricordate?): scriveremo:

```
Goal: test ([Testa|_]).
Test = a
1 Solution
Goal:
```

e ovviamente la stessa cosa avviene per la Coda:

```
Goal: test([_|Coda]).
Coda = [b, c, d]
1 Solution
Goal:
```

un uso più raffinato delle variabili anonime può essere visto nel programmino successivo:

```
Goal: test([_|_|Coda]).
Coda = [c, d]
1 Solution
Goal:
```

Per essere chiari fino in fondo, Turbo Prolog esegue una istanziazione delle due variabili anonime, e successivamente legge, come Coda, le altre. Come risultato, la cosa stessa è rappresentata dagli ultimi due elementi della lista di quattro.

Tipologia di struttura includente il separatore (barra verticale [|], ASCII 124).

Alterazione delle liste

Uno degli aspetti più utili del Prolog è la possibilità di intervenire sulle liste in maniera dinamica. Il contenuto di una lista può essere sottoposto a modifica, riduzione o aumento del numero dei suoi membri grazie alla sua natura dinamica anche durante l'esecuzione di un programma. Per far ciò Turbo, come altri dialetti, consente di creare e usare predicati che aggiungono nuovi termini alla lista, a modificare l'ordine e la sequenza con cui sono organizzati, e molte altre cose. Vediamo di seguito brevemente come è possibile accedere e usare questi predicati.

La struttura principale di Prolog, dedicata, come è noto, a problemi di programmazione di sistemi esperti e intelligenza artificiale impone la necessità di accedere e modificare continuamente basi di dati anche enormi.

Questo continuo aggiungere modificare e togliere dati non è altro, comunque, che il meccanismo di base per l'acquisizione della conoscenza anche nel campo umano.

Esiste un predicato, fondamentale, che va sotto il termine di [append]. Il suo uso è finalizzato appunto all'aggiunta di informazioni a una lista. L'uso del predicato è abbastanza semplice e può essere così riassunto in un esempio:

```
Goal: append ([x, y, z], [a, b, c], Lista).
Lista = [k, y, z, a, b, c]
1 Solution
Goal:
```

La seconda lista è stata agganciata alla prima; il terzo argomento, che vediamo nella definizione del Goal, rappresenta una variabile che è istanziata al valore della lista ottenuta dalla fusione.

Una tecnica molto utilizzata è quella di rappresentare la prima lista (o la seconda, invertendo il ragionamento) con un solo elemento, cosa che permette di aggiungere un singolo valore a una lista già esistente.

Ma chi ci ha seguito con attenzione fino ad adesso dirà: «Bene, accedere al primo elemento della lista è semplice, ma come si fa ad accedere, ad esempio, all'ultimo?». Un metodo per giungere a ciò senza eseguire la scansione di tutta la lista è quello di rovesciare il contenuto dell'intera lista stessa, e di accedere, con i normali mezzi al primo elemento di questa nuova lista, ricavata dall'inversione della precedente.

L'esempio riportato di seguito indica l'uso del predicato:

```
Goal: reverse([a, b, c, d, e, f, g, h],Lista).
Lista = [h, g, f, e, d, c, b, a]
1 Solution
Goal:
```

È possibile ancora approntare altri predicati destinati a manipolare le liste usufruendo dei predicati già presenti nella lista comandi del dialetto utilizzato. Per questo è opportuno rifarsi a quanto descritto nei relativi manuali d'istruzione. È giunto quindi il momento di usare le liste e vederne la relativa funzione. È quanto descriveremo in dettaglio la volta prossima.