

Furlan F. & Lanzarone G.A.

PROLOG

**Linguaggio e metodologia di
programmazione logica**

Sommario

Introduzione al Prolog.....	7
Introduzione a Prolog come linguaggio di programmazione.....	7
Metodologia di programmazione logica.	7
Risoluzione logica di problemi.	8
1. Rappresentazione di un problema.....	9
Oggetti.....	9
Variabili	11
Termini.....	12
Operatori	13
Liste.....	13
Relazioni	17
Proposizioni	20
Programmi Prolog	23
Note bibliografiche	23
Sommario	24
2. Interpretazione dichiarativa.....	25
Programmazione dichiarativa.	25
Interpretazione formale ed informale.....	26
Risoluzione.....	28
Quesiti.	29
Unificazione.....	35
Regole di unificazione.	37
Verifica di occorrenza nell'unificazione.	38
Note bibliografiche.	40
3. Interpretazione procedurale.....	41
Clausole e procedure.....	41
Esecuzione delle procedure.....	44
Regole di selezione e di ricerca.....	44
Alberi di ricerca e di dimostrazione.	45
Strategia standard di Prolog.	49
Funzionamento dell'interprete Prolog.	52
Conseguenze della strategia di Prolog sull'efficienza e sulla terminazione.....	53
Implementazione della strategia di Prolog.....	57
Predicati predefiniti ed effetti collaterali.	58
Note bibliografiche.	59
Sommario.	59
4. Utilizzo del sistema Prolog	60
L'interazione utente-sistema.....	60
L'accesso all'interprete.	61
Il file di sistema "user".....	63
Quesiti e risposte.....	64
Terminazione della sessione.	69
Note bibliografiche.	70
Sommario.	70
5. Strutturazione del controllo.....	71
La ricorsione.	71
Ricorsione e Induzione.....	72

Comportamento di clausole ricorsive	73
Efficienza e terminazione nelle definizioni ricorsive	75
Esecuzione in avanti ed all'indietro di procedure ricorsive.....	77
Ricorsione multipla	79
Procedure con più clausole ricorsive	81
Ricorsione in coda.....	82
Ottimizzazione della ricorsione in coda.....	83
Ricorsione mutua	83
Possibili cicli infiniti nella ricorsione	84
Osservazioni sulla ricorsione	87
Sequenza	88
Selezione	92
Iterazione.....	95
Iterazione per ritorno indietro	95
Iterazione come ricorsione in coda	97
Iterazione con il costrutto "repeat".....	98
Note bibliografiche	99
Sommario	99
6. Controllo del ritorno indietro	100
Il predicato predefinito di taglio.....	100
Il taglio nelle strutture di selezione.....	104
Un costrutto di tipo "if-then-else".....	108
Il taglio nelle strutture ripetitive.....	110
Determinismo dichiarativo e procedurale	115
Superamento tendenziale dell'uso del taglio	119
Note bibliografiche	120
Sommario	120
7. Strutturazione dei dati	121
Rappresentazione di dati con relazioni e con termini	121
Indicizzazione delle clausole	122
Strutturazione dei termini.....	127
Rappresentazione ennaria e binaria di relazioni	128
Astrazione procedurale e astrazione sui dati.....	132
Note bibliografiche	135
Sommario	136
8. Sviluppo, documentazione, riusabilità	137
Modularità	137
Uno schema di documentazione	139
Convenzioni di scrittura dei programmi	141
Riusabilità e prototipazione	142
Note bibliografiche	145
Sommario	145
9. Strutture di dati e programmi	146
Sequenze	146
Pile e code	149
Insiemi.....	151
Matrici	155
Tecniche di ordinamento.....	157
Ordinamento per inserimento.....	157
Ordinamento a bolla d'aria	158
Ordinamento per divisione e composizione.....	158

Ordinamento ibrido	160
Alberi	161
Grafi	167
Note bibliografiche	175
Sommario	175
10. Operatori e predicati aritmetici	176
Gli operatori in Prolog	176
Precedenza di un operatore	177
Associatività di un operatore.....	177
Dichiarazione ed uso di operatori.	179
Funtori e predicati aritmetici.....	180
Ricerca del risultato di un'espressione aritmetica	181
Confronto dei risultati di un'espressione aritmetica.....	182
Note bibliografiche	183
Sommario.....	183
11. Predicati di controllo e di ingresso/uscita	184
Predicati predefiniti di controllo	184
Il predicato "call".....	184
Il predicato di disgiunzione ";"	186
I predicati "true" e "fail".....	186
Il predicato "repeat".....	187
Predicati predefiniti di ingresso/uscita.....	188
Predicati per la gestione di files	188
Ingresso ed uscita di termini	189
Ingresso ed uscita di caratteri.....	191
Note bibliografiche	195
Sommario.....	195
12. Predicati meta-logici	196
Verifica del tipo di un termine.....	196
Confronto di termini.....	198
Unificazione di termini	200
Il predicato "name".....	202
Verifica, accesso e formazione di strutture.....	205
Il predicato "functor".....	205
Il predicato "arg".....	206
Il predicato "=.."	208
Note bibliografiche	211
Sommario.....	211
13. La negazione	212
L'ipotesi del mondo chiuso	212
La negazione come fallimento finito	214
La congiunzione "!, fail" per condizioni negative	218
Note bibliografiche	220
Sommario.....	221
14. Gestione della base di dati	222
Aggiunta di clausole.....	222
Cancellazione di clausole.....	223
Ricerca di clausole.	224
Simulazione dell'assegnamento	225
Generazione di lemmi.	226
Memorizzazione di informazioni durevoli.....	228

Generazione di costanti.....	229
Raccolta di tutte le soluzioni.....	230
Note bibliografiche.....	237
Sommario.....	237
15. Prova dei programmi.....	238
Il modello di Byrd.....	238
Strumenti per la ricerca degli errori.....	241
Predicati di tracciamento.....	242
Antenati di una meta.....	243
Informazioni sul programma.....	243
Controllo dell'esecuzione.....	244
Uso delle risorse.....	245
Trattamento degli errori.....	246
Verifica di un programma.....	247
Note bibliografiche.....	249
Sommario.....	249

Introduzione al Prolog

Obiettivo di questa parte del corso è di costituire un'introduzione all'uso della logica simbolica come linguaggio di programmazione. Questa possibilità si è affermata con il nome di programmazione logica ed ha trovato una concreta realizzazione nel linguaggio Prolog.

La programmazione logica differisce significativamente dalla programmazione tradizionale (con linguaggi anche ad alto livello quali Fortran, Cobol, Basic, Algol, Pascal, Ada, . . .) in quanto richiede - e consente - al programmatore di descrivere la struttura logica del problema piuttosto che il modo di risolverlo: è compito del programmatore far sì che il programma logico rappresenti adeguatamente il problema. È invece compito del sistema Prolog utilizzare le informazioni presenti nel programma per effettuare le operazioni necessarie a dare risposte al problema, sfruttando un potente meccanismo di deduzione logica insito in esso.

Da un punto di vista concettuale, il programmatore può così concentrarsi sugli aspetti logici del problema e sul modo migliore per rappresentarli, senza essere distratto dalla necessità di fornire dettagli sul modo di pervenire ai risultati. Da un punto di vista operativo, i programmi Prolog richiedono un minore tempo di sviluppo, risultano più brevi e compatti in termini di numero di istruzioni, e più generali come risultati ottenibili dall'esecuzione del programma, rispetto ai programmi scritti in linguaggi tradizionali.

A partire dall'esplicitazione di tali caratteristiche, nuove e rilevanti per l'attività di programmazione e per le applicazioni, questa parte del corso si propone di sviluppare una trattazione argomentata, ma non apologetica, di questo approccio e di questo linguaggio. Lo scopo è di garantire allo studente tanto uno strumento effettivo di lavoro quanto la comprensione dei concetti che ne stanno alla base, tentando di conciliare le due esigenze - entrambe imprescindibili - di chiarezza e di rigore.

Questa parte del corso è costruita su tre assi principali, che corrispondono ad altrettanti obiettivi concomitanti:

Introduzione a Prolog come linguaggio di programmazione.

Le caratteristiche, i costrutti e le modalità di utilizzo del linguaggio sono introdotti in modo graduale, illustrandone possibilità, aspetti critici ed alternative. Si discutono in particolare tutti quegli aspetti procedurali che comportano uno scostamento rispetto alla pura descrizione logica del problema. Poiché le caratteristiche logiche del linguaggio non fanno scomparire del tutto dalla scena della programmazione i dettagli operativi, si evidenziano anche gli aspetti pratici necessari a controllare le inevitabili idiosincrasie del Prolog e delle sue implementazioni.

Metodologia di programmazione logica.

Vengono presentati e discussi aspetti di sviluppo incrementale, di strutturazione e modularizzazione, nonché di ricerca degli errori, di prova e di documentazione dei programmi logici. La non proceduralità della programmazione logica non implica la possibilità di ottenere il meglio del Prolog usandolo in modo semplicemente intuitivo, né garantisce automaticamente la qualità del programma; al contrario, invita a, e rende più significativo, un approccio strutturato, orientato alla qualità del programma.

Risoluzione logica di problemi.

Si esemplifica abbondantemente l'utilizzo del Prolog sia mediante una serie di procedure di base per la rappresentazione e manipolazione delle strutture di dati di più ricorrente utilizzo, sia mediante programmi che fanno riferimento ad alcuni problemi applicativi tipici, mettendone in evidenza le caratteristiche peculiari dell'approccio logico.

Questa molteplicità di obiettivi mira ad evidenziare una visione della programmazione, favorita dalla programmazione logica, non come fatto tecnico isolato, ma come attività inserita in un processo complesso di sviluppo di sistemi utili ed efficienti. In quest'ottica si discuterà anche il possibile utilizzo della programmazione logica per la realizzazione rapida di prototipi delle applicazioni, e per la riusabilità dei programmi. Si vuole anche suggerire una distinzione tra programmazione logica come approccio concettuale generale, e Prolog come una sua particolare attuale realizzazione, suscettibile di miglioramenti e potenziamenti, alcuni dei quali sono per altro in corso secondo varie direzioni, che vengono menzionate.

Come notazione si è adottata la sintassi del Prolog di Edimburgo che, in mancanza di uno standard, può considerarsi quella più ricorrente.

Rispetto all'approccio che privilegia un'introduzione iniziale sommaria dei costrutti di base, tale da consentire allo studente di passare subito all'esercitazione pratica sull'elaboratore, rinviando a momenti successivi le necessarie precisazioni ed articolazioni, si è preferito adottare un atteggiamento volto a distinguere gli aspetti concettuali da quelli operativi, consolidando i primi per poi passare ai dettagli relativi ai secondi.

Si ritiene preferibile che lo studente abbia familiarità con almeno un linguaggio di programmazione tradizionale, possibilmente di alto livello, e con i concetti fondamentali dell'informatica; questo vale non tanto per la comprensione del Prolog come linguaggio, la cui presentazione è completamente autocontenuta quanto per alcuni riferimenti ad aspetti di implementazione del linguaggio stesso o di esecuzione dei programmi. Non sono invece presupposte conoscenze di logica, né conoscenze specifiche di altro genere.

1. Rappresentazione di un problema

Dove si introducono e si descrivono i principali concetti e costrutti linguistici di Prolog concernenti la rappresentazione di oggetti, le relazioni tra oggetti e le proprietà logiche delle relazioni. E dove si discutono e si esemplificano alcune possibili alternative che si incontrano nell'esprimere le caratteristiche di un problema.

Il concetto di base della programmazione logica è di descrivere la conoscenza generale che si ha sul problema, piuttosto che uno specifico procedimento di soluzione. Si tratta di trovare la rappresentazione più adeguata alla specifica area applicativa ed esprimerla in un linguaggio logico.

Un problema, o un'applicazione, è caratterizzato dall'esistenza di oggetti discreti, o individui, da relazioni tra essi, e da proposizioni che esprimono proprietà logiche delle relazioni. Per rappresentare simbolicamente la conoscenza relativa ad un problema, occorre fare uso di un linguaggio formale, assegnando innanzi tutto dei nomi sia agli oggetti che alle relazioni.

Oggetti

Un oggetto può essere concreto, ad esempio un minerale, un vegetale, un animale, una persona, una cosa; oppure astratto, ad esempio il numero 2, il concetto di giustizia, la chimera, l'ascetismo.

Nell'assegnare un nome ad un oggetto, è indifferente che esso sia concreto o astratto. La scelta del nome è arbitraria, entro determinate convenzioni, ma naturalmente è opportuno che il nome sia espressivo, tale cioè da favorire l'associazione mnemonica all'oggetto da esso denotato.

Nomi semplici di oggetti sono detti costanti. Essi denotano oggetti elementari definiti; volendo stabilire un'analogia con la lingua naturale, corrispondono a nomi comuni e nomi propri. In Prolog, le costanti sono di due tipi: i numeri, scritti nel modo usuale, e gli atomi, ossia ogni altro nome come: "libro", "tavolo", "grano", "cerchio", "fratello", "equazione", "programma". Tali nomi in Prolog sono scritti secondo la seguente convenzione: qualunque sequenza di lettere (dell'alfabeto anglosassone), cifre (numerali arabi) e carattere di sottolineatura "_" (underscore) che comincia con lettera minuscola, oppure una qualunque sequenza di caratteri tra apici singoli. Sono costanti, ad esempio, le seguenti: **123**, **1.23**, **uno**, **cerchio**, **mario**, **roma**, **equazione1**, **equazione_2**, **unNome**, **'123'**, **x**, **'ETC'**, **y321**, **'Y321'**. Il carattere "_" può venire utilmente impiegato per migliorare la leggibilità degli atomi, come in:

il_mio_primo_tentativo_in_Prolog

chiamata_di_procedura

nome_di_costante_davvero_molto_lungo

Un oggetto può essere semplice, come negli esempi precedenti, oppure composto, ossia formato da altri oggetti componenti. Per esempio, una data è un oggetto composto i cui oggetti componenti sono il giorno, il mese e l'anno. Oggetti composti sono denotati da nomi composti, detti strutture. In Prolog le strutture sono costituite da un primo nome, detto funtore, seguito (in parentesi) da una sequenza, cioè un insieme ordinato, di uno o più altri nomi (separati da virgola se più di uno), detti componenti o argomenti.

Per esempio:

padre(mario)

frazione(3, 4)

data(27, marzo, 1980)

sono strutture con funtore **padre**, **frazione** e **data** rispettivamente. Per brevità, in generale una sequenza di **n** nomi è detta **ennupla** o **n-pla**; in particolare, coppia, tripla, quadrupla, ... per **n = 2, 3, 4, ...**, rispettivamente. Ogni funtore può precedere n-ple con un particolare valore di **n**, ed è quindi detto funtore **n-ario** (o ad **n** posti, o ad **n** argomenti, o di molteplicità **n**). Se più funtori hanno lo stesso nome ma molteplicità diverse, sono a tutti gli effetti funtori diversi. Le convenzioni di scrittura dei funtori sono le stesse di quelle degli atomi. Si noti che le costanti possono essere considerate come funtori **0-ari**, cioè senza argomenti.

I funtori si possono considerare analoghi ai nomi comuni della lingua naturale. Da un altro punto di vista, sono analoghi ai nomi di funzioni intese nell'usuale significato matematico, in quanto fanno passare dagli oggetti componenti agli oggetti composti; la notazione (con gli argomenti in parentesi) e la terminologia riflettono questa somiglianza.

I nomi che costituiscono la n-pla di argomenti di un funtore possono a loro volta essere strutture; ciò consente la costituzione di nomi arbitrariamente complessi, a vari livelli di composizione. Per esempio:

libro(le_città_invisibili, autore(italo, calvino))

è una struttura con primo funtore (detto **funtore principale**) libro ed i cui nomi componenti sono il nome semplice **le_città_invisibili** e la struttura con funtore **autore**;

dato(dato(d,o),dato(d,i))

è una struttura con funtore principale **dato** e con due componenti che sono a loro volta strutture con lo stesso funtore **dato** (e quindi entrambe con due componenti).

Le strutture, scritte in Prolog nel modo visto, possono essere visualizzate nelle usuali forme di rappresentazione gerarchica; per l'esempio precedente:



Figura 1.1.

Si considerino le seguenti strutture:

1. **libro**
2. **libro(le_città_invisibili)**
3. **libro(le_città_invisibili, calvino)**
4. **libro(le_città_invisibili, italo, calvino)**
5. **libro(le_città_invisibili, autore(italo, calvino))**
6. **libro(titolo(le_città_invisibili),autore(italo,calvino))**

Pur evocando (mnemonicamente) lo stesso tipo di oggetto, lo descrivono o a livelli di dettaglio diversi, o con lo stesso livello di dettaglio ma con diversa strutturazione:

1. può fare riferimento al concetto di libro, o ad un libro particolare ma non specificato;
2. fa riferimento allo specifico libro **le_città_invisibili**, che può tuttavia essere non univocamente determinato nel caso più autori abbiano scritto libri diversi con quello stesso titolo;
3. menziona esplicitamente l'autore, con un minor livello di dettaglio dei tre successivi;
4. 5. e 6. contengono la stessa informazione, ma con differenti modalità di strutturazione.

Si ha quindi la scelta del grado di dettaglio della rappresentazione, ma - una volta compiuta questa scelta - dev'essere data una rappresentazione unica dell'oggetto in questione.

Variabili

È possibile dare un nome, oltre che ad oggetti particolari, anche ad oggetti specifici ma da determinare, cioè non ancora identificati (in modo analogo all'uso del pronomo nel linguaggio naturale). Oggetti non specificati sono rappresentati da nomi di variabili; anche questi ultimi sono arbitrari, ed in Prolog sono caratterizzati dall'iniziare con una lettera maiuscola, o con "**_**". Si noti che, mentre costanti distinte denotano sempre oggetti diversi, questo non vale per le variabili, in quanto variabili distinte potranno venire a rappresentare lo stesso oggetto.

I nomi di variabili possono stare al posto dei nomi di costanti, specificamente come argomenti di funtori. Ad esempio:

punto(3, 7, 25)

rappresenta uno specifico punto nello spazio, mentre:

punto(3, 7, Z)

punto(3, Y, Z)

punto(X, Y, Z)

rappresentano punti da determinare.

Si possono usare le variabili per denotare non solo un singolo oggetto da determinare, ma anche una classe di oggetti. Ad esempio:

quadro(tintoretto, Titolo)

"tutti i quadri di Tintoretto"

quadro(tintoretto, olio(Titolo, 1572))

"tutti i quadri ad olio dipinti da Tintoretto nel 1572".

Descrizione di un oggetto singolo ma non ancora definito e descrizione di una classe di oggetti sono due modi diversi (complementari) di intendere una stessa rappresentazione: un singolo oggetto la cui tipologia è nota solo come schema generale può essere considerato come rappresentativo della classe di tutti gli oggetti di quella tipologia.

A volte non si è interessati a certi aspetti di un oggetto e non si ha quindi bisogno di nomi di variabili per riferirsi ad essi. Questi aspetti possono essere sostituiti da variabili anonime, scritte con "_"; ad esempio:

quadro(tintoretto, olio(Titolo, _))

"tutti i quadri ad olio dipinti da Tintoretto (non importa quando)".

Un altro uso ancora è quello di una variabile in comune tra più oggetti per rappresentare un vincolo tra essi; ad esempio:

periodo(data(X1, Y1, Z), data(X2, Y2, Z))

indicando che la terza componente, qualunque essa sia, è la stessa per entrambe le date, impone il vincolo che il periodo sia non superiore ad un anno. In questo caso si dice che la **variabile è condivisa** (shared).

Termini

In generale, i nomi degli oggetti sono detti termini. I termini non contenenti variabili sono detti termini chiusi (ground terms). Riassumendo, sono quindi termini:

1. le costanti;
2. le variabili;
3. le strutture, cioè le espressioni della forma **f(t1, t2, ..., tn)**, dove **f** è un funtore n-ario e **t1, t2, ..., tn** sono termini.

Si noti che la definizione dei termini è ricorsiva, ossia ciò che si sta definendo (un termine) compare (ricorre) nella definizione stessa. Una definizione ricorsiva consente di costruire termini arbitrariamente complessi. Le definizioni ricorsive rivestono un ruolo molto importante in Prolog, e saranno considerate più ampiamente nel seguito.

Una classificazione riassuntiva degli oggetti rappresentabili in Prolog è la seguente:

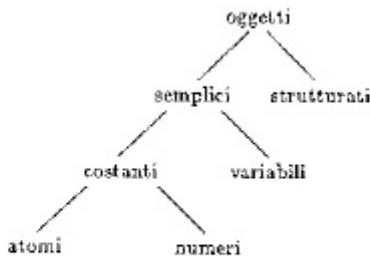


Figura 1.2.

Prolog mette a disposizione il termine, utilizzabile ricorsivamente, come unico strumento di rappresentazione di un oggetto, di applicabilità generale. Il tipo di un oggetto, in base alle convenzioni di scrittura viste prima, è rappresentato dalla forma sintattica del termine che lo denota, ed è quindi riconosciuto senza necessità di specificarlo esplicitamente: non occorrono, in altre parole, dichiarazioni di tipo. L'insieme degli oggetti denotati da tutti i termini usati in una data rappresentazione è detto l'universo del discorso, ossia costituisce tutto ciò di cui si parla in quella rappresentazione.

Operatori

Può essere conveniente scrivere alcuni funtori come operatori.

Si tratta di una forma sintattica che rende alcune strutture più facili da leggere. Le espressioni aritmetiche sono comunemente scritte con operatori, ad esempio:

"a + b" al posto di "+(a,b)"

"a + b * c" al posto di "+(a, *(b, c))"

La prima forma è detta notazione infissa, la seconda prefissa. La maggiore semplicità di uso della notazione infissa rispetto a quella prefissa deriva, nel caso delle espressioni aritmetiche, anche dal fatto che siamo abituati a scriverle e leggerle in questa forma. In generale, l'uso di un operatore evita di utilizzare le parentesi, che devono invece racchiudere gli argomenti del funtore di una struttura. Si noti che è comunque consuetudine usare per le espressioni aritmetiche la stessa rappresentazione astratta ad albero che abbiamo già considerato per le strutture; per esempio, nei due casi precedenti, qualunque sia la notazione usata, la rappresentazione ad albero è:



Figura 1.3.

Liste

In molti casi è opportuno utilizzare una lista, cioè una sequenza di un numero variabile (da nessuno ad un qualunque numero) di elementi. Tale numero viene detto lunghezza della lista. Il termine, che

- come si è visto - è l'unico tipo di dato in Prolog, può essere utilizzato per rappresentare una lista. Una lista può infatti essere considerata come una struttura che ha o nessuna componente, o due componenti, chiamate testa e coda, che rappresentano rispettivamente il primo elemento della lista, e tutti gli altri elementi escluso il primo. Nei due casi, coerentemente con le convenzioni Prolog, si possono usare come nomi una costante (un funtore a zero posti) ed una struttura con funtore a due posti.

L'uso frequente delle liste giustifica l'impiego, in Prolog, di due simboli speciali dedicati a questo scopo: la costante "[]" per denotare la lista vuota, ed il funtore ":" (punto) per separare la testa e la coda della lista. La testa è costituita da un singolo elemento, mentre la coda è a sua volta una lista. La fine della lista si rappresenta con una costituita dalla lista vuota. Ogni elemento di una lista può essere rappresentato da un qualunque termine, cioè una costante, od una variabile, od una struttura, compresa eventualmente un'altra lista. Come tutte le strutture, una lista può essere rappresentata ad albero. Alcuni esempi di liste espresse nella notazione suddetta, e delle relative rappresentazioni ad albero, sono i seguenti:



Figura 1.4.a: Lista avente come unico elemento l'atomo a.

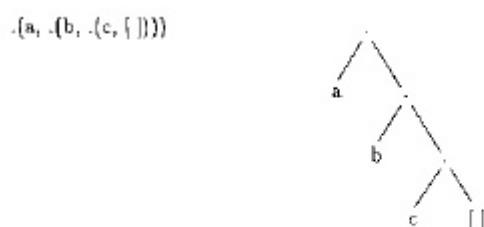


Figura 1.4.b: Lista consistente degli elementi atomici a, b, c.

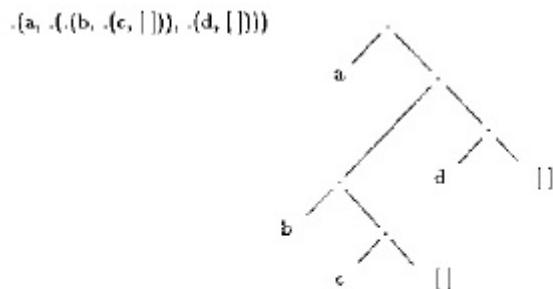


Figura 1.4.c: Lista consistente dell'elemento atomico a, di un elemento lista consistente di elementi atomici b e c, e di una lista il cui unico elemento atomico è d.

`.(a, .(X, []))`

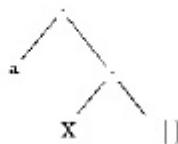


Figura 1.4.d: Lista costituita dall'atomo `a` e del termine generico rappresentato dalla variabile `X`.

Si noti che nelle liste l'ordine degli elementi è *rilevante*, in quanto una lista rappresenta un insieme *ordinato di elementi*; perciò, ad esempio, la lista `.(a, .(b, []))` è diversa dalla lista `.(b, .(a, []))`.

Il funtore `".."` può essere definito come operatore, cioè si può usare la notazione infissa al posto di quella prefissa. I casi precedenti possono per esempio essere espressi, rispettivamente, con le seguenti notazioni infisse:

`a.[]`

`a.b.c.[]`

`a.(b.c).d.[]`

`a.X.[]`

dove le parentesi sono omesse per gli elementi consecutivi perché l'operatore `".."` è definito come associativo a destra.

Poiché, come si vede chiaramente, rappresentare una lista con il funtore `".."` porta spesso ad espressioni difficilmente leggibili nel caso di liste complesse, è definita in Prolog una sintassi speciale, detta appunto **notazione a lista** (bracket notation). Questa consiste nel racchiudere l'intera lista tra parentesi quadre, scrivendo al loro interno gli elementi separati da virgolette (viene omesso il simbolo di lista vuota come elemento di chiusura della lista). I casi precedenti possono per esempio venire espressi, rispettivamente, con le seguenti notazioni a lista:

`[a]`

`[a, b,c]`

`[a,[b,c],d]`

`[a,X]`

S'intende che anche nella notazione con parentesi quadre, come in quella con funtore `".."`, la testa della lista è il primo elemento e la coda consiste della lista contenente ogni altro elemento tranne il primo. Negli esempi seguenti sono evidenziate, nella notazione a lista, la testa e la coda della lista:

<i>Lista</i>	<i>Testa</i>	<i>Coda</i>
[a,b,c]	a	[b,c]
[a,b]	a	[b]
[a]	a	[]
[]	-	-
[[a,b],c]	[a, b]	[c]
[a,[b,c]]	a	[[b,c]]
[a [b,c],d]	a	[[b,c],d]
[[1,2,3],[2,3,4],[]]	[1,2,3]	[[2,3,4],[]]

Nella notazione a lista, vi è in Prolog anche un simbolo speciale, "|" (barra verticale), per evidenziare la suddivisione di una lista in testa e coda: [X|L] (equivalente a .(X,L)) è un termine che denota una lista la cui testa è l'elemento X e la cui coda è il termine L, che denota una lista. Si noti che a sinistra di "|" possono comparire più termini, per esempio [X,Y|Z], mentre a destra deve comparire una lista; così, [X|Y, Z] non è ammesso, mentre [X|[Y|Z]] è possibile, ed è equivalente a [X,Y|Z]. Le seguenti notazioni sono tra loro equivalenti:

[1,2,3]

[1, 2,3| []]

[1,2|[3|[]]]

[1|[2,3|[]]]

[1|[2|[3|[]]]]

Negli esempi seguenti è illustrato il significato associato all'uso delle variabili (comprese le variabili anonime "_"):

[X|L] "tutte le liste non vuote"

[X, Y, Z | []] "tutte le liste di 3 elementi"

[X,Y,Z,Coda] "tutte le liste di 4 elementi"

[X, X, Coda] "tutte le liste di 3 elementi nelle quali si ha che il primo ed il secondo elemento sono lo stesso termine";

[41,X|Y]

"tutte le liste di almeno 2 elementi, di cui il primo è il numero 41";

[a,_,_,b]

"tutte le liste di 4 elementi in cui il primo è l'atomo a, il quarto è l'atomo b, il secondo e il terzo possono essere indifferentemente o due qualsiasi termini tra loro diversi o lo stesso termine".

Relazioni

Una relazione è l'attribuzione di una qualità comune a più oggetti. Ad esempio, una relazione di parentela, come l'essere genitore oppure l'essere figlio, correla ogni individuo ai suoi parenti (ogni genitore ai suoi figli, o viceversa). Naturalmente, una relazione può valere tra più di un gruppo di oggetti; per esempio, la relazione "padre di" vale tra molte coppie di persone. Viceversa, un singolo gruppo di oggetti può soddisfare più di una relazione, ad esempio, Rossi è "compagno di lavoro" di Bianchi, ed anche suo "vicino di casa".

In generale, una relazione è un insieme di n-ple, e correla gli oggetti nominati in ogni n-pla. In Prolog una relazione è denotata da un nome, detto predicato o simbolo predicativo, seguito (in parentesi) dalla n-pla dei nomi degli oggetti correlati (separati da virgola). Perciò ogni predicato, come ogni funtore, ha la sua molteplicità; in particolare, si dice binario se ha 2 argomenti, ternario se ha 3 argomenti, e così via (nel seguito, per semplicità, si dirà "predicato" sia il simbolo predicativo, sia l'insieme di esso e dei suoi argomenti, ed il modo di intenderlo sarà chiaro dal contesto).

Ad esempio, in:

madre(luisa, mario)

madre(luisa, giovanna)

madre(carla, ugo)

si ha una relazione con predicato binario **madre**.

Si noti che, poiché gli argomenti sono ordinati, la relazione **madre(giovanna, luisa)** è diversa dalla relazione **madre(luisa,giovanna)**. L'ordine degli argomenti, come il nome del predicato, è scelto arbitrariamente, ma - una volta fissato - deve rimanere coerente per tutte le n-ple della relazione. Ugualmente, va scelto il numero di argomenti ed il grado di dettaglio desiderato; si confrontino ad esempio le seguenti relazioni:

gioca(giuseppe, elena, tennis)

gioca_a_tennis(giuseppe, elena)

gioca(giuseppe, elena)

Gli oggetti nominati nella relazione possono essere semplici o composti, e questo dà un ulteriore grado di libertà nell'espressività della rappresentazione; si confrontino ad esempio le seguenti relazioni:

possiede(mario, libro)

possiede(mario, libro(titolo(le_città_invisibili), autore(italo, calvino)))

I predicati scelti esprimono quelle relazioni tra oggetti che si considerano pertinenti nel contesto del problema affrontato. In particolare, una relazione non deve necessariamente nominare tutti gli oggetti implicati: ad esempio può darsi che **carla** nella relazione **madre(carla, ugo)** abbia altri figli, ma ai fini della rappresentazione scelta la loro menzione è ininfluente. Gli oggetti nominati non devono necessariamente avere un legame nel mondo reale, ma si possono mettere insieme arbitrariamente, ad esempio:

relazione_arbitraria(temperatura_di_oggi, altezza_duomo_di_milano)

Un preredicato può essere unario (con un solo argomento); per esempio:

pesante(piombo)

Si può osservare, inoltre, che:

persona(giovanni)

ha un senso, mentre non lo ha:

giovanni(persona)

ma questo è dovuto unicamente al significato mnemonico associato ai nomi.

Un predicato unario esprime una proprietà di un oggetto piuttosto che una relazione tra oggetti, ma per estensione di linguaggio (e per semplicità) si può considerare una proprietà come un caso particolare di relazione. D'altra parte, è sempre possibile la scelta tra rappresentazioni diverse; si può avere, ad esempio:

maschio(mario)

femmina(luisa)

oppure:

sesso(mario, maschile)

sesso(luisa, femminile)

Una frase italiana con la copula, ad esempio "Roberto è ingegnere", può essere rappresentata o come proprietà:

ingegnere(roberto)

o come relazione binaria:

è_un(roberto,ingegnere)

Con ulteriore estensione di linguaggio, si possono considerare predicati senza argomenti, ad esempio:

piove

Si noti che è sempre possibile esprimere una relazione n-aria mediante n+1 relazioni binarie. Ad esempio, al posto della relazione ternaria:

gioca(giuseppe, elena, tennis)

si possono usare le quattro relazioni binarie seguenti:

azione(a, gioco)

attore(a, giuseppe)

co_attore(a, elena)

tipo_di_gioco(a, tennis)

Questa alternativa è meno espressiva, ma più flessibile, della precedente. Se ad esempio si volesse successivamente specificare il luogo, occorrerebbe nel primo caso aggiungere un argomento alla relazione, cambiando quindi il predicato (che è vincolato ad una molteplicità prefissata); nel secondo caso occorre invece un'ulteriore relazione, che si aggiunge alle precedenti senza alterarle. In generale, una relazione si può esprimere in tanti modi diversi; la rappresentazione di una relazione dipende dallo scopo.

Riassumendo, i predicati sono espressioni di forma **p(t₁, t₂, ..., t_n)**, dove p è un simbolo predicativo e t₁, t₂, ..., t_n sono termini. Si noti la *completa uguaglianza formale* che sussiste tra funtori e predicati; un predicato è semplicemente un funtore che compare come funtore principale in un particolare contesto, come meglio si vedrà tra breve. Analogamente ad un termine, si diranno *predicati chiusi (ground predicates)* i predicati che non contengono variabili negli argomenti.

Nel rappresentare una relazione, occorre scegliere il nome della relazione (in modo espressivo), il numero degli argomenti (gli aspetti considerati), l'ordine degli argomenti, ed eventualmente le n-pie che costituiscono la relazione: l'insieme di questi fattori influenza il significato che chi rappresenta la relazione decide di attribuire ad essa.

Un predicato **p(t)**, dove t è una n-pla t₁, ..., t_n, può essere considerato come l'asserzione che la relazione chiamata **p** vale tra gli individui chiamati t₁, ..., t_n, e può essere letto informalmente come "t appartiene alla relazione p" o, equivalentemente, come "la proposizione p è vera per t".

I predicati corrispondono in genere ai verbi del linguaggio naturale (ad esempio, "possiede", "gioca") ma possono anche corrispondere ad aggettivi ("maschio", "prezioso") od a nomi ("madre").

Proposizioni

Se i termini denotano oggetti e i predicati denotano relazioni tra gli oggetti denotati dai termini, le proprietà logiche delle relazioni sono descritte da proposizioni (sentences). Nella notazione Prolog, ogni proposizione termina con un punto. I predicati stessi, singolarmente considerati, possono costituire proposizioni atomiche. Ad esempio:

ama(mario, luisa).

è una proposizione atomica (di cui è ovvio il significato associato). Proposizioni non atomiche sono costituite da più predicati connessi da operatori (o connettivi) logici, denotati da simboli speciali.

Il connettivo ":-" è detto condizionale ("se"), o implicazione ("è implicato da"). Ad esempio, la proposizione:

ama(mario, luisa) :- ama(luisa, mario).

è costituita dai predicati **ama(mario, luisa)** e **ama(luisa, mario)** connessi dall'implicazione, e si può leggere come: "mario ama luisa se luisa ama mario", oppure "mario ama luisa è implicato da luisa ama mario", o ancora, equivalentemente, "luisa ama mario implica mario ama luisa".

Il connettivo "," denota una congiunzione ("e"). Ad esempio, la proposizione:

ama(mario, luisa) :ama(luisa, mario), ama(luisa, poesia).

è costituita dai predicati **ama(luisa, mario)** e **ama(luisa, poesia)**, connessi tra loro dalla congiunzione e connessi al predicato **ama(mario, luisa)** dall'implicazione; si può leggere come: "mario ama luisa se luisa ama mario e luisa ama la poesia". I predicati a destra del simbolo condizionale sono detti premesse o condizioni, e il predicato a sinistra è detto conclusione.

Proposizioni prive di variabili sono dette proposizioni chiuse (ground sentences). In Prolog le proposizioni atomiche base sono anche dette fatti. Le proposizioni possono essere costituite da predicati contenenti variabili. In questo caso esse sono intese come affermazioni che riguardano tutti i possibili oggetti rappresentati dalle variabili. In italiano, questo corrisponde all'uso dei pronomi, come "tutti", "ogni", "chiunque", "qualunque". Ad esempio, le proposizioni:

ama(mario, X).

ama(mario, X) :- ama(X, poesia).ama(mario, X) :- donna(X), ama(X, poesia).

si possono leggere rispettivamente come: "mario ama tutti" o "mario ama chiunque", "mario ama chiunque ami la poesia" e "mario ama qualunque donna ami la poesia".

Si osservi che un dato nome di variabile rappresenta sempre lo stesso oggetto all'interno di una proposizione, mentre nomi di variabili uguali in proposizioni diverse non hanno relazione tra loro, cioè non rappresentano lo stesso oggetto. Si dice che la proposizione è il campo di validità lessicale (Lexical scoping field o semplicemente scope) di un nome di variabile.

Le proposizioni atomiche sono dette anche asserzioni. Le proposizioni non atomiche condizionali (contenenti cioè l'implicazione) in Prolog vengono dette anche regole. Una proposizione costituisce

un'affermazione su oggetti specifici, se è una proposizione di base, o su classi di oggetti, se è una proposizione con variabili.

Una singola regola può sintetizzare un insieme di fatti; per esempio, i fatti:

pari(2).

pari(4).

...

possono essere espressi sinteticamente dalla regola:

pari(X) :- divisibile(X, 2).

Si può vedere questa regola come la definizione della proprietà **pari** mediante la relazione **divisibile**.

In generale, usando l'implicazione, si può definire una relazione mediante una o più altre relazioni; ad esempio:

figlio(X, Y) :- maschio(X), padre(Y, X).

Si può definire una relazione inversa di un'altra relazione; ad esempio:

figlio_a(X, Y) :- padre(Y, X).

È possibile definire una gerarchia di proprietà, come in:

animale(X) :- cane(X).

che esprime che "tutti i cani sono animali" ovvero "i cani sono un sottoinsieme degli animali". S'intende che, con questo significato associato, non è possibile la definizione inversa:

cane(X) :- animale(X).

La gerarchia può essere multipla; ad esempio:

animale(X) :- carnivoro(X).

carnivoro(X) :- cane(X).

In generale, una relazione può essere definita mediante altre relazioni, queste a loro volta con altre, e così via. Si ha quindi una gerarchia di relazioni; ad esempio:

cugini(X, Y) :- genitore(S, X), genitore(T, Y), fratelli(S, T), X \neq Y.

fratelli(X, Y) :- genitore(Z, X), genitore(Z, Y), X \neq Y.

Si noti che il fatto che un individuo non è cugino o fratello di se stesso, ovvio nel significato usuale di questi termini, deve essere espresso esplicitamente nella definizione formale, mediante il

predicato " \neq " ("diverso da"). Quest'ultimo è un predicato predefinito, ovvero il suo nome, la sua molteplicità ed il significato associato sono ritenuti già noti in Prolog e quindi possono essere usati senza definirli esplicitamente. Vi sono in Prolog numerosi prediciati predefiniti, che saranno considerati più avanti.

Naturalmente la relazione cugini può essere espressa in un'unica proposizione, come segue:

cugini(X, Y) :- genitore(S, X), genitore(T, Y), X ≠ Y, genitore(Z, S), genitore(Z, T), S ≠ T.

Quest'ultima rappresentazione è più compatta della precedente, ma la prima, oltre a suddividere la definizione in due sottoparti singolarmente più semplici, introduce la relazione **fratelli** che in questo contesto fa da intermediaria tra le due sottoparti della definizione, ma può essere utile anche in altri contesti, come in effetti avviene nell'uso di queste relazioni nella lingua naturale. In ogni caso, vale sempre la considerazione che, fra i vari modi possibili di formulare le regole, si sceglie quello considerato più pertinente ed utile allo scopo della rappresentazione.

Una relazione può essere definita da più proposizioni condizionali aventi lo stesso predicato come conclusione (o da più proposizioni atomiche con lo stesso predicato); in questo caso esse vengono considerate in alternativa tra loro, cioè implicitamente connesse dall'operatore logico di **disgiunzione** ("o"). Ad esempio, le proposizioni:

genitore(X, Y) :- padre(X, Y).

genitore(X, Y) :- madre(X, Y).

si leggono come: "X è genitore di Y se è padre o è madre di Y".

Una relazione può essere definita ricorsivamente. In questo caso la definizione richiede almeno due proposizioni: una è quella ricorsiva che corrisponde al caso generale, l'altra esprime il caso particolare più semplice. Ad esempio, la seguente definizione:

antenato(X,Y) :- genitore(X, Y).

antenato(X, Y) :- genitore(Z, Y), antenato(X, Z).

si può leggere come: "X è antenato di Y se è genitore di Y o è un antenato del genitore di Y".

Una proposizione può fare uso di variabili anonime, quando la rappresentazione specifica dei corrispondenti oggetti è considerata irrilevante. Ad esempio, la proposizione:

coniuge(X) :- sposato(X,_).

si può leggere come: "un individuo X è coniuge se è sposato, non importa con chi"; la definizione prescinde dall'identità dell'altro termine della relazione.

Programmi Prolog

Una proposizione del tipo:

A:-B1, B2, ..., Bn.

dove **A, B1, B2, ..., Bn** sono predicati, è detta una **clausola di Horn** (nel seguito chiamata semplicemente clausola). La conclusione **A**, che è costituita da un unico predicato, è detta testa della clausola. La congiunzione delle condizioni **B1, B2, ..., Bn** è detta corpo, o coda, della clausola. Le condizioni possono mancare, cioè si può avere la conclusione senza condizioni (e si omette anche il simbolo ":-"); in questo caso si parla di clausola unitaria.

Si può osservare che, strutturalmente, anche le clausole possono essere viste come termini. Infatti, una clausola unitaria ha già la forma di un termine (si è detto precedentemente dell'uguaglianza formale tra predicati e funtori). Una clausola non unitaria con $n > 0$ condizioni può essere vista come un termine che ha come funtore $n+1$ -ario il connettivo ":", come primo argomento la testa della clausola, e come argomenti successivi le n condizioni del corpo della clausola, come segue:

: - (A, B1, B2, ..., Bn).

Questa considerazione evidenzia l'uniformità del linguaggio, ed è alla base della possibilità di trattare clausole, e quindi più in generale programmi, come dati.

Un programma Prolog è un insieme finito di clausole, unitarie e non, scritte in sequenza. Tale insieme di clausole è detto anche la base di dati del programma. Essa rappresenta l'insieme delle conoscenze espresse (come fatti e come regole) sul problema.

Riassumendo, il linguaggio con cui descrivere un problema è costituito da:

1. i vocabolari (insiemi di simboli) di costanti, variabili, funzioni e predicati; questi simboli, ed il significato ad essi associato, sono scelti dall'utente del linguaggio, in funzione del problema considerato;
2. i connettivi logici ed i simboli ausiliari (le parentesi), il cui significato è prestabilito in Prolog;
3. le proposizioni costruite con i simboli precedenti, secondo le regole Prolog considerate.

Questo linguaggio consente di rappresentare problemi che concernono oggetti, relazioni tra oggetti, e proprietà logiche delle relazioni. La semplicità della sintassi del Prolog deriva dalla semplicità della sintassi del linguaggio logico. Il linguaggio della logica simbolica è allo stesso tempo semplice, non ambiguo e di grande potenza espressiva, e può essere considerato come una forma semplificata della lingua naturale.

Note bibliografiche

La logica dei predicati è ben più antica della scienza degli elaboratori, e pertanto di qualsiasi linguaggio di programmazione.

L'idea di usare tale logica come linguaggio di programmazione, e l'individuazione di quel suo sottoinsieme costituito dalle clausole di Horn come il più adatto allo scopo, è comunemente attribuita ai lavori di *Kowalski* a Edimburgo e di *Colmerauer* a Marsiglia. Una breve storia di tali

inizi è riportata dallo stesso *Kowalski (1985)*. Sulle radici di tali lavori si vedano anche *Robinson (1989)* e *Hogger (1981)*.

Il linguaggio introdotto da *Colmerauer* fu implementato in Algol W su un IBM 360/67 da *Roussel (1975)* e venne utilizzato per diverse applicazioni. Fu poi sostituito da una versione migliorata, in parte scritta in Fortran da *Battani e Meloni (1973)* ed in parte in Prolog. Anche se i suoi autori non le diedero un nome, viene chiamata Prolog I (*Kluzniak e Szpakowicz (1985)*).

Il linguaggio Prolog è stato inizialmente un fenomeno europeo. Più recentemente ha suscitato maggiore attenzione negli Stati Uniti, in seguito all'articolo di *McDermott (1980)*, ed in Giappone, dove è stato scelto come linguaggio di programmazione fondamentale nel progetto dei sistemi di elaborazione della quinta generazione, presentato in *Moto-oka (1981)*.

Una vasta bibliografia sulla programmazione logica è contenuta in *Poe, Nasr, Potter e Slinn (1984)*.

Sommario

Lo studente ha ora gli elementi più importanti per esercitarsi nella rappresentazione di un problema usando il linguaggio logico nella notazione Prolog.

2. Interpretazione dichiarativa.

Dove si precisa l'interpretazione dichiarativa delle descrizioni effettuate nel linguaggio descritto nella [Rappresentazione di un problema](#), sia in modo informale espresso in italiano, sia soprattutto in un modo formale basato su un procedimento di dimostrazione logica.

Quest'ultimo riguarda la derivazione, a partire dalle clausole di un programma Prolog e da quesiti riguardanti le relazioni in esso descritte e presupposte vere, di nuovi fatti che da esse conseguono e che costituiscono la risposta ai quesiti posti.

Programmazione dichiarativa.

L'idea fondamentale della programmazione logica, realizzata nel linguaggio Prolog con le caratteristiche e le limitazioni che saranno via via illustrate nel seguito, è quella di programmare per descrizioni.

Nella programmazione tradizionale, di tipo procedurale (o imperativo, o prescrittivo), il programma viene sviluppato specificando le operazioni da compiere per la risoluzione di un problema, ovvero indicando come il problema va risolto: le ipotesi sulle quali il programma si fonda rimangono pertanto generalmente implicite, non trovano una corrispondenza sistematica nel programma.

Nella programmazione logica, di natura dichiarativa, il programma viene costruito tramite la descrizione della sua area di applicazione, specificando cosa è supposto vero: le assunzioni fatte sono dunque ora esplicite; implicita, al contrario, risulta la scelta delle operazioni da effettuare. L'informazione espressa in logica può allora essere compresa senza riferimento al comportamento che essa evoca nella macchina, come è invece necessario negli usuali linguaggi di programmazione.

Dato tale carattere descrittivo, un programma Prolog è più vicino a ciò che normalmente è considerata una specifica di un problema, che tuttavia è eseguibile e quindi in grado di fornire risposte a domande che riguardano quel problema.

Come si è detto nella [Rappresentazione di un problema](#), la descrizione di un'area di applicazione nel linguaggio Prolog ha inizio con una rappresentazione degli oggetti che si ritengono rilevanti in tale area e delle relazioni che si presuppongono soddisfatte da questi oggetti. Una volta scelti i nomi (termini) con cui denotare gli oggetti, ed i nomi (predicati) con cui denotare le relazioni, le clausole rappresentano affermazioni descrittive delle connessioni tra esse.

L'interpretazione dichiarativa del linguaggio Prolog consente da un lato di descrivere il significato delle clausole, informalmente, con affermazioni espresse in lingua naturale; dall'altro lato, essa ha nella logica una base formale per una interpretazione rigorosa e non ambigua di tali proposizioni, e per dedurre da esse nuove informazioni. Il problema della programmazione viene così ricondotto a quello dell'individuazione di legami logici tra insiemi di variabili, in una forma tale da permettere ad un procedimento di deduzione di risolvere uno o più vincoli per i valori di alcune delle variabili, assegnati i valori per le altre. Risulta possibile in tal modo sottoporre ad elaborazione la rappresentazione di problemi che riguardano oggetti e relazioni tra oggetti.

Nel seguito si mostrerà l'uso di descrizioni in italiano a scopo di esplicazione delle clausole, e si illustrerà in quale modo la logica fornisce una semantica precisa per esse, ed un procedimento di deduzione che costituisce allo stesso tempo un meccanismo di computazione.

Interpretazione formale ed informale.

Riprendendo quanto accennato in [Programmi Prolog](#), una proposizione in forma di clausola è del tipo:

p :- p1, ... , pn

dove **p, p1, ..., pn**. sono prediciati che possono contenere un insieme di variabili **X1, ..., Xk** come argomenti. Poiché una variabile **X**, argomento di un predicato p, denota un dominio di individui, **p(X)** può essere considerato come un'abbreviazione di **p(a1) e p(a2) e ... e p(an) e ...**, (con ai appartenenti al dominio), che si può leggere come: " **p** è vera per ogni elemento (o per qualsiasi elemento, o per tutti gli elementi **ai**) del dominio considerato". Si dice che la variabile **X** è quantificata universalmente, ovvero è implicitamente soggetta all'operatore logico, chiamato quantificatore universale, "per ogni". Ciò vale per tutte le variabili e per tutti i prediciati che figurano nella clausola; essa non può contenere variabili che non siano soggette ad un quantificatore.

Il fatto di trattare le variabili come universalmente quantificate nella clausola in cui compaiono è correlato al considerare tali variabili come locali a quella clausola. Una proposizione espressa con una clausola, quindi, non fa affermazioni relativamente ad individui generici, ma solo relativamente ad individui particolari (denotati dalle costanti) oppure relativamente a tutti gli individui di un dominio (denotati dalle variabili).

La clausola del tipo suddetto è interpretata dichiarativamente come la proposizione (condizionale):

"per tutti gli **X1, ..., Xk**, si ha che **p** è implicato da **p1, ... pn**",

ovvero:

"**p** è vero se tutti i **p1, ..., pn** sono veri".

Se **n = 0**, la clausola diventa unitaria:

p :-

più semplicemente scritta nella forma:

p.

(nel seguito si userà sempre questa scrittura abbreviata) ed è interpretata dichiarativamente come la proposizione (incondizionata, o asserzione):

"per tutti gli **X1, ..., Xk**, si ha che **p** è vero (sotto tutte le condizioni)".

Se **p** manca, la clausola diventa:

:-p1, ..., pn.

ed è interpretata dichiarativamente come la proposizione (negativa):

"per nessun **X1, ..., Xk**, si ha che **p1, ..., pn**, sono tutti veri "

oppure, equivalentemente:

"per tutti gli **X₁, ..., X_k**, si ha che **non p₁ o ... o non p_n**"

o ancora:

"non esiste alcun **X₁, ..., X_k** per cui **p₁, ..., p_n** sono tutti veri".

Nelle proposizioni negative le variabili sono considerate quantificate esistenzialmente, ovvero soggette implicitamente all'operatore logico, chiamato quantificatore esistenziale, "esiste". Come caso particolare si considera anche la clausola vuota, scritta nella forma:

:-

interpretata dichiarativamente come contraddizione (o proposizione sempre falsa). Si noti che l'ordine con cui sono scritti i predicati **p₁, ..., p_n** non ha alcun significato dichiarativo (logico), perché ogni predicato stabilisce qualche relazione riguardo al problema indipendentemente dagli altri.

Un insieme di clausole aventi le stesse premesse e conclusioni diverse è interpretato come la proposizione (non esprimibile in un'unica clausola) costituita dalle premesse comuni e dalla congiunzione delle diverse conclusioni. Ad esempio, l'insieme delle due clausole:

a:-c.

b:-c.

è equivalente alla proposizione: "**a e b sono implicati da c**".

Un insieme di clausole aventi premesse diverse e la stessa conclusione è interpretato come l'implicazione della conclusione a partire dalla disgiunzione delle ipotesi. Per esempio l'insieme delle due clausole:

a:-b.

a:-c.

è equivalente alla proposizione: "**a è implicato da b o da c**". Di nuovo, l'ordine con cui sono scritte le clausole non ha significato logico.

Consideriamo alcuni esempi, nei quali faremo uso della seguente notazione Prolog per i commenti: tutto ciò che è compreso tra `/*` e `*/` è un commento. Useremo i commenti come documentazione della lettura dichiarativa, in italiano, delle clausole. Il caso più semplice è quello dei fatti (clausole unitarie senza variabili), per esempio:

inversione([1, 2, 3],[3,2, 1]). /* la lista [3, 2, 1] è l'inversa della lista [1, 2, 3] */

Un altro caso è costituito dalle asserzioni (clausole unitarie con variabili), ad esempio:

derivata(X, X, 1). /* per ogni X, la derivata di X rispetto a X è 1 */

primo([T|_], 1, T). /* per ogni T, il primo elemento della lista che ha testa T e coda qualsiasi è ancora T */

Nel caso più generale si ha un insieme di definizioni logiche di relazioni, con variabili. Ad esempio:

minorenne (Persona) :- anni(Persona, N), minore(N, 18). /* per ogni Persona, Persona è minorenne se ha N anni e N è minore di 18 */

L'interpretazione dichiarativa vista sopra, che esprime in italiano il "significato" associato alle clausole, è da intendere in senso informale, di tipo descrittivo di ciò che si suppone vero e rilevante rispetto al problema o al dominio applicativo considerato. Il significato formale di un insieme di clausole può essere identificato con l'insieme di tutte le proposizioni che sono conseguenza logica di esse; la nozione di conseguenza logica è necessaria, e sufficiente, per descrivere la semantica di un programma logico.

Al fine di stabilire se un insieme di proposizioni ammette come conseguenza logica una certa altra proposizione, si usano opportune regole di inferenza (o di deduzione, o di dimostrazione, o di derivazione). La programmazione logica è basata su una dimostrazione per confutazione (reductio ad absurdum), che consiste in questo: se si dimostra che la negazione della proposizione da dimostrare, considerata rispetto all'insieme delle proposizioni date, porta ad una contraddizione (è incoerente con esse), allora quella proposizione è vera. Se non si può confutare (non si ottiene una contraddizione), allora la proposizione di partenza (non negata) e le altre non possono mai essere contemporaneamente tutte vere; perciò la proposizione di partenza non è conseguenza logica delle altre.

Risoluzione.

Per attuare la dimostrazione per confutazione si usa un'unica regola di inferenza, molto potente, chiamata risoluzione. Data una clausola con un predicato p a sinistra di ":" (cioè come testa della clausola) ed un'altra clausola con lo stesso predicato p alla destra (nella coda), è possibile creare una nuova clausola in cui la parte sinistra è la congiunzione delle parti sinistre delle due clausole originarie con p cancellato, e la parte destra è la congiunzione delle parti destre delle due clausole con p cancellato. Le due clausole di partenza sono dette clausole genitrici, e quella ottenuta è detta clausola risolvente.

Per esempio, date le clausole genitrici:

genitore(antonio, giorgio) :- padre(antonio, giorgio).

nonno(antonio, carlo) :- genitore (antonio, giorgio), genitore (giorgio, carlo).

si ottiene la clausola risolvente:

nonno(antonio, carlo):-padre(antonio, giorgio), genitore (giorgio, carlo).

L'esempio precedente è il caso più semplice, nel quale i predicati tra cui avviene la risoluzione sono identici. In realtà, la regola di risoluzione è più generale, in quanto può essere usata anche quando i due predicati non sono identici, purché si possa renderli tali istanziando opportunamente le loro variabili, ovvero trovando opportune costanti (istanze) da sostituire alle variabili. Il procedimento che determina tali istanze di sostituzione è detto unificazione.

Per esempio, date le clausole:

genitore(X, giorgio) :- padre(X, giorgio).

nonno(antonio, carlo) :- genitore (antonio, Y), genitore(Y, carlo).

si ottiene la clausola:

nonno(antonio, carlo) :- padre(antonio, giorgio), genitore (giorgio, carlo).

con la sostituzione della costante **antonio** alla variabile **X**, e della costante **giorgio** alla variabile **Y**. In generale, si indicherà con **X/t** la sostituzione del termine **t** alla variabile **X**; si dice anche che **X** è legata (bound) (o istanziata) a **t**. Una variabile non (ancora) legata si dice libera o non istanziata.

Ricordando che variabili in clausole diverse sono da considerare distinte anche se hanno lo stesso nome, se le clausole genitrici hanno variabili in comune queste vanno considerate rinominate, dando clausole equivalenti (varianti) senza variabili in comune, prima di ricavare la risolvente. Negli esempi che seguono si effettuerà la rinominazione delle variabili a scopo dimostrativo; è però da tenere presente che questa operazione viene compiuta automaticamente dal sistema Prolog, il quale assicura così la località delle variabili alle singole clausole nelle quali si trovano. Per esempio, partendo da:

:- a(Y, W), b(W, Z).

a(X, Y) :- c(X, W), d(W, Y).

si rinominano le variabili della seconda clausola, che diventa:

a(X', Y') :- c(X', W'), d(W', Y').

e quindi si ottiene la risolvente:

:- c(Y, W'), d(W', W), b(W, Z). sostituendo **Y** a **X'** e **W** a **Y'**.

Quesiti.

Si è detto prima che, se si vuole dimostrare che una proposizione è vera rispetto a un dato insieme di proposizioni, la si nega, scrivendola nella forma:

:-p1, ... pn.

e si cerca di derivare una contraddizione. Nell'ambito della programmazione logica e della dimostrazione per confutazione, tale proposizione negativa è chiamata un quesito (query), la cui risposta è positiva se si deriva la clausola vuota, negativa altrimenti. Nella spiegazione dichiarativa, la proposizione negativa può essere espressa in forma interrogativa; per evidenziare questo, nel seguito si userà nei quesiti il simbolo **?-** al posto di **:-**.

Consideriamo il classico esempio di sillogismo aristotelico, espresso nella forma a clausole:

```

uomo(socrate). /* socrate è un uomo */
mortale(X) :- uomo(X). /* tutti gli uomini sono mortali */

```

e poniamo il quesito:

?- mortale(socrate). /* socrate è mortale? */

Mediante la sostituzione **X/socrate** si ha la risoluzione tra il quesito e la regola, ottenendo la risolvente.

:- uomo(socrate).

e quindi la risoluzione tra questa ed il fatto, da cui si deriva:

:-

concludendo così che la risposta è positiva ("socrate è mortale"). Parafrasando in italiano possiamo riassumere dicendo: "il fatto che socrate è mortale è conseguenza logica del fatto che socrate è un uomo e della regola che tutti gli uomini sono mortali".

È opportuno precisare subito, anche se si può intuire dì per sé, che le conseguenze logiche derivabili da un programma sono relative a quanto espresso nel programma stesso, e dipendono quindi dalla rappresentazione scelta. In riferimento all'esempio precedente, il quesito:

?- greco(socrate). /* socrate è greco? */

ha risposta negativa, cioè non è conseguenza logica del programma, in quanto non è presente in esso alcun predicato greco con il quale sia possibile la risoluzione del quesito. Il fatto che socrate fosse greco non è ovviamente falso in assoluto, ma rispetto alla rappresentazione fatta risulta falso, nel senso di "non dimostrabile".

Il procedimento di confutazione mediante risoluzione consente diverse possibilità nel formulare un quesito e nell'ottenere delle risposte Consideriamo il seguente esempio:

```

nonno(X, Z) :- padre(X, Y), padre(Y, Z).
padre(mario, carlo).
padre(carlo, ugo).

```

Nel seguito si illustreranno le dimostrazioni mediante schemi a più righe ed a tre colonne; nelle prime due colonne si scriveranno le clausole genitrici e nella terza la sostituzione. Nella prima colonna della prima riga si scriverà il quesito, nella prima colonna delle righe successive si scriverà la risolvente delle clausole genitrici della riga precedente con la sostituzione indicata. Nell'ultima riga si avrà la clausola vuota (se la dimostrazione giunge a termine) e la risposta ottenuta.

Come primo caso consideriamo il quesito:

?- **nonno(mario, ugo).**

Si ha:

:-nonno(mario, ugo)	nonno(X, Z):- padre(X, Y),	X/mario, Z/ugo
	padre(Y, Z)	
:-padre(mario, Y), padre(Y,	padre(mario, carlo):-	Y/carlo
ugo)		
:-padre(mario, carlo),	padre(carlo, ugo)	-
padre(carlo, ugo)		
:-		RISPOSTA: sì

Avendo derivato la clausola vuota, la risposta è affermativa: **nonno(mario, ugo)** è conseguenza logica delle tre clausole di partenza. Parafrasando in italiano: "è vero che mario è nonno di ugo" (relativamente alle proposizioni di partenza).

Sono qui da notare due aspetti. Il primo è che la sostituzione di variabili che consente di rendere un predicato di una clausola identico ad un predicato dell'altra va applicata coerentemente a tutti gli altri prediciati della clausola che contengono le stesse variabili; nell'esempio, la sostituzione **X/mario, Z/ugo** che consente di rendere il predicato del quesito identico al predicato di testa della clausola genitrice è applicata anche ai due prediciati della coda di quest'ultima. La seconda osservazione è che ad ogni passo può porsi la scelta di quale predicato considerare, se ce ne sono diversi, per l'unificazione e quale clausola considerare come genitrice; nell'esempio, nella seconda riga si è considerata la clausola **padre(mario, carlo)** per l'unificazione con **padre(mario, Y)**, piuttosto che **padre(carlo, ugo)** per l'unificazione con **padre(Y, ugo)**.

Consideriamo ora il quesito:

?- **nonno(mario, giovanni).**

Si ha:

:-nonno(mario, giovanni)	nonno(X, Z):- padre(X, Y),	X/mario, Z/giovanni
	padre(Y, Z)	
:-padre(mario, Y) padre(Y,	padre(mario, carlo) :-	Y/carlo
giovanni)		
:- padre(carlo, giovanni)	-	-
		RISPOSTA: no

Poiché tra le clausole di partenza non ce n'è alcuna che possa essere unificata con l'ultima clausola derivata, la risposta è negativa. Parafrasando in italiano: "in base alle informazioni disponibili, non è vero che mario è nonno di giovanni".

Negli esempi precedenti si era posto un quesito chiuso, cioè non contenente variabili, cosicché la risposta possibile era solo sì o no. Più interessante dei precedenti è il caso in cui il quesito è aperto, ossia contiene delle variabili. Allora la risposta non è più solo sì o no, ma fornisce (se la

dimostrazione arriva a conclusione) i nomi degli individui per i quali il quesito ha risposta affermativa, ricavandoli dalle sostituzioni effettuate.

Consideriamo il seguente quesito aperto:

?- nonno(mario, T).

Si ha:

: - nonno(mario, T)	nonno(X, Z) :- padre(X, Y),	X/mario, Z => T
	padre(Y, Z).	
: - padre(mario, Y), padre(Y,	padre(mario, carlo) :-	Y/carlo
Z)		
: - padre(carlo, Z)	padre(carlo, ugo) :-	Z/ugo
		RISPOSTA: T = ugo
: -		

Qui si è indicata con **Z => T** l'unificazione tra le due variabili non istanziate **Z** e **T**: esse vengono messe in corrispondenza, o legate tra loro, cosicché non appena una di esse diventa istanziata, anche a tra lo diventa alla stessa istanza; si dice anche che le variabili sono in condìvisiōne (shared). Ciò consente di riportare alla variabile che appare nel quesito l'istanza individuata, comunque lunga e complessa sia la dimostrazione.

Il risultato ottenuto sì può interpretare nel modo seguente. Il quesito (come clausola negativa) afferma letteralmente: "per nessun **T** (o non esiste alcun **T** per il quale) è vero che mario è nonno di **T**". La dimostrazione per confutazione in questo caso non solo smentisce la proposizione, ma fornisce, mediante l'unificazione, l'istanza di sostituzione che la smentisce (si può dire che fornisce un controesempio, esibendo l'oggetto specifico che confuta la proposizione negata). Parafrasando in italiano, si può così riassumere il risultato: "non è vero che non c'è alcun individuo di cui mario è nonno; infatti, in base alle informazioni di partenza, questo individuo esiste, ed è ugo".

Come si è detto, il quesito può essere visto come proposizione negativa, o - equivalentemente - come proposizione interrogativa sull'esistenza o meno di individui che soddisfano le relazioni richieste. Il quesito precedente può quindi essere letto come: "esiste un individuo **T** di cui mario è nonno ?".

Un'altra possibilità è quella di invertire il quesito, considerando variabile non il secondo termine come nel caso precedente, bensì il primo:

?- nonno(S, ugo). /* esiste un individuo S che è nonno di ugo ? */

In questo caso si ha:

: - nonno(S, ugo)	nonno(X, Z) :- padre(X, Y),	Z/ugo, X => S
	padre(Y, Z)	
: - padre(X, Y), padre(Y,	padre(mario, carlo) :-	X/mario, Y/carlo
ugo)		
: - padre(carlo, ugo)	padre(carlo, ugo) :-	_

:-

RISPOSTA: S = mario

Si noti che la risposta deriva dal fatto che **X**, istanziato a mario, era stato legato a **S**, quindi **S** è istanziato a **mario**, fornendo la risposta.

Una ulteriore possibilità è che nel quesito vi sia più di una variabile; nell'esempio seguente, entrambi i termini sono variabili:

?- nonno(S,T). /* esistono due individui S e T tali che S è nonno di T? */

Si ha:

:- nonno(S, T)	nonno(X, Z) :- padre(X, Y),	X => S, Z => T
	padre(Y, Z)	
:- padre(X, Y), padre(Y, T)	padre(mario, carlo) :-	X/mario, Y/carlo
:- padre(carlo, Z)	padre(carlo, ugo)	Z/ugo
:-		RISPOSTA: S = mario, T = ugo

In generale un quesito può avere più di una risposta, ossia può ammettere più sostituzioni di costanti alle variabili; questo corrisponde al fatto che più n-ple possono soddisfare una data relazione, e viene detto non-determinismo. Supponiamo, per esempio, di aggiungere alle tre clausole iniziali anche le seguenti:

padre(carlo, giorgio).

padre(giorgio, mauro).

ottenendo il programma:

```
nonno(X, Z) :- padre(X, Y), padre(Y, Z).
padre(mario, carlo).
padre(carlo, ugo).
padre(carlo, giorgio).
padre(giorgio, mauro).
```

Il quesito:

?- nonno(mario, T).

in aggiunta alla risposta precedente **T = ugo** ammette ora anche la risposta **T = giorgio**, che si determina in modo del tutto analogo. Possiamo parafrasare questa situazione leggendo ora il quesito come: "di chi è nonno mario ?" e la risposta come: "mario è nonno di ugo e di giorgio" . Il quesito:

?- nonno(S, T).

oltre alle risposte ottenute in precedenza (**S = mario, T = ugo; S = mario, T = giorgio**), ammette in questo caso l'ulteriore risposta **S = carlo, T = mauro**. Possiamo leggere quindi il quesito come: "chi è nonno di chi?" e la risposta come: "mario è nonno di ugo e di giorgio, e carlo è nonno di mauro". Ovviamente le alternative che si presentano nella scelta dei predicati da considerare per l'unificazione sono ora maggiori. Risulta così più evidente che la dimostrazione per confutazione con risoluzione è un processo combinatorico, intrinsecamente non deterministico.

Continuando con l'esempio considerato, supponiamo di aggiungere ancora alle precedenti la clausola:

bisnonno(X, Z) :- padre(X, Y), nonno(Y, Z).

ottenendo il programma:

```
nonno(X, Z) :- padre(X, Y), padre(Y, Z).
padre(mario, carlo).
padre(carlo, ugo).
padre(carlo, giorgio).
padre(giorgio, mauro).
bisnonno(X, Z) :- padre(X, Y), nonno(Y, Z).
```

E facile vedere che il quesito:

?- bisnonno(S, T). /* chi è bisnonno di chi? */

ammette l'unica soluzione **S = mario, T = mauro**.

Le relazioni di **nonno** e **bisnonno** sono casi particolari della relazione più generale **antenato**, che si può esprimere come:

antenato(X, Y) :- padre(X, Y). /* X è antenato di Y se X è padre di Y

antenato(X, Y) :- padre(Z, Y), antenato(X, Z). /* o è antenato di qualche Z che è padre di Y */

aggiungendo a tali clausole i fatti del programma precedente si ottiene:

```
antenato(X, Y) :- padre(X, Y). /* X è antenato di Y se X è padre di Y */
antenato(X, Y) :- padre(Z, Y), antenato(X, Z). /* o è antenato di qualche Z che è
padre di Y */
padre(mario, carlo).
padre(carlo, ugo).
padre(carlo, giorgio).
padre(giorgio, mauro).
```

Il quesito:

?- antenato(S, T).

ammette ora molteplici risposte, in cui quelle prima considerate sono tutte contenute come casi particolari:

S = mario, T = carlo; (soddisfano la relazione padre)

S = carlo, T = ugo;

S = carlo, T = giorgio;

S = giorgio, T = mauro;

S = mario, T = ugo; (relazione nonno)

S = mario, T = giorgio;

S = carlo, T = mauro;

S = mario, T = mauro; (relazione bisnonno)

Volendo questa generalizzazione, le clausole che esprimono la relazione antenato vanno ovviamente sostituite, e non aggiunte, a quelle per nonno e bisnonno, che altrimenti sarebbero ridondanti con esse.

Si noti che l'unificazione, una volta effettuata, è irreversibile, ovvero vale fino a quando si giunge alla conclusione (positiva o negativa) di una risposta al quesito. Per le eventuali risposte successive invece il procedimento riprende da capo, annullando le sostituzioni precedenti e cercandone di nuove.

Nella sua forma generale, un quesito consiste di una congiunzione di condizioni. Sempre in riferimento all'esempio precedente, si può porre il quesito:

?- padre(mario, T), padre(T, Z).

Esso ammette ovviamente le risposte già viste, anche se non menziona esplicitamente la relazione **nonno**. L'esempio mostra che un quesito che contiene una congiunzione di condizioni può essere sostituito da un altro costituito da un'unica condizione (diversa dalle precedenti), purché vi sia (o si aggiunga) una clausola avente come testa quest'unica condizione e come corpo la congiunzione di condizioni sostituita. Naturalmente vale anche il viceversa.

Unificazione.

Il tentativo di unificazione può essere applicato ad una coppia qualsiasi di termini. Come esempi di unificazione di termini che coinvolgono liste, si considerino i seguenti quesiti con le rispettive sostituzioni di risposta, a partire dal fatto:

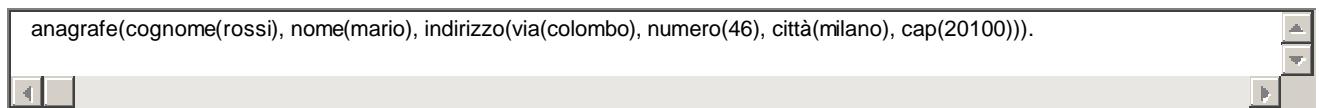
lettere([a, b, c, d]).

?- lettere(X).

X = [a, b, c, d]

- ?- lettere([T|C]).** **T = a C = [b, c, d]**
?- lettere([X, Y |C]). **X = a Y = b C = [c, d]**
?- lettere([T|_]). **T = a**

Mediante l'unificazione si può accedere alle componenti delle strutture a qualunque livello di profondità desiderato. Come esempi si considerino i seguenti quesiti con le rispettive sostituzioni di risposta, a partire dal fatto:



```
anagrafe(cognome(rossi), nome(mario), indirizzo(via(colombo), numero(46), città(milano), cap(20100))).
```

?- anagrafe(X, Y, Z).

X = cognome(rossi)

Y = nome(mario)

Z = indirizzo (via (colombo), numero(46), città(milano), cap(20100)).

?- anagrafe(cognome(X), nome(Y), indirizzo(Z1, Z2, Z3, Z4)).

X = rossi

Y = mario

Z1 = via(colombo)

Z2 = numero(46)

Z3 = città(milano)

Z4 = cap(20100)

?- anagrafe(cognome(X), nome(Y), indirizzo(via(Z1), numero(Z2), città(Z3), cap(Z4))).

X = rossi

Y = mario

Z1 = colombo

Z2 = 46

Z3 = milano

Z4 = 20100

Come esempio più articolato consideriamo l'unificazione tra i seguenti due termini, nei quali tutte le variabili sono inizialmente libere:

termine(a, prova(X, Y, z))

termine(W, prova(W, b, W))

L'unificazione determina l'istanziamento di **W** ad **a**, la condivisione tra le variabili **W**, **X** e **Z** con il conseguente istanziamento di **X** e **Z** ad **a**, e l'istanziamento di **Y** a **b**.

È poi da osservare che - in armonia con le modalità d'uso della variabile anonima illustrate in [Variabili](#) - due o più variabili anonime presenti entro una stessa clausola, pur essendo denotate dallo stesso simbolo, non sono mai in condivisione fra loro, ossia non rappresentano necessariamente una stessa variabile e pertanto possono venire istanziate, nel seguito della computazione, a termini differenti. Il confronto fra i seguenti termini:

termine(X, X, Y)

termine(_, _, _)

permette di chiarire questo concetto. Nel primo, la presenza della variabile **X** al primo ed al secondo argomento richiede la loro unificazione, che presenta tre possibilità: che le prime due variabili siano già istanziate ad uno stesso termine, o che una sola delle due lo sia - con il risultato di istanziare l'altra allo stesso termine - o, infine, che entrambe le variabili siano libere, con l'effetto di porle in condivisione. L'utilizzo di un diverso nome di variabile per il terzo argomento lo rende invece libero da vincoli con gli altri due, potendo assumere - se ancora non istanziato - lo stesso valore o valori diversi. Nel secondo termine, al contrario, non sono evidenziate tali richieste, cosicché l'unificazione potrà avvenire indipendentemente dagli istanziamenti delle tre componenti interessate.

Le regole generali con cui si effettua l'unificazione di due termini qualsiasi e la sostituzione di termini a variabili sono riassunte nella scheda "Regole di unificazione". Si noti che esse determinano l'unificatore più generale (in un senso simile a quello del massimo comune divisore di due numeri). Si dimostra che l'unificatore più generale di due termini, se esiste, è unico.

Regole di unificazione.

Le regole generali per determinare se e in che modo due termini **T1** e **T2** possono essere unificati, dando eventualmente luogo ad istanziamento e/o condivisione delle loro variabili, sono le seguenti.

Innanzitutto, se il termine **T1** (o **T2**) è una variabile istanziata, vale esattamente come il termine a cui è istanziata, che potrà essere, in generale, una costante, od una struttura, od un'altra variabile. Perciò nel seguito si dirà: "se **T** è un ..." in modo equivalente a: "se **T** è un ... oppure è una variabile istanziata ad un ...".

Se **T1** e **T2** sono entrambi un atomo, od un numero, possono essere unificati se e solo se sono lo stesso atomo, o numero. In particolare, un reale non può essere unificato con un intero, anche se hanno lo stesso valore.

Se **T1** è un termine diverso da una variabile (ovvero è una costante od una struttura) e **T2** è una variabile non istanziata, **T1** e **T2** possono venire unificati, istanziando **T2** a **T1**.

Se **T1** e **T2** sono entrambi strutture, possono essere unificati se e solo se hanno lo stesso funtore principale e lo stesso numero di argomenti, e gli argomenti che si corrispondono per posizione possono essere a loro volta unificati.

Infine, se **T1** e **T2** sono entrambi variabili non istanziate, possono essere unificate: in tal caso rimangono, da quel momento in avanti, in condivisione (ossia legate) l'una con l'altra, pur continuando ad essere non istanziate, e non appena una di esse viene istanziata ad un termine secondo le regole precedenti, anche l'altra lo diviene, ed allo stesso termine.

Si noti il carattere ricorsivo del procedimento di unificazione: le regole di base riguardano l'unificazione di termini di cui uno almeno è un termine semplice; la regola ricorsiva interessa l'unificazione di termini composti, facendo appello all'unificazione dei termini componenti. L'applicazione delle regole di base ha evidentemente luogo o quando i termini di partenza sono semplici, o quando si incontrano termini semplici come argomenti dei termini composti su cui il procedimento è applicato ricorsivamente. Il procedimento di unificazione può quindi facilmente essere espresso anche in Prolog.

È infine da osservare che la regola d'inferenza della risoluzione presenta le due seguenti proprietà essenziali:

- di fondatezza (soundness), o correttezza: non è possibile, mediante risoluzione, derivare da un insieme di proposizioni una proposizione che non è conseguenza logica delle precedenti
- di completezza : se una proposizione è conseguenza logica di un insieme di altre proposizioni, è possibile derivarla da esse mediante risoluzione.

In altri termini, un quesito è conseguenza logica di un dato gruppo di clausole se e solo se il loro insieme ammette una confutazione per risoluzione. In linea di principio, il procedimento fornisce come risposta ad un quesito tutti e soli gli individui (termini) che soddisfano le relazioni espresse dai predicati contenuti nel quesito, relativamente all'insieme di clausole considerate. Una di esse viene descritta nella scheda seguente.

Verifica di occorrenza nell'unificazione.

Si è detto che la risoluzione basata sull'unificazione è una regola d'inferenza corretta e completa. Tuttavia, per ragioni di efficienza, la maggior parte dei sistemi Prolog implementano una versione incompleta dell'algoritmo di unificazione.

Il processo di unificazione deve costruire l'unificatore più generale degli argomenti dati (cioè la sostituzione minima che li rende uguali), o mostrare che non esiste alcun unificatore per essi. Un passo elementare di questo processo è un tentativo di unificare un termine **t**, che non sia una variabile, con una variabile **X**. Un unificatore di **X** e **t** esiste (come termine finito) se e solo se **X** non compare in **t** (altrimenti è un termine infinito). Perciò l'algoritmo di unificazione dovrebbe effettuare, ad ogni passo elementare, una tale verifica di occorrenza (occur check) della variabile nel termine, e - se la variabile risultasse presente nel termine - l'unificazione dovrebbe fallire.

Molti sistemi Prolog non effettuano però tale verifica, non per dimenticanza ma per scelta, poiché essa è molto dispendiosa come tempo di esecuzione (che cresce in modo quadratico al crescere della lunghezza dei termini da unificare, anziché in modo lineare come si ha senza la verifica),

mentre si può ritenere empiricamente che la suddetta situazione non si presenti nella massima parte dei programmi logici. Tuttavia, senza verifica di occorrenza, l'unificazione dà una variabile con un termine che la contiene riesce, dando origine ad un termine circolare. Il tentativo di stampare un termine circolare, o di verificare due termini circolari, causerà allora un ciclo infinito. Pertanto non sono più garantite, in generale, la correttezza e completezza della regola di risoluzione: di conseguenza in qualche caso i risultati di una computazione di un programma logico possono risultare incompatibili con la sua semantica dichiarativa.

Due semplici esempi sono i seguenti (le risposte indicate sono quelle fornite in mancanza della verifica di occorrenza). Date le clausole:

```
prova_1 :- p(X, X).
p(X, f(X)).
```

ed il quesito:

?- prova_1.

si ottiene erroneamente la risposta sì.

Assegnate invece le clausole:

```
prova_2 :- p(X, X).
p(X, f(X)) :- p(X, X).
```

ed il quesito:

?- prova_2.

il programma va in un ciclo infinito, costruendo la struttura ciclica:

f(f(f(f...

Sono note alcune condizioni sufficienti affinché l'omissione della verifica di occorrenza non infici la correttezza e completezza della risoluzione. La più semplice è che in ognuna delle teste delle clausole del programma ogni variabile compaia al massimo una volta. Tale condizione è evidentemente molto restrittiva. Una condizione più generale è basata sulla conoscenza dei ruoli di ingresso e di uscita degli argomenti dei predicati.

Alcune versioni del linguaggio implementano l'algoritmo di risoluzione nelle due varianti (con e senza verifica di occorrenza), permettendo all'utente di decidere (e di indicare all'interprete) se effettuare o meno tale verifica. Un'altra versione del linguaggio, Prolog II, è stata realizzata per utilizzare esplicitamente e con vantaggio le strutture di dati cicliche, considerate come alberi

infiniti; sia l'algoritmo di unificazione che la semantica del linguaggio risultano tuttavia più complessi rispetto al Prolog ordinario.

Note bibliografiche.

La programmazione logica trae origine in buona misura dai progressi nel campo della dimostrazione automatica di teoremi ed in particolare dallo sviluppo del principio di risoluzione di Robinson (1965). Robinson (1983) stesso menziona i contributi precedenti più significativi a questo riguardo.

La semantica formale dei linguaggi di programmazione logica è stata studiata da Van Emden e Kowalski (1976) e da Apt e Van Emden (1982). Una trattazione esauriente degli aspetti formali del linguaggio di programmazione logica è data da Lloyd (1984).

Alcuni aspetti dei limiti espressivi delle clausole di Horn rispetto al linguaggio dei predicati del primo ordine sono discussi in Kowalski (1979a) e Gallaire (1983).

Il problema della verifica di occorrenza nell'unificazione è trattato in Lloyd (1984); una condizione sufficiente, di tipo generale, per l'omissione senza conseguenze di tale verifica è formulata in Deransart e Maluszynski (1985). L'utilizzo di strutture di dati cicliche considerate come alberi infiniti è discusso in Colmerauer (1982).

3. Interpretazione procedurale

Dove si considera l'interpretazione procedurale di un programma logico, la cui esecuzione è vista come una successione di chiamate delle procedure che corrispondono alle relazioni definite. E dove si esamina la strategia con cui il sistema Prolog opera proceduralmente, e le sue conseguenze riguardo all'ordinamento delle clausole e dei predicati all'interno di ogni clausola che occorre considerare ai fini dell'efficienza e della terminazione del programma.

Clausole e procedure.

Nell'[Interpretazione dichiarativa](#) le clausole sono state considerate come definizioni di relazioni tra oggetti, secondo la tradizionale concezione della logica. Un'altra conveniente interpretazione è quella detta procedurale, in quanto considera i predicati come nomi di procedure, i loro argomenti come parametri di procedura, le clausole come dichiarazioni di procedura, le teste delle clausole come punti d'ingresso di procedura e le condizioni dei corpi delle clausole come chiamate (o invocazioni) di procedura.

Una clausola, nella sua forma generale:

p :- p1, ..., pn.

è interpretata proceduralmente come dichiarazione di una procedura di nome p ed il cui corpo è costituito dall'insieme di chiamate di procedura **p1, ..., pn**.

Una clausola asserzionale:

p.

è interpretata come caso particolare di procedura il cui corpo è vuoto.

Una clausola negativa:

:- p1, ..., pn.

è interpretata come una procedura senza nome.

La clausola vuota:

:-

è interpretata come una procedura senza nome con corpo vuoto. Mentre nell'interpretazione dichiarativa una clausola è considerata come una proposizione, nell'interpretazione procedurale una clausola è considerata come un'istruzione da eseguire. Una relazione definita da una o più clausole aventi come testa lo stesso predicato costituisce ora una procedura, avendo uno o più punti d'ingresso in corrispondenza all'occorrenza del predicato quale testa di una o più clausole; se la relazione è definita ricorsivamente, la procedura è ricorsiva.

Una clausola negativa, prima considerata un quesito, è interpretata proceduralmente come una meta (goal), ovvero un'istruzione di una o più chiamate di procedura (sottomete) che costituisce l'obiettivo da raggiungere. La clausola vuota è interpretata come istruzione di arresto (halt), ovvero,

come meta soddisfatta. Un programma logico, o insieme di clausole, che nell'interpretazione dichiarativa è visto come consistente di un insieme di (definizioni di) relazioni sottoponibili ad un quesito, nell'interpretazione procedurale viene visto come consistente di un insieme di (definizioni di) procedure eseguibili assegnando una meta iniziale.

Nella lettura dichiarativa le clausole possono essere parafrasate informalmente in italiano come l'affermazione di verità di proposizioni; nella lettura procedurale le clausole possono essere parafrasate come l'esecuzione di operazioni rivolte a raggiungere un obiettivo. Come esempio, si considerino le seguenti due clausole:

```
concatenazione([], L, L).
concatenazione([T|L1], L2, [T|L3]) :- concatenazione(L1, L2, L3).
```

Nell'interpretazione dichiarativa esse definiscono la relazione di concatenazione di due liste. Si può parafrasare la prima clausola dicendo che (è vero che) la concatenazione di una lista vuota ed una qualunque lista **L** è la lista **L** stessa; e la seconda clausola dicendo che, se **L3** è la concatenazione di una qualunque lista **L2** ed una qualunque lista **L1**, allora la lista **[T|L3]** è la concatenazione di **[T|L1]** e **L2**.

Il quesito:

?- concatenazione([a], [b,c], X).

utilizzando come premesse le due clausole che costituiscono proposizioni vere riguardo alla relazione di concatenazione di liste dà luogo ad una dimostrazione costruttiva dell'esistenza di un termine **X** tale che **concatenazione([a], [b,c], X)** è vera. La dimostrazione è costruttiva in quanto porta al risultato di legare **X** al termine **[a, b, c]**.

Che questo termine denoti la concatenazione delle liste è garantito dal fatto che, poiché la derivazione dei risultato è un'inferenza logica che preserva la verità delle proposizioni utilizzate come premesse, esso è un'istanza vera della relazione.

Nell'interpretazione procedurale concatenazione è una procedura ricorsiva con tre parametri e due punti d'ingresso costituiti dalle due clausole, di cui la seconda contiene una chiamata ricorsiva alla stessa procedura. L'illustrazione procedurale della procedura concatenazione può essere espressa come: "concatenando la lista vuota **[]** ad una lista **L** si ottiene la lista **L**; altrimenti, per concatenare una lista **[T|L1]** ad una lista **L2**, occorre dapprima concatenare **L1** ad **L2** ottenendo **L3**, e poi restituire la lista **[T|L3]**".

Questa lettura corrisponde ad interpretare concatenazione come una procedura per la concatenazione di una coppia di liste, come si è fatto assegnando come meta:

?- concatenazione([a], [b, c], X).

ed ottenendo **[a, b, c]**.

Sono però possibili altri modi di utilizzare la procedura concatenazione, che dipendono da quali parametri sono variabili nella meta. Un secondo tipo di uso è per separare una lista data in due sottoliste, per esempio assegnando come meta:

?- concatenazione(X, Y, [a, b]).

In questo caso, ognuna delle tre sostituzioni:

X/[], Y/[a,b]

X/[a], Y/[b]

X/[a, b], Y/[]

è una possibile risposta, in quanto ciascuna denota un'istanza della relazione. Questo utilizzo è meglio espresso dalla seguente lettura procedurale:

"per decomporre la lista vuota, basta restituire la coppia ([], []); altrimenti, per decomporre la lista [T|L3], o si restituisce la coppia ([], [T|L3]) o si decompone L3 nella coppia (L1, L2) e si restituisce la coppia ([T|L1], L2)".

In questa lettura, la presenza dell'alternativa indica le diverse possibilità di esecuzione della procedura.

La meta:

?- concatenazione(X, Y, Z).

rappresenta la ricerca di una qualsiasi tripla di liste nella relazione di concatenazione. Come è facile intuire, esiste un numero infinito di possibili sostituzioni; le risposte restituite, tuttavia, non denoteranno una singola istanza, ma un insieme infinito di istanze. Ciò accade per il fatto che le risposte restituite sono le più generali possibili. Una risposta è:

X/[], Y/Y, Z/Y

che denota l'insieme infinito di triple di lista: ([], L, L), nelle quali L sia una lista qualsiasi. Un'altra risposta è:

X/[U], Y/Y, Z/[U|Y]

che assegna la forma generale dell'insieme infinito di istanze di concatenazione che hanno una lista unitaria quale primo argomento.

Ogni sostituzione per questa meta non denota semplicemente una singola istanza della relazione, bensì un sottoinsieme infinito della relazione. Come programma per la generazione della forma generale di un'istanza della relazione concatenazione, esso può avere la seguente lettura procedurale: "o si restituisce l'istanza ([], L, L), oppure si ricerca un'istanza (L1, L2, L3) e si restituisce la lista ([T|L1 , L2, [T|L3]]) dove T è una variabile che non compare in L1, L2, L3";.

Si noti il diverso comportamento dei parametri rispetto a quelli delle procedure dei linguaggi tradizionali: il loro ruolo d'ingresso e d'uscita non è fissato a priori, ma dipende dal modo in cui la procedura è usata. La possibilità, sopra esemplificata, di utilizzare uno stesso insieme di clausole sia

per il calcolo di una relazione che della sua inversa e, in generale, per la ricerca di una qualsiasi istanza di relazione, è una caratteristica specifica di programmi logici (dovuta alla risoluzione), detta non-determinismo di ingresso/uscita o invertibilità delle procedure.

Esecuzione delle procedure.

Nell'interpretazione procedurale, considerando come parametri formali gli argomenti di un predicato testa di una clausola di definizione di una procedura, e come parametri attuali gli argomenti dello stesso predicato che compare nella chiamata di procedura (meta iniziale o mete derivate), l'unificazione costituisce la modalità di esecuzione della chiamata di procedura. Trovando se esistono, le istanze comuni più generali tra parametri formali e parametri attuali corrispondenti, essa determina una corrispondenza strutturale (pattern matching) che consente la sostituzione della chiamata con il corpo della procedura.

L'unificazione tra la chiamata di procedura **Ai** selezionata in una meta:

: - A1, ... Ai, ..., An

con il nome di una procedura **B** in una clausola:

B :- B1, ..., Bm.

Comporta il trasferimento di valori in ingresso ed in uscita alla procedura. L'istanziamento di variabili che compaiono nel nome di procedura **B** per mezzo di termini che compaiono nella chiamata di procedura **Ai** corrisponde a passare degli ingressi da **Ai** al corpo **B1, ... Bm** della procedura, che diventa così più istanziato. L'istanziamento di variabili che compaiono in **Ai** per mezzo di termini che compaiono in **B** corrisponde a passare indietro delle uscite ad **Ai**, che le distribuisce alle rimanenti chiamate **A1, ..., An** le quali diventano così più istanziate.

Dato un insieme di clausole **C** ed una meta **M**, la computazione è iniziata da **M**, procede utilizzando le dichiarazioni di procedura di **C** per derivare nuove mete, e termina con la derivazione dell'istruzione di arresto. Questo processo si svolge in un ambiente di computazione (binding environment) costituito dalla memorizzazione dei legami di variabili a termini man mano determinati dalle successive chiamate di procedura.

L'esecuzione di un programma logico può essere allora considerata come un procedimento di dimostrazione di tipo costruttivo, con il quale il sistema tenta di dimostrare una clausola meta in un determinato ambiente di computazione. Se una dimostrazione riesce, gli istanziamenti delle variabili che compaiono nella meta assegnata costituiscono il risultato della computazione.

Questa proprietà di preservazione della verità che compete alla computazione di un programma logico è la sua caratteristica più importante, in quanto consente di comprendere un programma logico sia dichiarativamente che proceduralmente, cioè di considerarlo o come un complesso di proposizioni riguardanti una certa relazione o come un metodo per la ricerca di istanze vere di una relazione. La programmazione logica compone in un'unica attività i compiti del costruire una dimostrazione, dell'eseguire un programma e del risolvere un problema.

Regole di selezione e di ricerca.

Esaminiamo più in dettaglio il processo di computazione. Assegnata la meta:

?- p₁, ..., p_n.

un passo di computazione è l'esecuzione di una delle chiamate di procedura **p₁, ..., p_n**. La scelta della chiamata di procedura da eseguire viene effettuata da una regola di selezione, o di computazione, che - per qualsiasi congiunzione di chiamate di procedura incontrate nel corso di una computazione - determina in maniera univoca quale chiamata vada eseguita per prima. Nell'ipotesi che la chiamata selezionata sia **pi**, e che pi sia il predicato **pi(t₁, ..., t_n)**, si effettua un tentativo di eseguire tale chiamata utilizzando una procedura il cui predicato **pi(t₁', ..., t_n')** corrisponde a (match) **pi(t₁, ..., t_n)**, cioè sia possibile unificare i parametri formali con quelli attuali. I casi possibili sono i seguenti:

- Se non esiste alcuna procedura di definizione o essa consiste di un'unica clausola la cui testa non corrisponde alla chiamata, il passo di computazione termina con un fallimento.
- Se la procedura di definizione per pi consiste di una sola clausola con corpo vuoto, la cui testa corrisponde alla chiamata, il passo di computazione termina con successo (ovvero viene derivata la clausola vuota).
- Se la procedura di definizione per pi consiste di un'unica clausola la cui testa corrisponde alla chiamata ed il cui corpo non è vuoto, si ripete il procedimento, dando luogo a nuovi passi di computazione.
- Se (nel caso generale), la procedura di definizione per pi consiste di più clausole, viene effettuato un tentativo di eseguire la chiamata per ciascuna alternativa, che verrà provata secondo un ordine dettato da una ricerca (search rule) che, per qualsiasi insieme di clausole aventi come testa il predicato della chiamata, determina in maniera univoca quale chiamata vada eseguita per prima.

Una computazione di un programma logico attivato da una meta consiste quindi di uno o più passi di computazione, ciascuno dei quali o termina con successo (si dice anche che riesce), o termina con un fallimento (fallisce), o dà luogo ad uno o più altri passi di computazione. La computazione è finita (termina) quando ogni passo di computazione o termina (con successo o con fallimento), o dà luogo a passi di computazione che terminano; oppure è infinita (non termina) se viene eseguito un passo di computazione che dà luogo a passi di computazione che non terminano (anche un programma logico, quindi - come i programmi tradizionali - può andare in ciclo!).

Al meccanismo di chiamata di procedura mediante la corrispondenza strutturale determinata dall'unificazione realizza il non determinismo intrinseco alla logica dei predicati, evidenziato in [Interpretazione dichiarativa](#), in quanto permette che più di una clausola, tra quelle che definiscono una procedura, corrisponda alla chiamata di procedura stessa. Poiché, come si è visto, ad ogni passo di computazione vi possono essere diverse alternative, diverse computazioni possono essere generate a partire da una meta assegnata, con risultati potenzialmente diversi.

Alberi di ricerca e di dimostrazione.

Dati un programma, una meta iniziale ed una regola di computazione, si può rappresentare la ricerca delle alternative possibili nella forma di un albero, detto albero di ricerca (o di valutazione, o albero OR). In tale albero, la radice è la meta iniziale; ogni nodo non terminale è etichettato con la congiunzione delle mete ancora da soddisfare, derivata dal nodo genitore mediante un singolo passo di computazione; i discendenti di un singolo nodo sono le mete alternative derivabili dalla meta presente in quel nodo. La ricerca termina quando tutti i nodi sono terminali; ogni nodo terminale è etichettato con "□" nel caso di terminazione con successo (si è ottenuta la clausola vuota e quindi la meta è soddisfatta), o con "■" nel caso di terminazione con fallimento (la meta non può essere soddisfatta).

Ogni cammino dell'albero (una sequenza di nodi dalla radice ad un nodo terminale) rappresenta una possibile computazione; il risultato è dato dalla composizione dei legami, effettuati lungo il cammino, che interessano le variabili della meta iniziale. Possono esserci cammini infiniti (computazioni che non terminano). Come primo semplice esempio consideriamo le seguenti clausole con predici senza argomenti (procedure senza parametri):

```
a :- b, c. /* clausola 1 */
a :- d. /* clausola 2 */
b :- e. /* clausola 3 */
d. /* clausola 4 */
e. /* clausola 5 */





```

con la meta iniziale:

?- a.

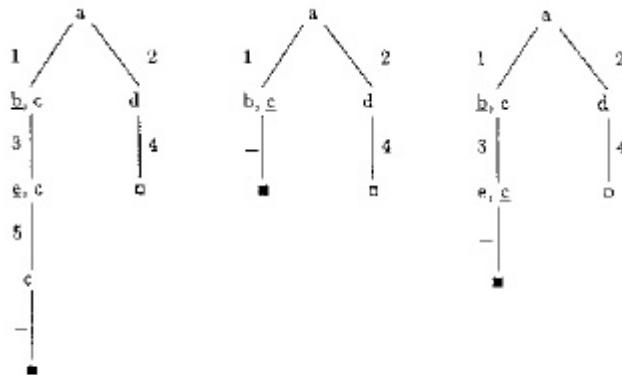


Figura 3.1.

A seconda delle selezioni effettuate dalla regola di computazione i possibili alberi di ricerca sono quelli indicati nella figura (dove nella congiunzione di mete è sottolineato il predicato via via scelto, e sui rami sono indicati i numeri delle clausole applicate).

Per converso, assegnati un programma, una meta iniziale ed una regola di ricerca, si possono rappresentare le sottomete la cui congiunzione costituisce una meta quali rami di un altro tipo di albero, detto albero di dimostrazione (oppure di prova, o di derivazione, od anche albero AND). In tale albero, ogni nodo è una (sotto)meta. La radice ha, come discendenti immediati, le sottomete della meta iniziale; ognuna di queste ultime ha, come discendenti immediati, le sottomete della clausola selezionata. I nodi terminali sono "■" e "□". L'insieme dei nodi immediatamente precedenti quelli terminali dà la congiunzione di sottomete che costituiscono la meta complessiva corrispondente all'albero di dimostrazione.

Rispetto alle clausole ed alla meta iniziale dell'esempio precedente, gli alberi di dimostrazione che corrispondono rispettivamente alla selezione della clausola 1 ed a quella della clausola 2 sono i seguenti (le diramazioni dell'albero AND sono collegate con un archetto, per distinguerlo dall'albero OR):

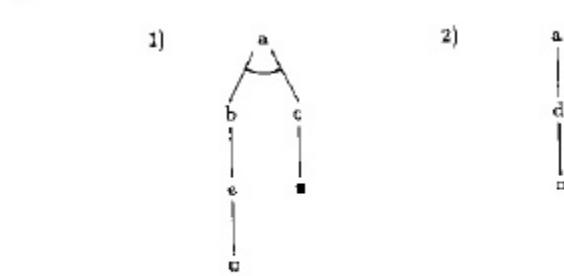


Figura 3.2.

Si noti che nell'albero di dimostrazione non appare l'effetto della regola di ricerca. Quest'ultima influenza il modo in cui l'albero di dimostrazione complessivo è costruito attraverso una successione di progressivi alberi di dimostrazione parziali. Sempre riguardo all'esempio in esame, si hanno le tre seguenti possibili costruzioni dell'albero di dimostrazione 1) sopra mostrato:

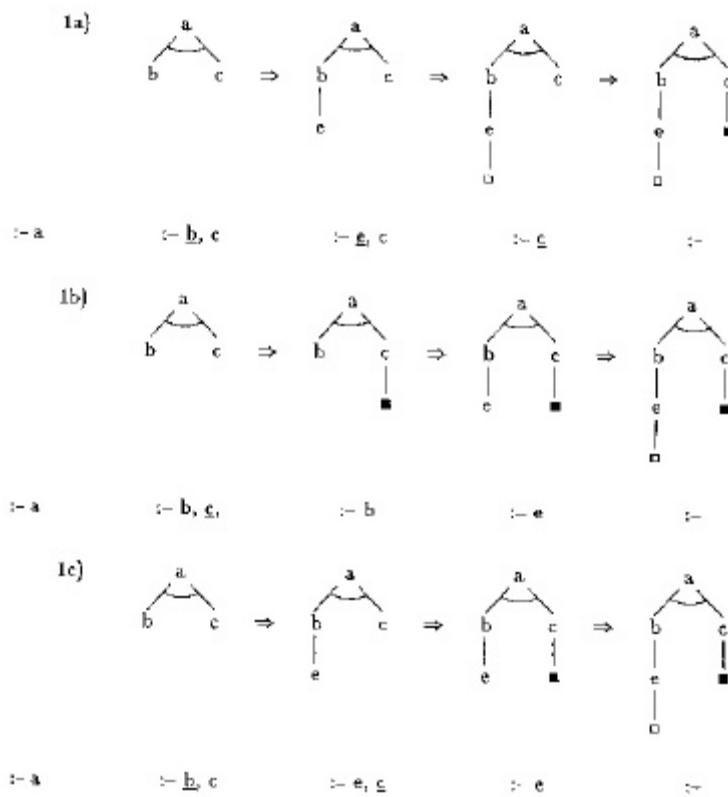


Figura 3.3.

Esse corrispondono ai tre alberi di ricerca precedenti, se si considera che in questi ultimi non venivano espansate ulteriormente le congiunzioni di mete non appena una di esse dava luogo ad un nodo di fallimento.

I due tipi d'informazione forniti dall'albero di dimostrazione (AND), e dall'albero di ricerca (OR), possono in realtà essere rappresentati congiuntamente in un unico albero AND-OR; esso rappresenta lo spazio totale di computazione di un dato insieme di clausole e di una meta iniziale, cioè l'insieme di tutte le computazioni derivabili da essi secondo tutti i possibili modi di selezionare chiamate e procedure.

Per le clausole e la meta iniziale dell'esempio, l'albero AND-OR è il seguente:

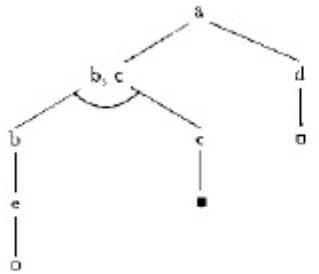


Figura 3.4.

Consideriamo ora il seguente esempio di clausole con argomenti (procedure con parametri):

```
nonno(X, Y) :- padre(X, Z), padre(Z, Y).  
nonno(X, Y) :- padre(X, Z), madre(Z, Y).  
madre(maria, paolo).  
madre(I, J) :- madre(I, K), fratello(K, J).  
padre(giovanni, maria).  
fratello(paolo, pietro).
```

con la meta:

?- nonno(giovanni, pietro).

L'albero AND-OR contiene ora nodi corrispondenti a mete per le quali esistono istanze di sostituzione che fanno corrispondere una chiamata procedura ad un punto d'ingresso di procedura (testa di una clausola con lo stesso predicato). Per l'esempio, l'albero AND-OR è il seguente (sono indicate le istanze di sostituzione che stabiliscono la corrispondenza):

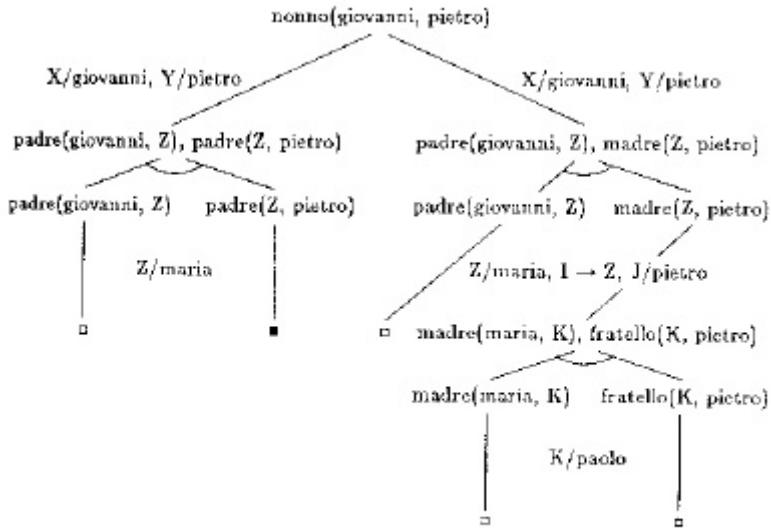


Figura 3.5.

Si noti che l'istanza di sostituzione che dà luogo all'unificazione (o qualsiasi legame di variabili determinato da essa) si applica a tutta le chiamate di procedura che compaiono in una congiunzione di mete (tutti i nodi del sottoalbero AND). Il sottoinsieme delle successive sostituzioni relative ad una computazione terminata con successo, che determina legami per le variabili contenute nella meta iniziale, è la risposta fornita da tale computazione; se tale sottoinsieme è vuoto, in quanto nella meta non sono presenti variabili, la risposta è semplicemente sì.

Strategia standard di Prolog.

Il significato procedurale di un insieme di clausole ed una meta iniziale è così caratterizzato in termini di un meccanismo per calcolare le sostituzioni di risposta, che costituiscono l'uscita della computazione. Tale meccanismo, come si è detto, richiede una strategia nella scelta di quale predicato in una congiunzione di mete considerare per l'unificazione con quale clausola di una procedura avente come testa lo stesso predicato.

Gli interpreti Prolog adottano una strategia prefissata, che consiste in una regola di computazione che seleziona sempre il predicato più a sinistra nella congiunzione di mete da soddisfare, ed in una regola di ricerca che seleziona sempre la prima (nell'ordine di scrittura dall'alto verso il basso) dell'insieme di clausole che costituiscono la procedura definita dal predicato corrispondente al letterale selezionato.

Il sistema Prolog costruisce quindi l'albero AND -OR in un modo specifico, considerando un'alternativa alla volta, operando su una congiunzione di mete in modo sequenziale da sinistra a destra, passando ad una meta successiva solo quando la meta precedente è stata completamente soddisfatta (considerando tutte, le sue sottomete al livello più basso possibile). In caso di fallimento effettua un ritorno indietro (backtracking), cioè un tentativo di utilizzare, uno alla volta, rami alternativi dell'albero (se ne esistono e se non sono stati ancora esplorati) per cercare di effettuare la dimostrazione prima fallita.

Iniziando il ritorno indietro, gli istanziamenti e le condivisioni di variabili già effettuati vengono annullati in ordine inverso, da destra a sinistra. Vengono quindi ricercate altre corrispondenze non con le stesse clausole (in quanto l'algoritmo di unificazione non prevede possibilità alternative),

bensì con clausole differenti da quelle utilizzate in precedenza, con l'obiettivo di ottenere il soddisfacimento della meta fallita.

Nella figura sottostante è riportato l'albero AND - OR dell'ultimo esempio, evidenziando in esso, con numeri progressivi dei nodi, l'ordine con cui viene esplorato dall'interprete Prolog. Partendo dalla meta (nodo 0), mediante la sostituzione X/giovanni, Y/pietro, viene effettuata l'unificazione con la testa della prima clausola della procedura nonno, ottenendo la risolvente (nodo 1). Poiché questa consiste della congiunzione di due sottomete, viene prima esplorata quella più a sinistra (nodo 2) che, mediante la sostituzione Z/maria, riesce (nodo 3). Quindi si passa alla seconda sottometa (rappresentata dal nodo 4), per la quale non esiste, per la stessa istanza di sostituzione di Z, alcuna clausola risolvente. Constatato questo fallimento (nodo 5), che comporta il fallimento complessivo della congiunzione di mete del nodo 1, si ritorna al nodo 0 per provare, in alternativa, la seconda clausola della procedura nono, che conduce ad una nuova congiunzione di sottomete (nodo 6). La prima di queste (nodo 7) riesce con la sostituzione Z/maria (nodo 8); la seconda (nodo 9), con la stessa sostituzione per I (legata a Z) e l'ulteriore sostituzione J/pietro, dà luogo ad un'altra congiunzione di sottomete (nodo 10), che riescono entrambe con la sostituzione K/paolo (nell'ordine, nodi 11-12-13-14).

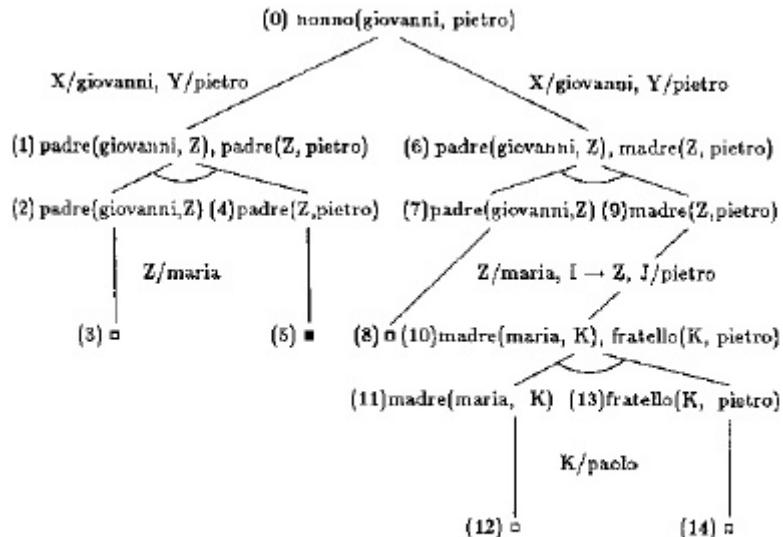


Figura 3.6.

Il punto di ritorno indietro (detto anche punto di fallimento, o punto di scelta) è il nodo dell'albero più vicino dal quale sono possibili alternative non ancora esplorate, cioè l'ultimo nodo esaminato in ordine di tempo; per tale ragione il ritorno indietro è detto cronologico. Quando il ritorno indietro viene effettuato, tutti gli istanziamenti delle variabili, compiuti da quel punto in poi, vengono annullati.

Si noti come la strategia adottata dall'interprete Prolog comporti una esplorazione dell'albero che privilegia una dicesa alla massima profondità del ramo più a sinistra (depth-first), cioè fino a che riesce o fallisce la meta più a sinistra, prima di passare a considerare le altre. Questo modo di procedere viene riassuntivamente chiamato strategia in profondità da sinistra a destra con ritorno indietro cronologico.

Come, ulteriori esempi, che serviranno ad evidenziare altri aspetti del meccanismo di ricerca del meccanismo di ricerca dell'interprete Prolog, consideriamo la procedura [concatenazione](#) già esaminata, con la meta iniziale:

?- concatenazione([a], b, X).

L'albero corrispondente è illustrato nella figura successiva. In esso, il primo ramo più a sinistra rappresenta il tentativo di far corrispondere la meta iniziale con la prima clausola:

concatenazione([], L, L).

tentativo che fallisce perché la lista vuota non può essere unificata con la lista contenente un elemento **[a]**. Il tentativo di corrispondenza con la seconda clausola:

concatenazione([T|L1], L2, [T|L3]) :- concatenazione (L1, L2, L3).

invece riesce, in quanto è possibile la sostituzione **T/a**, **L1/[]** della testa **T** e della coda **L1** della lista **[T|L1]** con la testa **a** e la coda **[]** della lista **[a]**, rispettivamente; è possibile inoltre istanziare **L2** alla lista **[b]**, e legare **X** alla lista **[a|L3]** per effetto della stessa sostituzione **T/a**. Si ottiene così la risolvente:

concatenazione([], [b], L3)

in cui compare la variabile non istanziata **L3**. È possibile ora unificare questa nuova meta con la prima clausola mediante la sostituzione **L/[b]** ed il legame **L3 => L**, ottenendo la risolvente:

concatenazione([], [b], [b]).

La sostituzione di risposta di questo cammino di computazione è: **X/[a|[b]]** (ovvero **X/[a, b]**); infatti **L3**, essendo legata a **L**, ne assume la stessa istanza **[b]**, che quindi sostituisce **L3** nella lista **[a|L3]** cui **X** è legata.

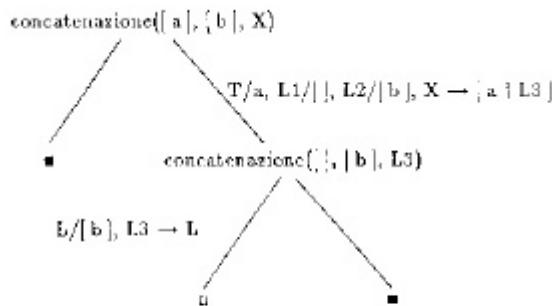


Figura 3.7.

Si vede quindi come la soluzione finale viene prodotta attraverso successive approssimazioni, durante le quali alcune variabili rimangono non istanziate, per diventarlo nelle successive chiamate. Questo comportamento delle variabili di un insieme di clausole, dovuto all'unificazione, viene enfatizzato usando per esse il termine di variabili logiche.

Il ritorno indietro alla metà, concatenazione([b], L3), per esaminare una possibile corrispondenza con la seconda clausola, conduce ad un fallimento. Vi è quindi un'unica risposta alla metà assegnata.

Consideriamo invece, per la stessa procedura concatenazione, la metà iniziale:

?- concatenazione(X, Y, [a, b]).

L'albero esplorato dall'interprete Prolog è in questo caso quello indicato nella figura successiva, dove si sono indicati sotto i nodi terminali di successo le rispettive sostituzioni di risposta, ognuna delle quali dà una delle possibili decomposizioni della lista di partenza in coppie di liste (nella seconda applicazione della seconda clausola si sono rinominate le variabili).

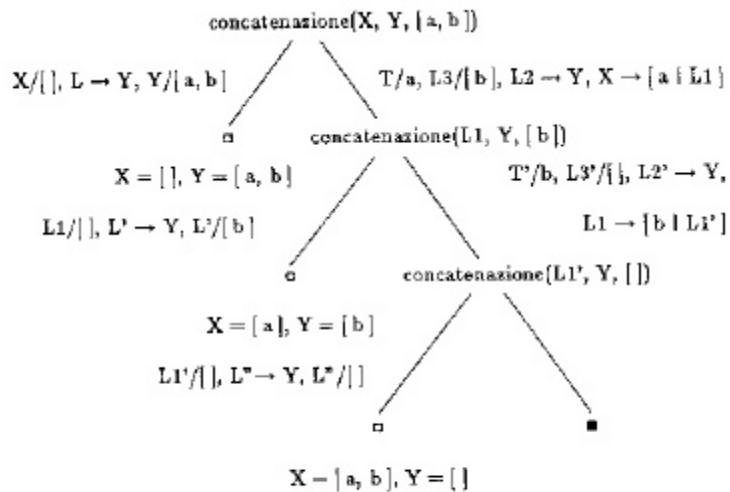


Figura 3.8.

Si noti che, per esaminare l'esistenza di altre possibili risposte una volta trovata la prima, l'interprete Prolog effettua un ritorno indietro analogo a quello che farebbe se il precedente tentativo fallisse; si può dire che simula un fallimento.

Funzionamento dell'interprete Prolog.

Una compuazione dell'interprete Prolog può essere così riassunta:

- È attivata da un quesito, consistente in una o più mete da soddisfare, ovvero in una o più chiamate di procedura. La metà costituisce la prima risolvente (il nodo radice dell'albero di dimostrazione).
- Seleziona il predicato più a sinistra della risolvente (il nodo più a sinistra dell'albero di dimostrazione), ossia la chiamata da eseguire.
- Seleziona la prima clausola (nell'ordine in cui le clausole compaiono nel programma) la cui testa corrisponde al predicato prescelto, unificandoli.
- Sostituisce la testa con il corpo della clausola, propagando gli istanziamenti ottenuti tramite l'unificazione sia al corpo che agli altri predicati della risolvente; rinomina eventualmente le variabili secondo necessità.
- Ottiene così la nuova risolvente ed il nuovo insieme di nodi dell'albero di dimostrazione, che sono discendenti immediati del nodo corrispondente al predicato selezionato. Se il corpo

della clausola è vuoto, il discendente del nodo selezionato è il nodo vuoto " \square "; la risolvente è allora l'insieme dei nodi terminali dell'albero.

- Ricorda la clausola scelta, e quindi l'insieme delle clausole non ancora provate per l'unificazione nel nodo dell'albero corrispondente al predicato selezionato.
- Torna indietro, quando un predicato non trova una corrispondenza, a quel nodo visitato più recentemente che abbia almeno un'altra diramazione.
- Riesce quando la risolvente è vuota, cioè quando non vi sono più nodi da visitare, ovvero tutti i nodi terminali sono " \square ".
- Fallisce parzialmente quando un nodo non trova alcuna corrispondenza, e totalmente quando il nodo che fallisce è la radice dell'albero.
- Trova tutte le soluzioni della meta iniziale simulando il fallimento dopo ogni successo globale della meta assegnata.

I due tipi di scelta descritti precedentemente, cioè la scelta di una meta in una congiunzione e la scelta di una clausola in una procedura, costituiscono due aspetti di non-determinismo della computazione dell'interprete Prolog. Poiché la strategia adottata al riguardo dall'interprete è - come si è detto - prefissata, la computazione che essa effettuerà dipende dalle scelte dell'utente, che decide in quale ordine scrivere le clausole e in quale ordine scrivere le congiunzioni di mete nei loro corpi. Quest'ordine, che come si è detto in [Interpretazione dichiarativa](#) non ha alcuna rilevanza logica, ovvero non ha significato dichiarativo, può avere invece una notevole influenza sull'efficienza, ed anche sull'esito, della computazione, ha cioè un significato procedurale.

Conseguenze della strategia di Prolog sull'efficienza e sulla terminazione.

La regola di computazione non influisce sulla completezza della risoluzione, cioè sull'esistenza e sul numero delle soluzioni da essa determinate: se un insieme di clausole ed un quesito sono incoerenti, la risoluzione trova le istanze di confutazione, indipendentemente da come vengono scelti i predicati nelle congiunzioni dei corpi delle clausole. Tuttavia, tale scelta può influire notevolmente sulla struttura e sulla dimensione dell'albero di ricerca, e quindi sull'efficienza della computazione. Consideriamo ad esempio il seguente insieme di clausole:

The screenshot shows a Prolog interface with the following code in the query window:

```
nonno_a(X, Y) :- genitore(X, Z), genitore(Z, Y).
genitore(X, Y) :- madre(X, Y).
genitore(X, Y) :- padre(X, Y).
padre(ugo, bruno).
padre(bruno, valeria).
padre(carlo, susanna).
padre(dario, fulvio).
madre(lucia, bruno).
madre(anna, valeria).
madre(valeria, susanna).
madre(susanna, fulvio).
```

At the bottom left are navigation buttons (left, right, up, down). On the right side of the interface, there are scroll bars and a vertical toolbar.

con la meta:

```
?- nonno_a(N, valeria). /* chi è nonno_a di Valeria ? */
```

Scegliendo il predicato più a destra si ha l'albero di ricerca della figura successiva (per brevità, vi compaiono solo le iniziali degli atomi).

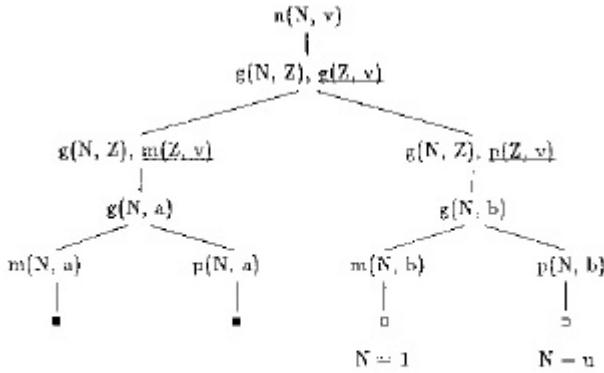


Figura 3.9.

Scegliendo invece il predicato più a sinistra (secondo la strategia standard degli interpreti Prolog), l'albero di ricerca diventa quello della figura successiva (si sono abbreviati i rami ripetuti). Esso contiene le stesse soluzioni del precedente, ma ha un numero di rami ben maggiore.

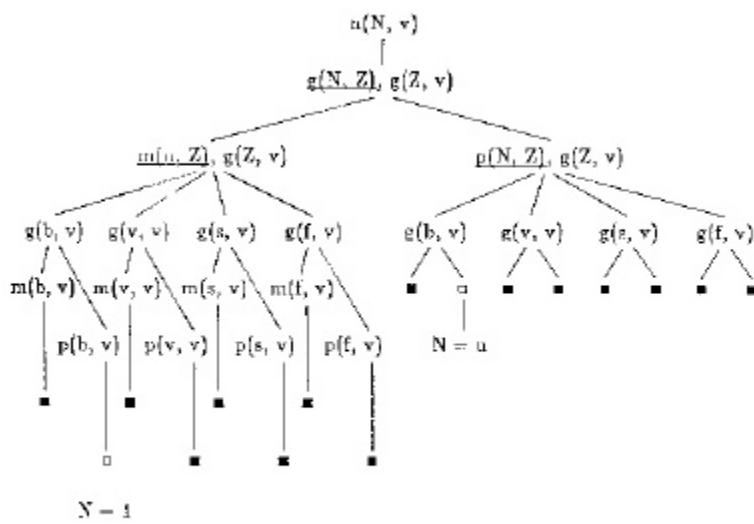


Figura 3.10.

Si noti che, se si considera la meta:

?- nonno_a(lucia, N). /* di chi lucia è nonna ? */

la situazione risulta invertita: selezionando il predicato più a sinistra si ha un albero di ricerca con un numero di rami minore di quello che si ottiene selezionando il più a destra. Non esiste, perciò, una regola di computazione che sia sempre ottimale. Gli alberi di ricerca ottenuti con regole di computazione diverse hanno gli stessi rami di successo, anche se uno può essere finito e l'altro infinito. Consideriamo ad esempio le clausole (non interpretate) seguenti:

```

p(X, Z) :- q(X, Y), p(Y, Z). /* clausola 1 */
p(S, S). /* clausola 2 */
q(a, b). /* clausola 3 */

```

con il quesito:

?- $p(W, b)$.

Si hanno i due alberi di ricerca della figura successiva, scegliendo il predicato più a sinistra o quello più a destra rispettivamente (per ogni nuova applicazione della regola ricorsiva, le variabili sono rinominate).

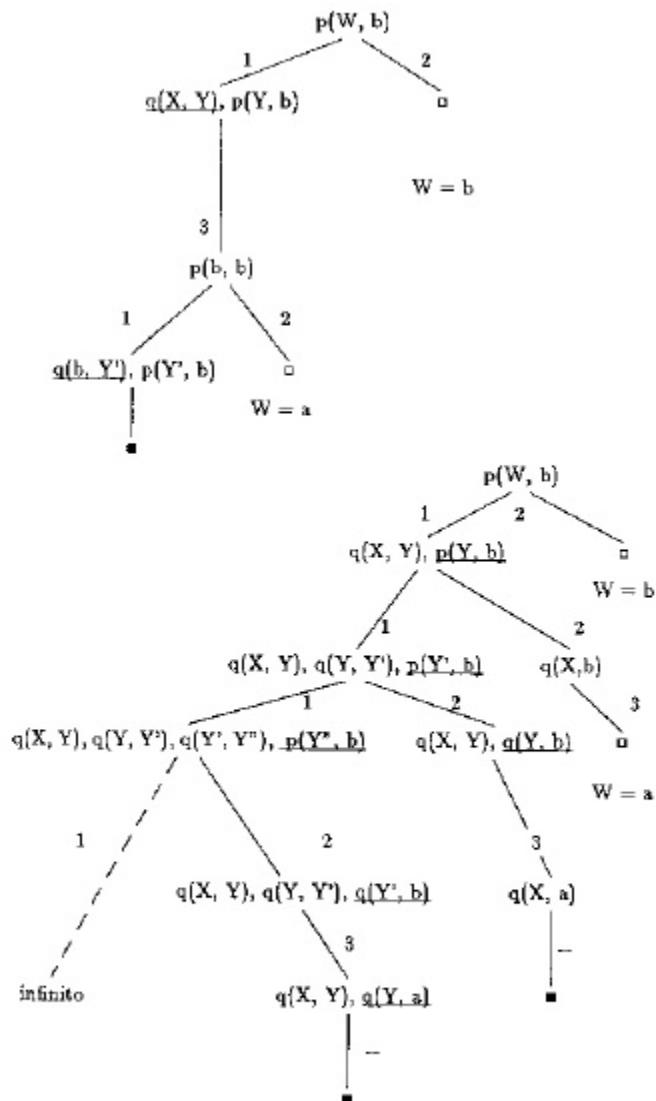


Figura 3.11.

Entrambi hanno gli stessi due rami di successo, ma il primo è finito, mentre il secondo ha un ramo infinito. Anche in questo caso, se nella prima clausola si scambia l'ordine dei due predicati della congiunzione, cioè si considera al suo posto la clausola 1':

p(X, Z) :- p(Y, Z), q(X, Y).

ottenendo:

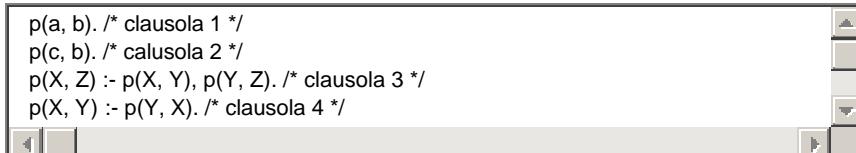


```
p(X, Z) :- p(Y, Z), q(X, Y). /* clausola 1 */
p(S, S). /* clausola 2 */
q(a, b). /* clausola 3 */
```

la situazione si inverte, cioè ha un ramo infinito l'albero costruito selezionando sempre il predicato più a sinistra.

Il fatto che l'albero contenga dei rami di successo non comporta però che essi vengano necessariamente incontrati; questo dipende dalla regola di ricerca. Una regola di ricerca è esaustiva (fair) quando consente ad ogni ramo un'opportunità di essere (prima o poi) esplorato, e trova quindi tutte le soluzioni (i nodi terminali dei rami di successo) prima di perdersi nei rami infiniti. Una ricerca di questo tipo è quella che procede in ampiezza (breadth-first), ossia esplora tutti i rami alternativi di un certo livello dell'albero, prima, di passare al livello successivo. Questa regola è però difficile da implementare, e sono poco frequenti i sistemi che la adottano, in quanto la quantità di memoria necessaria per la dimostrazione può crescere esponenzialmente. La regola di ricerca in profondità, adottata dai sistemi Prolog per semplicità di implementazione, non è esaustiva, in quanto - se incontra un ramo infinito - non può più passare ad esplorare altri rami, che tuttavia possono essere rami di successo. Perciò tale regola può dar luogo a computazioni che non terminano, e fa venir meno in generale la completezza della risoluzione, cioè la possibilità di derivare tutti i fatti che sono conseguenza logica del programma. Nell'esempio precedente, la clausola 1') insieme con la 2) e la 3) scritte in quest'ordine, dà luogo ad una computazione infinita da parte dell'interprete Prolog. Per evitarlo è tuttavia sufficiente invertire l'ordine di scrittura, ossia porre la 1') dopo la 2) e la 3').

Si vede quindi come, mentre il significato dichiarativo di un insieme di clausole è indifferente all'ordine in cui sono scritte, il significato procedurale (l'effetto sulla computazione) ne può dipendere in modo vitale. Si noti che non sempre è sufficiente cambiare l'ordine delle clausole. Consideriamo per esempio le clausole seguenti:



```
p(a, b). /* clausola 1 */
p(c, b). /* clausola 2 */
p(X, Z) :- p(X, Y), p(Y, Z). /* clausola 3 */
p(X, Y) :- p(Y, X). /* clausola 4 */
```

con la meta:

?- p(a, c).

Esse ammettono la seguente confutazione:

:= p (a, c).

:= p(a, Y), p(Y, c). (applicando la clausola 3)

:= p(b, c). (applicando la clausola 1)

:= p(c, b). (applicando la clausola 4)

:= (applicando la clausola 2).

Ma è facile verificare che, adottando la strategia di ricerca in profondità, comunque siano ordinate le clausole e qualunque sia la regola di computazione, tale confutazione non può essere trovata, perché il ramo più a sinistra del corrispondente albero di ricerca è infinito.

Le considerazioni precedenti evidenziano che quando - come nel caso del Prolog standard - l'interprete del linguaggio di programmazione logica adotta la strategia di ricerca in profondità, il compito di scrivere le clausole in modo da evitare inefficienze o computazioni infinite è interamente affidato al programmatore, ed è un compito che richiede attenzione ed esperienza.

La programmazione logica si articola quindi in due componenti, entrambe importanti: la componente logica, che corrisponde al significato dichiarativo delle clausole, e la componente di controllo, che corrisponde al significato procedurale relativo alla strategia di ricerca dell'interprete, cioè concerne i modi con cui la componente logica viene utilizzata per giungere alla risoluzione dei problemi.

A differenza dei linguaggi tradizionali, queste due componenti sono chiaramente distinte. L'efficienza di un programma logico può spesso venire migliorata modificando la componente di controllo, senza modificare la componente logica e quindi il significato del programma.

Implementazione della strategia di Prolog.

La strategia Prolog è basata su considerazioni di facilità ed efficienza Privilegiando la discesa in profondità, essi considerano sempre come meta corrente la più recente meta attiva, ovvero la meta derivata per ultima per la quale esistono delle procedure ancora da provare.

Questa strategia consente una semplificazione delle informazioni memorizzate durante l'esecuzione. In tal modo, infatti, tutte le mete attive si trovano sul ramo dell'albero di dimostrazione che conduce dalla radice alla meta corrente, ed è quindi sufficiente conservare l'albero di dimostrazione che corrisponde alla meta corrente.

La sottometa selezionata nella meta attiva, che costituisce l'ultimo punto di ritorno indietro, dà luogo ad un'estensione dell'albero. Quando esso diviene inattivo (ovvero sono state esplorate tutte le sue diramazioni), si può ripristinare la meta attiva precedente effettuando un ritorno indietro all'ultimo punto di ritorno, cancellando le ultime estensioni con le relative sostituzioni applicate. La strategia in profondità è la più efficiente rispetto all'utilizzo di memoria, perché la maggior parte della memoria allocata relativa ad un ramo può venire riutilizzata al termine dell'esplorazione di quel ramo.

Nell'implementazione, un ramo parzialmente costruito dell'albero di dimostrazione può venire rappresentato mediante una pila (stack) di records di attivazione. L'estensione di tale ramo, al fine di derivare una nuova clausola meta, è allora un'operazione di inserimento (push) nella pila di un

nuovo record di attivazione, inserimento che si ha quando il predicato scelto nella metà che si trova in cima alla pila è unificato con successo con la testa di una clausola. Tale nuovo record di attivazione, che rappresenta la nuova risolvente, contiene un puntatore alla clausola che è stata utilizzata, ed un puntatore all'ambiente di computazione che contiene gli istanziamenti ed i legami, effettuati dall'unificazione, di tutte le nuove variabili introdotte da tale clausola nella computazione.

Un ritorno indietro alla ricerca di una clausola alternativa per il passo precedente della computazione determina allora un'operazione di cancellazione (pop) dalla cima della pila del record di attivazione inserito per ultimo.

Questa rappresentazione a pila delle clausole mete precedenti ed attuali, e questa ricerca in profondità con ritorni indietro che dà luogo ad una successione di inserimenti e cancellazioni sulla pila, corrispondono all'implementazione convenzionale su pila della ricorsione. Data la familiarità di tale meccanismo, tutte le implementazioni di interpreti Prolog lo utilizzano quale strategia di ricerca.

Predicati predefiniti ed effetti collaterali.

Al fine di consentire operazioni di ingresso e di uscita, di manipolazione di termini e della base di dati, di correzione degli errori ed altre funzionalità, ogni sistema Prolog mette a disposizione del programmatore un certo numero di predicatori predefiniti, detti anche predicatori di sistema. Molti di essi, al momento della loro esecuzione come mete, producono degli effetti collaterali che sono in realtà la loro funzione primaria, ossia l'unico motivo per il quale vengono invocati come mete all'interno di una clausola.

Un effetto collaterale (side effect) può essere definito come operazione che non può essere annullata né ripetuta quando un eventuale ritorno indietro porta a raggiungere di nuovo quella meta e tenta di risoddisfarla. Un tentativo di risoddisfare una meta che comporta un effetto collaterale fallisce sempre.

La semantica dichiarativa del linguaggio non tiene in alcun modo conto degli effetti collaterali che possono determinarsi come conseguenza dell'esecuzione di una meta, in quanto, la logica dei predicatori non include tale concetto. Essi sono tuttavia necessari per gli aspetti pratici della programmazione, ed i predicatori di sistema che realizzano hanno un significato procedurale predefinito nel linguaggio.

Per la scrittura si ha il predicato unario **write**: se l'argomento è una variabile istanziata, produce la scrittura del valore al quale la variabile è istanziata; se l'argomento è una stringa (qualunque sequenza di caratteri fra apici) produce la scrittura della stringa stessa. Il predicato senza argomenti nl produce un salto di riga.

Per l'aritmetica è disponibile il predicato binario **is**, scritto in forma infissa:

Ris is Esp

dove **Ris** è una variabile istanziata o meno ad **Esp** è un'espressione aritmetica le cui variabili componenti devono essere istanziate. L'espressione aritmetica **Esp** viene valutata e **Ris** viene istanziata al valore calcolato; si ha un errore di esecuzione se **Ris** è già istanziata, oppure se **Esp** non è un'espressione aritmetica valida o se le variabili contenute in essa non sono tutte istanziate. Nel seguito si useranno nelle espressioni aritmetiche:

- l'operatore unario **-** (cambiamento di segno) e gli operatori binari **+** (somma), **-** (differenza), ***** (prodotto), **/** (divisione reale), **//** (divisione intera), **mod** (resto della divisione intera);
- il predicato unario **sqrt** (radice quadrata);
- gli operatori binari di relazione **=:=** (uguale), **=\=** (diverso da), **<** (minore), **>** (maggiore), **=<** (minore o uguale), **\=** (maggiore o uguale).

Per il confronto di termini vanno utilizzati i predicati binari **==** e **\==**, scritti in forma infissa:

X == Y

X\== Y

che verificano se i termini che istanziano **X** ed **Y** sono letteralmente identici, o diversi, rispettivamente. Si noti che clausole dei tipo:

p(N) :- N == 3.

q(X) :- X == termine.

possono essere sostituite, sfruttando l'unificazione, con:

p(3).

q(termine).

Note bibliografiche.

L'idea di usare la logica come linguaggio di programmazione, mediante l'interpretazione procedurale delle clausole di Horn, fu esposta per primo da Kowalski (1974). Con la nota 'formula' Algorithm = Logic + Control, Kowalski (1979) indicò poi Prolog quale prima approssimazione del paradigma della programmazione logica.

Una sintesi introduttiva alla programmazione logica è costituita da Genesereth e Ginsberg (1985). Colmerauer (1985) ne fornisce un'altra presentazione. Cohen (1985) approfondisce la relazione fra Prolog ed i linguaggi convenzionali.

Sommario.

Lo studente è ora edotto dei vincoli di natura procedurale ai quali deve porre attenzione nella descrizione di un problema. Conosce inoltre il procedimento adottato dal sistema Prolog nel derivare le risposte ad un quesito eseguendo un programma logico. Ha quindi tutti gli elementi concettualmente rilevanti per porsi l'obiettivo di sviluppare ed eseguire un programma Prolog.

4. Utilizzo del sistema Prolog

Dove alfine si svelano i piccoli segreti pratici del Prolog, e tutto quanto occorre sapere per porre mano all'uso concreto del sistema. Si descrive infatti l'interazione utente-sistema, in termini di come richiamare un programma, porre un quesito e sollecitare una o più risposte, con esempi ditali possibilità.

L'interazione utente-sistema.

Le implementazioni di Prolog consistono per la maggior parte di un interprete del linguaggio; in alcune di esse, oltre all'interprete, è disponibile un compilatore. Nel seguito, per sistema si intenderà l'interprete, mentre non verranno considerati aspetti relativi all'utilizzo di compilatori. Nel diagramma seguente sono schematizzate le componenti più importanti di un generico interprete Prolog e le loro connessioni con il mondo esterno.

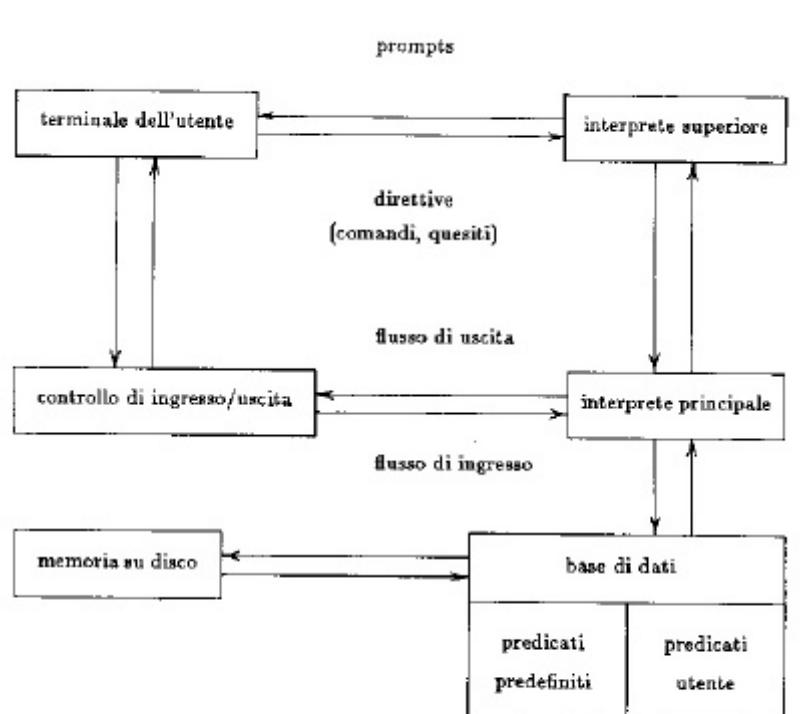


Figura 4.1.

L'interprete è costituito da un componente, detto interprete superiore (top-level), che gestisce l'interazione diretta con l'utente, ed un componente, detto interprete principale, che gestisce le richieste di esecuzione. Queste interessano la base di dati del sistema, che risiede in memoria, e può essere considerata suddivisa in due parti: una nella quale risiedono i predicati predefiniti, realizzati nel linguaggio di implementazione del sistema, e l'altra in cui vengono memorizzati i predicati utente, al momento della lettura del programma. Nel seguito, nei casi di differenza tra le diverse versioni, verranno qui utilizzate le convenzioni adottate dal sistema Prolog/DEC-10 facendole precedere dalla dicitura "ad esempio"; diversamente sono da considerare come convenzioni più generali. S'intende che, per gli aspetti specifici di ogni sistema, va consultato il relativo manuale di utente.

L'accesso all'interprete.

Il sistema sul quale è installato l'interprete Prolog dispone generalmente del comando:

prolog.

o di un comando ad esso equivalente, che invoca l'interprete Prolog. Dopo essere stato caricato in memoria, l'interprete (specificamente, l'interprete superiore) emette una segnalazione di identificazione seguita da un prompt, ad esempio "?-". In questo stato l'interprete è pronto a ricevere comandi, o direttive, dell'utente; essi hanno la forma di una congiunzione di mete che fanno riferimento a predicati utente o di sistema. Va ricordato che le mete Prolog devono essere seguite da un punto ("."), e che quindi Prolog non potrà eseguire nulla sino a che l'utente non avrà digitato "." (seguito da un <return>) al termine della direttiva. Nel caso questo manchi, verrà emesso un prompt di continuazione, ad esempio "|", o "-". Alla ricezione della direttiva completa, l'interprete superiore passa il controllo all'interprete principale, che tenta di soddisfare in congiunzione tutte le mete in essa presenti. Poiché la sintassi del linguaggio presuppone usualmente l'utilizzo di un insieme completo di caratteri **ASCII**, ed in particolare fa uso della distinzione fra caratteri minuscoli e maiuscoli, nel seguito assumeremo la disponibilità di una tastiera con tali caratteri. Nell'ambito di tale convenzione (detta **LC**, per *lower case*, o del *full character set*), le variabili vengono normalmente distinte per mezzo di una lettera maiuscola iniziale, mentre gli atomi e gli altri funtori devono cominciare con una lettera minuscola, a meno che non vengano racchiusi entro apici singoli: in tal caso è possibile qualsiasi combinazione di caratteri, anche di spaziatura. Quando invece i caratteri minuscoli non sono disponibili, è necessario adottare la convenzione **NOLC** (*no lower case*): con essa le variabili vengono distinte per mezzo di un carattere iniziale di sottolineatura "_", ed i nomi degli atomi e degli altri funtori, che ora devono essere scritti in maiuscolo, vengono implicitamente tradotti in minuscolo (a meno che non siano racchiusi entro apici singoli). Programmi scritti con la convenzione **NOLC** non sono sintatticamente compatibili con quelli in **LC**. Poiché l'adozione della sintassi standard consente una maggiore facilità di stesura del codice e risulta di migliore leggibilità, la convenzione隐式 è normalmente **LC**. Nei casi di necessità, è possibile passare alla convenzione no lower case invocando la procedura predefinita '**NOLC**', per esempio mediante la direttiva:

?- 'NOLC'.

Per tornare alla convenzione **LC**, si chiama analogamente la procedura '**LC**', per esempio con:

?- 'LC'.

Lettura di programmi.

Il testo di un programma Prolog viene normalmente creato utilizzando un editor di testi e memorizzato in uno o più files. All'interprete Prolog può quindi venire rivolto il comando di leggere quanto è contenuto in essi; tale operazione viene detta consultazione. Per leggere un programma da un file, per esempio di nome prova, si impedisce la direttiva:

?- consult(prova).

oppure, equivalentemente, una direttiva che consiste di una lista contenente il nome del file, nell'esempio:

?- [prova].

La meta così posta termina sempre con successo e determina, da parte dell'interprete, la lettura (consultazione) del programma contenuto nel file prova ed il suo caricamento in memoria con la conseguente aggiunta, nella base di dati del sistema, delle clausole che in esso si trovano, nello stesso ordine relativo. Per consultare due o più files, si può invocare più volte la procedura predefinita **consult**, come nella direttiva:

?- consult(file_1), consult(file_2), consult(file_3).

o, equivalentemente:

?- [file_1, file_2, file_3].

La specificazione del nome del file, per ciascun file consultato, dev'essere un atomo Prolog; se il nome contiene caratteri normalmente non permessi in un atomo, è allora necessario racchiudere l'intera specificazione del file (compresi eventuali prefissi e suffissi) entro apici singoli, per esempio:

?- ['utente_1/dir_3/nome_file', 'prog1.pro', 'file x:4'].

Viene generato un errore di esecuzione se quello fornito non è un nome di file corretto, o se lo è ma non può venire aperto. I files così specificati vengono letti, e le clausole in essi contenute vengono memorizzate in ordine progressivo nella base di dati dell'interprete, pronte per l'esecuzione del programma. Una volta raggiunto l'end-of-file, l'interprete emette un messaggio di conferma dell'avvenuta consultazione, nonché eventualmente del tempo impiegato per la consultazione e del numero di bytes complessivamente occupati dal programma consultato. Non è possibile la consultazione di clausole con nome di predicato e molteplicità uguali a quelli dei prediciati predefiniti. Qualsiasi tentativo in tal senso porterà soltanto ad ottenere segnalazioni di errore, ed in ogni caso non avrà alcun effetto sulle procedure predefinite. La modifica di parti di un programma dopo la sua consultazione è resa possibile dalla riconsultazione dei files che contengono le parti modificate, che si ottiene invocando l'apposita procedura **reconsult**, ad esempio:

?- reconsult(nuovo_programma), reconsult(ciclo1).

o, equivalentemente, nella notazione a lista, facendo precedere il nome dei files dal carattere "-", come in:

?- [-nuovo_programma, -ciclo1].

La differenza tra le operazioni di consultazione e di riconsultazione di un file è rimarchevole, e consiste in quanto segue. Se un file viene consultato, tutte le clausole in esso definite vengono semplicemente aggiunte alla base di dati: di conseguenza, consultando due o più volte lo stesso file, si otterrebbero due o più copie di tutte le clausole. Allorché un file viene riconsultato, invece, le clausole relative a tutte le procedure presenti nel file riconsultato sostituiscono qualsiasi altra clausola già presente, per quelle stesse procedure, nella base di dati: tutte le clausole precedentemente esistenti per quelle procedure vengono soppresse. Il meccanismo di riconsultazione si rende utile per effettuare correzioni nei programmi, evitando il disagio di scaricare e ricaricare nella base di dati Prolog tutti i files interessati al programma. Se le procedure modificate sono in un unico file, la sua riconsultazione comporta la sostituzione della nuova versione delle clausole al posto delle vecchie, nella stessa posizione che esse avevano nella base di dati. S'intende che la riconsultazione non apporterà alcuna modifica alle procedure del file riconsultato che non siano state sottoposte a correzione. Nella scrittura di un programma è

consigliabile fare uso di un certo numero di files. Poiché l'utente edita e successivamente consulta/riconsulta singoli files, risulta conveniente utilizzarli per raggruppare procedure tra loro correlate logicamente, mantenendo collezioni di procedure che rispondono a funzionalità diverse in files separati e dotati di nomi mnemonicamente significativi. In tal modo, ogni programma Prolog consiste di un certo numero di files, ognuno dei quali contiene procedure tra loro correlate. È da osservare che le correzioni divengono problematiche se le clausole definitorie di qualche procedura risultano distribuite su più di un file consultato: la riconsultazione del solo file sottoposto a correzione modificherebbe tutte le clausole aventi quello stesso nome di predicato, senza tenere conto del file nel quale compaiono. Quando un programma raggiunge dimensioni considerevoli, può essere utile porre in un singolo file i comandi per la consultazione di tutti i files componenti il programma: in tal modo l'intero programma può venire caricato nella base di dati mediante la sola consultazione di tale file. Questo può essere fatto molto semplicemente, in quanto le direttive all'interprete, precedute da "`:-`", possono essere memorizzate in un file, eventualmente inframmezzate alle clausole. Consultando tale file, le direttive presenti vengono eseguite man mano che sono incontrate durante la lettura. Se ad esempio nel file start fosse contenuta la direttiva:

```
:- ([dati, file_1, ciclo, file_2, file_3,prove]).
```

una maniera rapida per caricare l'intero programma sarebbe quella di digitare, al livello dell'interprete superiore:

```
?- [start].
```

La consultazione del file start porta infatti alla immediata esecuzione del comando di consultazione dei files **dati**, **file_1**, e così via. Si crea in tal modo la possibilità di effettuare la consultazione di un gran numero di files invocandone esplicitamente solo uno.

Il file di sistema "user".

Le clausole di un programma possono anche essere scritte direttamente da tastiera durante la sessione Prolog, evitando la fase intermedia della loro memorizzazione su un file. A questo scopo è necessario formulare la direttiva:

```
?- consult(user).
```

oppure:

```
?- [user].
```

Con essa viene effettuata una chiamata al file user, predefinito nell'interprete. A tale punto il sistema si trova in attesa di clausole o direttive; per ogni clausola o direttiva viene emesso un prompt specifico, ad esempio "`:`", mentre le linee di continuazione sono distinte da un diverso prompt, ad esempio "`|`". Per terminare va digitato il carattere di *end-of-file*, che chiude user e ne determina la normale consultazione. Questa modalità di immissione diretta delle clausole nella base di dati è da utilizzarsi soltanto se esse sono molto ridotte in numero e se non è necessario potere disporne permanentemente: in genere può essere utile per la sperimentazione di brevi procedure, con il proposito di scrivere il programma finale su un file normalmente creato utilizzando un editor di testi. Anche per il file di sistema user è possibile effettuare la normale operazione di riconsultazione, formulando la direttiva:

```
?- [user].
```

oppure:

?- reconsult(user).

Se il file riconsultato è il file di sistema user, il prompt per ogni nuova clausola o direttiva può essere ancora diverso dai precedenti.

Quesiti e risposte.

Supponiamo che sia stata definita in un file, già consultato dall'interprete, una relazione che descrive la proprietà di appartenenza di elementi ad una lista, nel modo seguente:

```
appartenenza(E, [E|_]).  
appartenenza(E, [_|L]):-appartenenza(E, L).  
  
[<|>] [ ] [ ] [ ]
```

Se le mete specificate in un quesito possono venire soddisfatte, e se non sono presenti variabili, come nel caso seguente:

?- appartenenza(1, [1,2,3,4]).

l'interprete superiore, non appena gli viene restituito il controllo dall'interprete principale, risponde:

yes

e l'esecuzione del quesito termina. Se nel quesito compaiono delle variabili, viene emesso il valore finale di ciascuna, eccezion fatta per le variabili anonime, che non possono essere interessate da alcuna operazione di istanziamento. Per esempio, il quesito:

?- appartenenza(X, [1,2,3,4]).

ha come risposta:

X=1

A questo punto l'interprete si pone in attesa che l'utente indichi se desidera o meno attivare la ricerca di altre eventuali soluzioni; in alcuni sistemi l'interprete richiede questo esplicitamente, con:

more (y/n)?

In caso negativo, l'utente digita <return> (oppure "n"). L'interprete superiore emette allora la risposta:

yes

per indicare il successo del quesito. Se invece l'utente risponde con il carattere ";" seguito da <return> (oppure "y"), forza il ritorno indietro alla ricerca di soluzioni alternative, ossia cerca di

risoddisfare la meta o la congiunzione di mete che compongono il quesito. L'interprete superiore restituisce allora il controllo all'interprete principale, che cerca di risoddisfare le mete presenti nella direttiva. Se non risulta possibile trovare nuove soluzioni che si aggiungano a quelle già trovate in precedenza, l'interprete superiore emette la risposta:

no

Naturalmente tale risposta può essere ottenuta anche subito dopo la formulazione di un quesito, se il sistema non è in grado di trovare alcuna soluzione per esso. Dopo la comparsa delle segnalazioni **yes** o **no**, l'interprete superiore emette il prompt "?-" e si pone in attesa di un'altra direttiva da parte dell'utente. L'esecuzione relativa al quesito precedente si intende in tal modo terminata. Durante l'interazione descritta l'interprete superiore invia i suoi prompts al flusso corrente di uscita (*current output stream*), che potrà essere il video (è quello implicito) od un file in precedenza indicato dall'utente invocando un'apposita procedura di sistema, e legge dal flusso corrente di ingresso (*current input stream*), che a sua volta può essere quello implicito (la tastiera) od un file indicato, In alcune implementazioni, prima della risposta **yes** o **no** e dell'emissione del prompt "?-", i flussi di ingresso e di uscita vengono sempre riportati al flusso implicito user. Di seguito compare, a titolo illustrativo, un possibile frammento di una sessione interattiva; un numero preceduto da "_" è un simbolo generato dal sistema per la rappresentazione interna delle variabili non istanziate.

?- appartenessa(X, [matematica, fisica, informatica]).

X = matematica;

X = fisica;

X = informatica;

no

?- appartenessa(X, [1, 2,a(Y, b)]), appartenessa(X, [c, a(d, Z), e, f]).

X = a(d,b)

Y = d

Z = b (<return> digitato dall'utente)

yes

?- appartenessa(X, [a, b, c(_)]).

X = a;

X = b;

X = c(_1223); (è un numero di sistema)

no

?- appartenessa(b, [a, b, c, d]).

yes

?- appartenenza(e, [a, b, c, d]).

no

?- appartenenza(b, [a, b, c, d]), appartenenza(e, [a, b, c, d]).

no

Nell'ultimo quesito la prima metà viene soddisfatta, ma la seconda fallisce; il ritorno indietro porta allora al tentativo di risoddisfare la prima metà, operazione che fallisce in quanto b è presente entro la lista con una sola occorrenza; di conseguenza, l'intera congiunzione di mete fallisce.

?- appartenenza(e, [a,b,c,d]), appartenenza(b, [a, b, c, d]).

no

Il quesito termina immediatamente con un fallimento per l'impossibilità di soddisfare la prima metà, mentre la seconda non viene neppure tentata.

?- appartenenza(1, X).

X = [1|_1202];

X = [_1202, 1|1208];

X = [_1202, _1208, 1|_1214] <return>

yes

Il quesito richiede la generazione di (almeno) uno schema di lista al quale appartenga l'elemento 1. Per ottenere la prima risposta viene utilizzata la clausola unitaria della definizione per appartenenza: si noti la presenza, per denotare la coda della lista, di un numero di sistema, corrispondentemente al fatto che la coda non è istanziata. Dopo la forzatura del primo ritorno indietro, la seconda risposta viene ottenuta mediante l'utilizzo della clausola ricorsiva: l'elemento 1 è preceduto e seguito da variabili non istanziate, la prima relativa al secondo elemento della lista, la seconda alla coda. Successive richieste di ulteriori soluzioni portano a generare altri schemi di lista, sempre con l'utilizzo della clausola ricorsiva. L'utente può terminare la computazione digitando <return>. Si voglia ora utilizzare la procedura appartenenza per ottenere tutti e soli gli elementi comuni alle tre liste [a, b, c, d, e], [c, a, b] e [a, b, d, e]. Per questo scopo è sufficiente formulare il quesito:

?- appartenenza(X, [a, b, e, d, e]), appartenenza(X, [c, a, b]), appartenenza(X, [a,b,d,e]).

In esso la ricerca di uno stesso elemento è garantita dall'uso di una medesima variabile, X, all'interno di tutte le tre mete da soddisfare in congiunzione. La prima metà genera un elemento appartenente alla prima lista, la seconda controlla che tale elemento appartenga a [c, a, b] e, in caso positivo, tale controllo viene effettuato anche relativamente alla terza lista: se pure quest'ultima metà viene soddisfatta, l'elemento interessato viene fornito in uscita quale soluzione. La prima risposta sarà allora:

X = a

Scegliendo di proseguire con la ricerca di altre soluzioni, si otterrà l'ulteriore uscita:

X = b

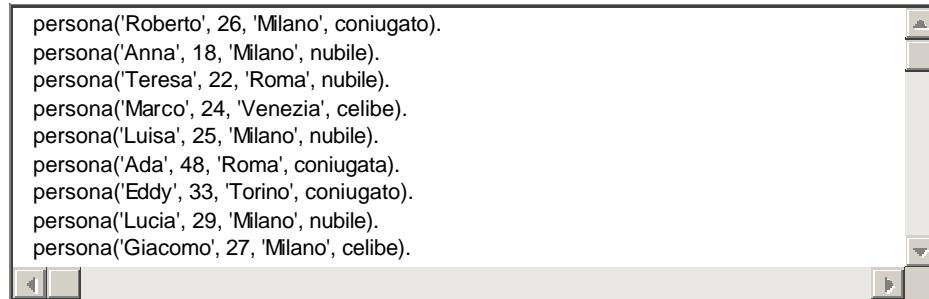
Digitando nuovamente ";", si otterrà infine la segnalazione:

no

indicante che a e b sono i due soli elementi comuni alle tre liste. Utilizzando il meccanismo di unificazione predefinito nei sistemi Prolog è possibile accedere a singole parti di informazioni presenti nella base di dati. Supponiamo di avere già definito e fatto consultare un insieme di clausole del tipo:

persona(Nome, Età, Luogo_di_residenza, Stato_civile).

contenenti dati relativi ad un insieme di persone, ad esempio:



The screenshot shows a window with a scroll bar on the right. Inside the window, there is a list of facts for the `persona` predicate. The facts are:

```
persona('Roberto', 26, 'Milano', coniugato).
persona('Anna', 18, 'Milano', nubile).
persona('Teresa', 22, 'Roma', nubile).
persona('Marco', 24, 'Venezia', celibe).
persona('Luisa', 25, 'Milano', nubile).
persona('Ada', 48, 'Roma', coniugata).
persona('Eddy', 33, 'Torino', coniugato).
persona('Lucia', 29, 'Milano', nubile).
persona('Giacomo', 27, 'Milano', celibe).
```

At the bottom of the window, there are standard window controls (minimize, maximize, close) and a scroll bar.

Possiamo sottoporre questa base di dati ad una serie di domande volte a ricavare da essa diversi tipi di informazioni. Per conoscere il nome di tutte le persone residenti a Milano basterà formulare il quesito:

?- persona(P, _, 'Milano', _).

Il secondo ed il quarto argomento sono stati indicati con una variabile anonima, in quanto riguardano informazioni non pertinenti al problema affrontato dal quesito stesso; l'uso di due variabili annime entro la medesima meta è d'altra parte in accordo con il fatto, già evidenziato, che le variabili anonime non vanno mai considerate in condivisione fra loro. Otterremo come risposte, forzando via via il ritorno indietro:

P = Roberto;

P = Anna;

P = Luisa;

P = Lucia;

P = Giacomo;

no

Volendo conoscere le persone residenti a Milano, di sesso maschile e non sposate:

?- persona(P, _, 'Milano', celibe).

P = Giacomo;

no

In questo caso l'imposizione di un vincolo aggiuntivo, quello relativo al quarto argomento, ha ristretto ad una sola il numero delle soluzioni possibili. Per ottenere una panoramica sul nome e l'età di tutte le persone delle quali sono memorizzati i dati, avremo:

?- persona(P, E, _, _).

P = Roberto

E = 26;

P = Anna

E = 18;

P = Teresa

E = 22;

P = Marco

E = 24;

P = Luisa

E = 25;

P = Ada

E = 48;

P = Eddy

E = 33;

P = Lucia

E = 29;

P = Giacomo

E = 27;

no

che evidenzia la possibilità di ottenere uscite multiple (il nome delle persone e la loro età), mentre la ricerca di persona residenti a Roma e di età non superiore a 40 anni, delle quali si intenda conoscere anche lo stato civile, è facilmente esprimibile con il quesito:

?- persona(P, E, 'Roma', S), E=<40.

P = Teresa

E = 22

S = nubile;

no

Ancora, le persone di età superiore ai 25 anni possono essere ricercate con il quesito:

?- persona(P, E, _, _), E >25.

P = Roberto

E = 26;

P = Ada

E = 48;

P = Eddy

E = 33;

P = Lucia

E = 29;

P = Giacomo

E = 27;

no

Terminazione della sessione.

L'invocazione della meta:

?- halt.

attiva l'omonima procedura predefinita e determina l'uscita dal sistema. Può essere chiamata interattivamente o all'interno del programma. Se si desidera uscire dal sistema mentre un programma si trova ancora in esecuzione, è necessario interromperlo per ritornare all'interprete superiore, e quindi invocare halt. L'utente può uscire dall'interprete anche digitando il carattere di end-of-file in risposta al prompt dell'interprete superiore, o ancora in alcuni sistemi digitando <CTRL C> come primo carattere di una linea di ingresso, che causa la terminazione del programma.

Note bibliografiche.

Per l'utilizzo dei vari sistemi Prolog lo studente è rimandato ai relativi manuali di utente. La maggior parte dei sistemi oggi disponibili in commercio fa comunque riferimento alle versioni DEC-10 (Pereira, Byrd, Pereira e Warren (1979), e Bowen (1982)) e CProlog (Pereira (1982)), realizzate all'università di Edimburgo. Il riferimento classico per gli aspetti implementativi del Prolog/DEC-10 è Warren (1977). Alle problematiche di implementazione di sistemi Prolog è dedicato Campbell (1984).

Sommario.

Lo studente può (finalmente) esercitarsi nell'interrogare piccoli programmi logici, come quelli fin qui considerati o da lui stesso concepiti e realizzati, interagendo con il sistema ed ottenendone risposte. Dopo tale esperienza avrà forse maturato l'esigenza di conoscere più in dettaglio le possibili tecniche di strutturazione di un programma.

5. Strutturazione del controllo.

Dove si descrivono le possibilità di strutturazione del flusso di controllo in un programma, cominciando con le diverse forme della ricorsione, che è il meccanismo principale in Prolog, e considerando poi i modi per realizzare strutture di sequenza, selezione ed iterazione, che sono usualmente le più intuitive e familiari, perché presenti nei più diffusi linguaggi di programmazione.

In questo capitolo è necessariamente prevalente l'aspetto procedurale del linguaggio, quello direttamente correlato al modo di operare dell'interprete, in quanto l'aspetto dichiarativo, correlato alle conseguenze logiche delle clausole, è del tutto trasparente rispetto a come tali conseguenze sono derivate. Tuttavia le caratteristiche della regola d'inferenza di risoluzione hanno anch'esse, come si vedrà, un effetto sul modo in cui il controllo può essere esercitato.

La ricorsione.

Lo strumento maggiormente utilizzato in Prolog per la realizzazione di strutture di programma è la ricorsione. Si è già accennato nella [Rappresentazione di un problema](#) che essa è un modo per definire insiemi infiniti di oggetti, e loro proprietà, mediante descrizioni finite.

Un esempio già considerato è quello della relazione **antenato**. In italiano, un possibile modo per definirla è il seguente: "dato un individuo, i suoi genitori sono suoi antenati; i genitori dei suoi genitori sono suoi antenati, come pure i genitori dei genitori dei genitori, e così via". La corrispondente definizione Prolog è la seguente:

antenato(X, Z):-genitore(X, Z).

antenato(X, Z):-genitore(X, Y), genitore(Y, Z).

antenato(X, Z):-genitore(X, Y), genitore(Y, W), genitore(W, Z).

A parte la lunghezza e ripetitività di una simile scrittura, il problema è che il grado di ascendenza che si può così rappresentare è pari al numero di clausole che si scrivono, e quindi in ogni caso limitato, perché non è possibile scrivere un numero infinito di clausole. D'altro canto non c'è motivo di avere un limite a priori, e di fatto nell'accezione del termine antenato non è in alcun modo contenuto un limite superiore. Al contrario, la lingua naturale stessa consente una definizione che elude tale limite, come la seguente: "dato un individuo, sono suoi antenati o i suoi genitori, o gli antenati dei suoi genitori". Una tale definizione trova in Prolog una naturale corrispondenza, nel modo seguente:

antenato(X, Y):-genitore(X, Y).

antenato(X, Z):-genitore(X, Y), antenato(Y, Z).

Si tratta di una definizione ricorsiva, che consente di definire una relazione vera tra individui senza porre limiti al numero di individui considerati.

Ricorsione e Induzione.

Definizioni di questa natura sono comuni in matematica, dove sono chiamate definizioni induttive e sono usate sia per definire insiemi, sia per definire (e dimostrare) proprietà degli elementi di tali insiemi.

Una definizione induttiva di un insieme di elementi è composta in generale da:

1. una descrizione di uno o più elementi iniziali dell'insieme;
2. un modo per costruire altri elementi dell'insieme, a partire da quelli iniziali;
3. un'affermazione che elementi dell'insieme sono solo quelli ammessi da 1. e 2. (nel seguito questa terza parte sarà considerata implicita, ed omessa).

Per esempio, la definizione dell'insieme **N** dei numeri interi non negativi (o naturali) è la seguente:

1. zero è un elemento di **N**;
2. se **n** è un elemento di **N**, allora il successore di **n** è un elemento di **N**.

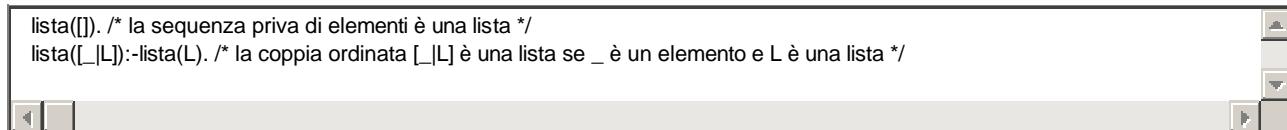
La corrispondente definizione ricorsiva in Prolog è la seguente:

```
numero_naturale(0). /* zero è un numero_naturale */
```

```
numero_naturale(successore(X)):-numero_naturale(X). /* il successore di X è un numero_naturale se X è un numero_naturale */
```

dove **successore** è un funtore ad un posto; **successore(t)** è un termine che denota il numero naturale immediatamente superiore a quello denotato dal termine **t** (il successore di **t** nella successione dei numeri naturali); **X** è una variabile che denota un qualunque numero naturale, cioè il dominio di **X** è l'insieme dei numeri naturali (nella scrittura normale si abbrevia **successore(0)** con "1", **successore(successore(0))** con "2" e così via).

Nella programmazione, insiemi di valori sono spesso detti "tipi" (di dati). Un altro esempio di definizione ricorsiva in Prolog di un "tipo" è quella che definisce una sequenza di elementi, o lista:



```
lista([]). /* la sequenza priva di elementi è una lista */
lista([_|L]):-lista(L). /* la coppia ordinata [_|L] è una lista se _ è un elemento e L è una lista */
```

Come detto sopra, l'induzione in matematica serve anche a definire e dimostrare proprietà degli elementi di un insieme. Il principio di induzione, nella sua forma più semplice, si può esprimere come segue:

Data una proprietà (un predicato) **p** sull'insieme dei numeri naturali, se:

1. **p(0)** è vero, e
2. per ogni **k**, se **p(k)** è vero, allora **p(k + 1)** è vero, allora **p(n)** è vero per ogni **n**.

dove 1. è detto il passo base, e 2. è detto il passo di induzione.

In programmazione logica, la ricorsione svolge un ruolo analogo a quello che l'induzione svolge in matematica. Un tipico stile di definizione ricorsiva di una relazione è quindi quello induttivo che, in forma più generale, può essere considerato consistente di:

1. uno o più casi di base, che stabiliscono il valore della relazione in uno o più casi particolarmente semplici (ad esempio, l'elemento o gli elementi iniziali dell'insieme, o tipo, su cui è definita);
2. uno o più casi generali, che stabiliscono come computare il valore della relazione, dati i valori di uno o più precedenti computazioni di quella relazione.

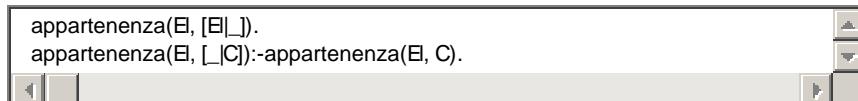
I casi del punto 1. sono espressi con clausole unitarie (asserzioni) e sono detti condizioni limite (boundary conditions) o anche condizioni di terminazione (corrispondono al caso base dell'induzione). I casi in 2. sono espressi con clausole (regole) ricorsive, corrispondenti al passo di induzione.

Le definizioni ricorsive sono particolarmente naturali per relazioni su insiemi definiti ricorsivamente, come i numeri naturali e le liste. Si definisce allora la relazione con un'asserzione per il caso del numero **0** o della lista vuota, e con una regola per il caso di un numero **k** o di una lista di lunghezza **k**, in funzione del caso del numero **k - 1** o della lista di lunghezza **k - 1**, e la relazione vale per un numero qualunque o una lista di lunghezza qualunque. In questi casi risulta chiaramente come una definizione ricorsiva ben formulata definisca una relazione non proprio in termini di sè stessa, come può apparire a prima vista, bensì in termini di una versione più semplice di sè stessa. Applicazioni ripetute della regola ricorsiva riconducono quindi al caso base, che assicura la terminazione.

Comportamento di clausole ricorsive.

Come primo esempio al riguardo, consideriamo la relazione di appartenenza di un elemento ad una lista, espressa dalle clausole seguenti:

```
appartenenza(E, [E|_]).  
appartenenza(E, [_|C]):-appartenenza(E, C).
```



La lettura dichiarativa di questa procedura è la seguente. La prima clausola è l'asserzione che esprime il caso base, costituendo la condizione limite; essa afferma che un elemento **E** appartiene ad una lista la cui testa sia **E** stesso, qualunque sia la coda. La seconda clausola esprime il caso generale, costituendo il passo ricorsivo; essa afferma che un elemento **E** appartiene ad una lista se appartiene alla sua coda, qualunque sia la testa. La relazione di appartenenza che le due clausole definiscono vale qualunque sia la lunghezza della lista. Dichiarativamente, esse implicano che sono vere ad esempio le relazioni:

appartenenza(a, [a,b,c,d]).

appartenenza(c, [a,b,c,d]).

Proceduralmente, la meta:

?- appartenenza(a, [a,b,c,d]).

dà luogo a risposta affermativa, mediante la sostituzione **E/a** nella prima clausola. La meta:

?- appartenenza(c, [a,b,c,d]).

determina, dopo il ritorno indietro causato dal fallimento della prima clausola, l'utilizzo della seconda; ogni volta che la seconda clausola viene invocata, la meta risultante riceve in ingresso una lista dotata di un elemento in meno rispetto alla meta precedente. Non appena **c** è divenuto il primo elemento della lista, viene utilizzata la prima clausola, che - essendo un fatto - non richiede l'invocazione di alcuna sottometta. L'albero di prova è quindi:

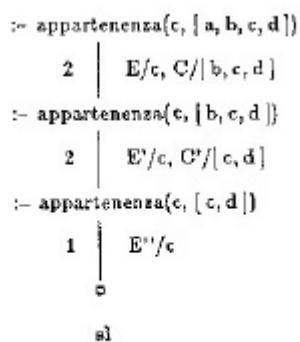


Figura 5.1.

Un quesito in cui il primo argomento non è istanziato ha tante soluzioni quanti sono gli elementi della lista fornita come secondo argomento. Consideriamo ad esempio il quesito:

?- appartenenza(X, [a,b,c]).

La prima soluzione si trova con una corrispondenza con la prima clausola, che istanzia **X** alla testa della lista, cioè ad **a**. Se si desidera un'altra soluzione, la prima clausola non può più fornirla direttamente: la ricerca continua quindi con la seconda clausola che istanzia **C** a **[b,c]** e, essendo ricorsiva, comporta un ritorno indietro alla prima clausola; questa, operando ora sulla lista **[b,c]**, fornisce come seconda soluzione la sua testa, cioè **b**. Analogamente per la terza soluzione, **X = c**. L'albero di prova è il seguente:

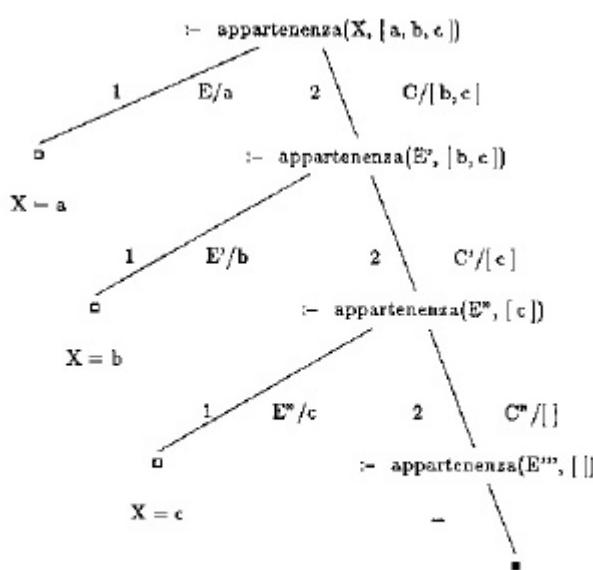


Figura 5.2.

È da notare che nell'esempio la clausola ricorsiva non fornisce mai direttamente una soluzione; la sua funzione è di rimuovere successivamente la testa della lista e determinare un ritorno indietro alla condizione limite, che può allora fornire soluzioni successive.

Efficienza e terminazione nelle definizioni ricorsive.

Si è già osservato nell'[Interpretazione dichiarativa](#) che dichiarativamente l'ordine delle clausole non è rilevante, mentre è importante proceduralmente.

Nell'esempio, invertendo le clausole:

appartenenza(E, [_ C]):-appartenenza(E, C). appartenenza(E, [E _]).	<input type="button" value="▲"/> <input type="button" value="▼"/>
--	--

appartenenza(El, [_|C]):-appartenenza(El, C).

appartenenza(El, [El|_]).

e considerando la meta:

?- appartenenza(a, [a|_]).

si entra in un ciclo senza fine, in quanto l'interprete Prolog costruisce un albero di prova che ha un unico ramo infinito:

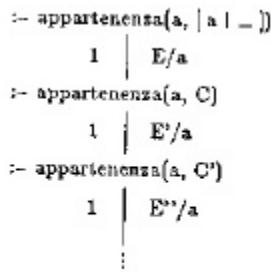


Figura 5.3.

In generale è consigliabile anteporre le condizioni limite alle clausole ricorsive; ciò consente di evitare che il sistema, continuando a riutilizzare le regole ricorsive, entri in un ciclo infinito. Tuttavia questo comporta un più alto numero di ritorni indietro.

Consideriamo la procedura ricorsiva **stessa_lunghezza**, definita come segue:

```

stessa_lunghezza([_|C1], [_|C2]):-stessa_lunghezza(C1, C2).
stessa_lunghezza([],[]).

```

La meta **stessa_lunghezza(L1, L2)** è soddisfatta se le due variabili **L1** e **L2** sono istanziate a liste aventi lo stesso numero di elementi. La prima clausola, ricorsiva, esprime il fatto che due liste hanno la stessa lunghezza se, a prescindere da quali siano le loro teste, tale proprietà è rispettata dalle loro code (si ricordi che variabili anonime che figurino all'interno di una stessa clausola non sono mai in condivisione tra loro). La seconda clausola afferma che due liste vuote hanno la stessa lunghezza, e fa da condizione di terminazione della procedura ricorsiva.

Ragioni di efficienza consigliano, in questo caso, di posporre la condizione limite alla regola ricorsiva: infatti in tal modo la condizione limite viene utilizzata soltanto in corrispondenza dell'ultima chiamata, cosicché il risoddisfacimento di un quesito richiede un solo ritorno indietro, a fronte degli **n - 1** (dove **n** è la lunghezza comune alle due liste) che sarebbero necessari se l'ordine relativo delle due clausole fosse invertito.

Considerazioni opposte valgono per la procedura:

```

stessa_posizione(_, [E|_], [E|_]).
stessa_posizione(_, [_|C1], [_|C2]):-stessa_posizione(_, C1, C2).

```

una chiamata della quale riesce se il termine a primo argomento compare (almeno una volta) nella stessa posizione entro le due liste. In questo caso è consigliabile anteporre la condizione limite, in maniera che la successione di chiamate si arresti non appena si giunge a trovare, nelle due liste, l'elemento desiderato.

Un semplice esempio che riguarda liste e numeri insieme è la procedura per determinare la lunghezza di una lista. La relazione **lunghezza(L, N)** stabilisce che la lista **L** ha **N** elementi. La sua definizione è immediata:

```
lunghezza([], 0).
lunghezza([_|C], N):-lunghezza(C, N1), N is N1 + 1.
```

Si noti che le due sottomete della seconda clausola non possono essere scambiate di posto, perché **N1** deve essere istanziato prima che la sottometa **N is N1 + 1** possa essere eseguita. Il predicato predefinito **is** introduce perciò un vincolo procedurale non trascurabile.

Esecuzione in avanti ed all'indietro di procedure ricorsive.

Un esempio numerico tradizionale è quello della funzione intera fattoriale. La sua definizione è la seguente:

1. la funzione è definita per i numeri interi non negativi;
2. il valore della funzione per **0** è **1**;
3. il valore della funzione per **n > 0** è **n** volte il valore della funzione per **n - 1**.

Le clausole Prolog corrispondenti sono:

```
fattoriale(0, 1).
fattoriale(N, F):-N>0, N1 is N - 1, fattoriale(N1, F1), F is N * F1.
```

Esse si basano sull'idea, tipica della ricorsione, che è più facile calcolare il fattoriale di **n - 1** che il fattoriale di **n**; la seconda clausola, dopo aver controllato che il primo argomento sia positivo, lo diminuisce di una unità e passa ricorsivamente alla ricerca del fattoriale del numero così ottenuto, sino a che viene raggiunta la condizione limite, che fornisce il risultato desiderato.

Dichiarativamente, la procedura è utilizzabile per verificare la relazione tra due numeri assegnati, o per calcolare il fattoriale di un numero assegnato, o per risalire al numero del quale è assegnato il fattoriale. Proceduralmente, quest'ultimo uso non è però possibile, perché le espressioni delle prime due condizioni risulterebbero non valutabili, non essendo **N** istanziata. Sono quindi possibili i quesiti:

?- fattoriale(5, 120).

?- fattoriale(5, X).

ma non il quesito:

?- fattoriale(N, 120).

L'esecuzione della procedura viene effettuata nel modo tipico dell'interprete Prolog, cioè partendo dalla metà, riducendola a sottomete sempre più semplici, fino alla corrispondenza con il fatto. Questo modo di esecuzione è detto dall'alto verso l'alto (top-down), ovvero all'indietro (backward), o ancora di analisi, con riferimento al procedere dalla conclusione verso le premesse ed all'analizzare il problema in termini di sottoproblemi.

Un altro modo generalmente possibile per calcolare il fattoriale sarebbe quello di partire dal passo di base ("il fattoriale di **0** è **1**") e calcolare progressivamente i fattoriali dei numeri successivi, ogni volta utilizzando quelli calcolati prima, fino a N. Questo modo di esecuzione è l'inverso del precedente, in quanto si comincia dal fatto e si costruiscono fatti ulteriori, fino a raggiungere la metà. Esso è detto dal basso verso l'alto (bottom-up), ovvero in avanti (forward), o ancora di sintesi, con riferimento al procedere dalle premesse verso la conclusione ed al sintetizzare nuova informazione da quella preesistente.

Poiché, come si è visto, l'interprete Prolog opera intrinsecamente dall'alto verso il basso, una procedura che venga da esso eseguita dal basso verso l'alto non può essere realizzata in modo diretto; può però esserlo indirettamente, mediante una relazione ausiliaria, chiamiamola **fatt**, il cui effetto sia quello di simulare l'esecuzione in avanti, accumulando mediante opportuni argomenti aggiuntivi i fattoriali parziali via via ottenuti; la relazione **fattoriale_1** viene quindi definita mediante **fatt**, incorporando nella sottometa l'informazione che il fattoriale di **0** è **1**. La realizzazione è la seguente:

```
fattoriale_1(N, F):-fatt(N, F, 0, 1).
fatt(N, F, N, F).
fatt(N, F, N1, F1):-N2 is N1 + 1, F2 is N2*F1, fatt(N, F, N2, F2).
```

Il predicato ausiliario **fatt(N, F, N1, F1)** esprime la relazione: "il fattoriale di **N** è **F** se il fattoriale di **N1** è **F1**". In questo caso non interessa però la lettura dichiarativa, che anzi risulta meno chiara rispetto alla precedente formulazione, bensì il fatto che l'esecuzione è più efficiente. Infatti, nel caso precedente, l'interprete mette nello stack n records di attivazione della sottometa fattoriale, che vengono poi rimossi alla risoluzione delle sottomete **N * F1** (che costituiscono il calcolo effettivo del risultato). Nel caso presente l'uso di variabili come accumulatori evita quello dei records di attivazione: ce n'è sempre uno solo, corrispondente a **fatt(N, F, Indice_attuale, Risultato_parziale)**, oltre al record iniziale per **fattoriale_1(N, F)**.

Si noti che gli ultimi due argomenti di **fatt** durante le successive chiamate contengono l'informazione riguardante i fattoriali costruiti progressivamente a partire dal caso base, tipici dell'esecuzione in avanti; tali argomenti fanno quindi da accumulatori dei risultati parziali. Ad esempio, con la metà:

?- fattoriale_1(3, X).

la successione delle chiamate di **fatt** è:

fatt(3, X, 0, 1)

fatt(3, X, 1, 1)

fatt(3, X, 2, 2)

fatt(3, X, 3, 6)

con risposta **X = 6**.

È da osservare che, nella definizione data sopra, la relazione **fattonale_1** a due argomenti è quella da usare nei quesiti, mentre la relazione fatt a quattro argomenti è strumentale rispetto ad essa, e può essere ignorata dal punto di vista dell'utilizzo della procedura in altri punti del programma.

Ricorsione multipla.

Un esempio in cui sono necessarie due condizioni limite è quello della funzione numerica detta di Fibonacci. Essa genera la serie di numeri interi:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34,...

caratterizzata dalla seguente definizione:

1. la funzione è definita per numeri interi non negativi;
2. il valore della funzione per **0** è **0**;
3. il valore della funzione per **1** è **1**;
4. il valore della funzione per **n > 1** è uguale alla somma del valore per **n - 1** e di quello per **n - 2**.

La procedura Prolog con esecuzione dall'alto verso il basso è:

```
fibonacci(0, 0).  
fibonacci(1, 1).  
fibonacci(N, F):-N>1, N1 is N-1, fibonacci(N1, F1), N2 is N-2, fibonacci(N2, F2), F is F1+F2.  
[<] [>] [x] [y]
```

Questa procedura è doppiamente ricorsiva e risulta poco efficiente, in quanto ripete più volte le stesse computazioni, come si può vedere dallo schema dell'albero di prova, ad esempio nel caso del quesito:

?- fibonacci(4, F).

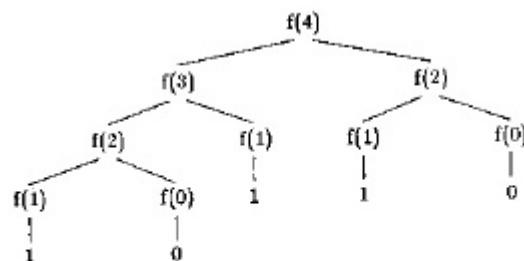


Figura 5.4.

albero i cui nodi (corrispondenti ai principali passi di computazione) crescono esponenzialmente al crescere del primo argomento.

La procedura con esecuzione simulata dal basso verso l'alto è:

```
fibonacci_1(N, F):-fib(1, N, 0, 1, F).
fib(M, N, _, F2, F2):-M>=N.
fib(M, N, F1, F2, F):-M<N, M1 is M+1, FM is F1+F2, fib(M1, N, F2, FM, F).
```

dove la relazione ausiliaria **fib(M, N, F1, F2, F)** si può leggere come segue: "**F** è il numero di Fibonacci di **N** se **F2** lo è di **M** e **F1** di **M - 1**". Con questa procedura, che ha una ricorsione singola, la metà:

?- fibonacci_1(5, X).

genera la seguente successione di chiamate di **fib**:

fib(1, 5, 0, 1, X)

fib(2, 5, 1, 1, X)

fib(3, 5, 1, 2, X)

fib(4, 5, 2, 3, X)

fib(5, 5, 3, 5, X)

con risposta **X = 5**, ed il numero di passi di computazione cresce linearmente al crescere del primo argomento della metà. Qui il miglioramento di efficienza è molto maggiore rispetto al caso del fattoriale, ed è dovuto ad un'effettiva eliminazione di passi di computazione ridondanti nel programma, mentre là era dovuto al modo in cui è gestita la ricorsione durante l'esecuzione da parte dell'interprete.

Le due versioni della procedura fibonacci sono un esempio vistoso di procedure dichiarativamente equivalenti, nel senso che calcolano le stesse coppie della relazione, ma molto diverse proceduralmente.

Un esempio di procedure su liste nelle due versioni, con esecuzione dall'alto verso il basso o viceversa, è il seguente. La relazione **inversione(L1, L2)** afferma che la lista **L2** contiene gli stessi elementi della lista **L1**, ma disposti in ordine inverso. Una realizzazione con esecuzione dall'alto verso il basso, che fa uso della relazione [concatenazione](#), è la seguente:

```

inversione([], []). /* l'inversa della lista vuota è la lista vuota */
inversione([T|C], L2):- inversione(C, L1), concatenazione(L1, [T], L2). /* l'inversa di una lista non vuota è uguale
all'inversa della coda concatenata con la lista il cui unico elemento è la testa.*/
concatenazione([], L, L).
concatenazione([T|L1], L2, [T|L3]):-concatenazione(L1, L2, L3).

```

Una realizzazione con esecuzione simulata dal basso verso l'alto, che fa uso di una relazione ausiliaria, è:

```

inversione_1(L1,L2):-inv(L1, [],L2).
inv([],L, L).
inv([T|C], L1, L2):-inv(C, [T|L1], L2).

```

dove **inv(L1, L2, L3)** ha il significato: la lista **L3** è la concatenazione dell'inversa della lista **L1** e della lista **L2**. Anche in questo caso, inversione è dichiarativamente più chiara, mentre **inversione_1** è proceduralmente più efficiente.

Un esempio di più condizioni limite per procedure ricorsive su liste è il seguente: **ordinata(L)** indica che la lista di numeri **L** è ordinata in senso crescente. La sua definizione è:

```

ordinata([]).
ordinata([_]).
ordinata([X,Y|Z]):-X=<Y, ordinata([Y|Z]).

```

Le tre clausole corrispondono rispettivamente ad una lista vuota, ad una lista con un solo elemento e ad una lista con più di un elemento.

Procedure con più clausole ricorsive.

Un esempio di procedura in cui è utile avere più regole ricorsive è quello della funzione di esponenziazione intera **X^Y=Z**, definita da:

1. **X** e **Y** rappresentano numeri interi;
2. **X⁰=1**;
3. **X^Y=X * (X^(Y-1))**.

Le corrispondenti clausole Prolog sono:

```

esp(_, 0, 1).
esp(X, Y, Z):-Y1 is Y - 1, esp(X, Y1, Z1), Z is Z1 * X.

```

D'altra parte, la funzione ha anche la proprietà:

$$X^{(2+Y)} - (X^2)^Y,$$

equivalente a:

$$X^Y = (X * X)^{Y/2}$$

poiché $2 * Y/2 = Y$. Questa proprietà può essere utilizzata ogni volta che $Y/2$ è intero, ossia ogni volta che Y è pari, per dimezzare il numero di operazioni (ricorsioni) da effettuare. La clausola corrispondente è:

```
esp(X, Y, Z):- pari(Y), Y1 in Y//2, X1 is X * esp(X1, Y1, Z).
```

Essa non modifica il contenuto dichiarativo della relazione in quanto esprime una proposizione vera per essa, ed anzi - aggiunta alle clausole precedenti in posizione opportuna - rende la procedura più efficiente. Definendo anche la relazione **pari(Y)**, il programma completo è:

```

esp_1(_, 0, 1).
esp_1(X, Y, Z):-pari(Y), Y1 is Y//2, X1 is X * X, esp_1(X1, Y1, Z).
esp_1(X, Y, Z):-Y1 is Y - 1, esp_1(X, Y1, Z1), Z is Z1 * X.
pari(Y):-R is Y mod 2, R=:=0.

```

Ricorsione in coda.

Una forma particolare di definizione ricorsiva di una relazione, detta ricorsiva in coda (tail recursive), è tale che la corrispondente procedura può operare in modo efficiente. Chiamiamo ricorsiva a destra (a sinistra) una clausola nella quale il predicato della testa compare come il più a destra (a sinistra) nella congiunzione di predici del corpo. Una procedura è ricorsiva in coda quando contiene una sola clausola ricorsiva, questa compare come ultima clausola della procedura, ed è ricorsiva a destra. Ad esempio, la procedura ordinato è ricorsiva in coda, mentre le procedure esp ed esp_1 non lo sono.

Si è detto, nell'[Efficienza e terminazione nelle definizioni ricorsive](#), della minore efficienza che si può avere ponendo le condizioni limite prima delle clausole ricorsive. D'altra parte tale disposizione è una condizione necessaria affinché la procedura sia ricorsiva in coda e possa quindi godere dei benefici di seguito indicati.

In una procedura ricorsiva in coda, se nella clausola ricorsiva tutte le condizioni che precedono quella ricorsiva sono tali da ammettere un'unica soluzione (oppure mancano), allora la sua esecuzione risulta particolarmente efficiente per quei sistemi che implementano la ottimizzazione della ricorsione in coda (tail recursion optimization), per la quale si rimanda alla scheda. Ad esempio, la maggiore efficienza della procedura **inversione_1** rispetto ad **inversione**, e di

fattoriale_1 rispetto a **fattoriale**, è correlata al fatto di essere ricorsive in coda, soddisfacendo la suddetta condizione.

Ottimizzazione della ricorsione in coda.

Si è visto nell'[Implementazione della strategia Prolog](#) che l'interprete Prolog utilizza una pila in cui memorizzare un record per ogni procedura attiva, contenente i valori delle variabili locali ed altre informazioni. A differenza dei linguaggi tradizionali, questo record non può essere annullato quando la procedura ha ottenuto il suo risultato, perché possono esserci ritorni indietro, e deve quindi essere mantenuto sino a quando la procedura ha ottenuto tutti i suoi risultati.

Quando il sistema può rilevare che ha raggiunto l'ultima metà di una clausola, e che non vi sono punti di ritorno indietro rinianenti nella procedura a cui quella clausola appartiene, allora può riutilizzare il record di attivazione, se è fornito di una opportuna gestione della memoria non più utile (garbage collection). In questo modo una procedura ricorsiva in coda che produce un solo risultato non usa mai più di un record della pila.

Questo tipo di ottimizzazione è per altro più generale, perché si applica non solo alle procedure ricorsive in coda, ma a tutte le procedure per le quali sia noto che non possono fornire più di un risultato, cioè per le quali il sistema possa rilevare che quella in corso è la loro ultima chiamata. Il **Prolog/DEC-10** è il primo sistema nel quale è stata implementata questa caratteristica.

Una ulteriore realizzazione della procedura per il calcolo del fattoriale, che utilizza la tecnica della ricorsione a destra e fa uso di una procedura ausiliaria dotata di una definizione ricorsiva in coda, è la seguente. La relazione **fattor(N, X, F)** vale se **F** è il prodotto di **X** e del fattoriale di **N**; **fattoriale_2** viene poi definito in termini di **fattor**:

```
fattoriale_2(N, F):-fattor(N, 1, F).
fattor(0, F, F).
fattor(N, X, F):-N1 is N - 1, Y is N * X, fattor(N1, Y, F).
```

In generale, la combinazione di ricorsione a sinistra con la regola di computazione di Prolog che seleziona il predicato più a sinistra di una congiunzione risulta in un albero di ricerca che ha un ramo infinito, mentre con la ricorsione a destra l'albero di ricerca è finito.

Perciò, con la regola di ricerca in profondità di Prolog, nel primo caso la possibilità di trovare una soluzione (od entrare in un ciclo infinito) dipende dall'ordinamento delle clausole, mentre nel secondo caso questo è ininfluente. D'altra parte l'ordine delle mete in una congiunzione e l'ordine delle clausole influiscono, come si è visto, sull'efficienza della computazione (si riveda, al riguardo, l'esempio della relazione [stessa lunghezza](#)).

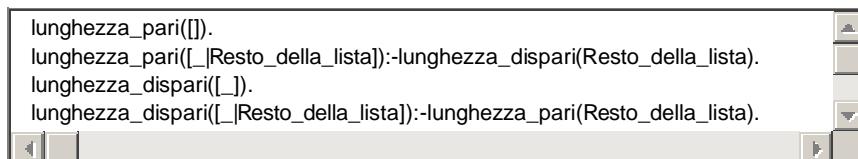
Ricorsione mutua.

La forma di ricorsione considerata fino a questo punto è quella della ricorsione diretta, nella quale cioè il predicato che costituisce la testa di una clausola compare anche (una o più volte) nel corpo della stessa clausola.

Un'altra forma di ricorsione è quella della ricorsione indiretta o ricorsione mutua, nella quale il predicato che costituisce la testa di una clausola compare anche nel corpo di un'altra clausola.

Un semplice esempio è dato dalle seguenti procedure, **lunghezza_pari** e **lunghezza_dispari**, le cui invocazioni terminano con successo se i loro argomenti sono rispettivamente liste di lunghezza pari o dispari:

```
lunghezza_pari([]).
lunghezza_pari([_|Resto_della_lista]):-lunghezza_dispari(Resto_della_lista).
lunghezza_dispari([]).
lunghezza_dispari([_|Resto_della_lista]):-lunghezza_pari(Resto_della_lista).
```



Naturalmente è necessario porre attenzione ad evitare definizioni circolari, come la seguente:

```
genitore(X, Y):-figlio_a(Y, X).
figlio_a(X, Y):-genitore(Y, X).
```



Con tale definizione, un qualunque quesito riguardante un genitore od un **figlio_a** porta ad un ciclo senza fine.

Si può osservare che la lettura dichiarativa di una tale definizione può essere espressa come: "**X** è genitore di **Y** se e solo se **Y** è figlio_a di **X**" (o viceversa), cioè corrisponde ad impiegare l'operatore logico di doppia implicazione (o bicondizionale), denotato in logica da " \Leftrightarrow " ("se e solo se"), che non è però utilizzabile in una clausola di Horn. Le definizioni bicondizionali non sono perciò esprimibili con clausole di Horn, e questo costituisce una limitazione del linguaggio.

In generale, possono portare a definizioni circolari quelle ricorsioni indirette che coinvolgono una serie arbitrariamente lunga di regole, del tipo:

regola_1(...):-regola_2(...).

regola_2(...):-regola_3(...).

...

regola_n(...):-regola_1(...).

nelle quali la circolarità può non essere facilmente riconoscibile.

Possibili cicli infiniti nella ricorsione.

Una clausola del tipo:

p(X):-p(X).

contenente lo stesso predicato sia come condizione che come conclusione, è detta una tautologia.

Innocua da un punto di vista dichiarativo, una tautologia conduce proceduralmente ad un ciclo infinito; pertanto va considerata un errore di programmazione in un programma sottoposto ad un interprete Prolog standard. Può essere eliminata dall'insieme delle clausole senza influenzare il significato dichiarativo del programma.

Una clausola che esprime la simmetria di una relazione, come:

p(X, Y) :- p(Y, X).

può dar luogo ad un ciclo infinito. Per esempio, con il programma:

```
ama(giuseppe, maria).  
ama(X, Y) :- ama(Y, X).  

```

il quesito:

?- ama(maria, giuseppe).

ottiene la risposta corretta sì, mentre il quesito:

?- ama(maria, maria).

dà luogo ad un ciclo infinito.

Una possibile soluzione consiste nel definire una relazione simmetrica mediante due regole aventi come condizioni le due parti asimmetriche di una relazione ausiliaria:

p(X, Y) :- p_1(X, Y).

p(X, Y) :- p_1(Y, X).

Un'altra possibilità di cicli infiniti si ha con una clausola che esprime la transitività di una relazione, del tipo:

p(X, Y) :- p(X, Z), p(Z, Y).

Consideriamo ad esempio il programma:

```
antenato(a, b).  
antenato(b, c).  
antenato(c, d).  
antenato(X, Y):-antenato(X, Z), antenato(Z, Y).  

```

Il quesito:

?- antenato(b, d).

ottiene correttamente la risposta si, mentre il quesito che chiede tutte le coppie della relazione:

?- antenato(X, Y).

genera mediante ritorno indietro le risposte:

X=a Y=b;

X=b Y=c;

X=c Y=d;

X=a Y=c;

X=a Y=d;

dopodiché dà luogo ad un ciclo infinito, senza essere più in grado di generare l'ulteriore coppia:

X=b Y=d.

L'albero di dimostrazione è il seguente:

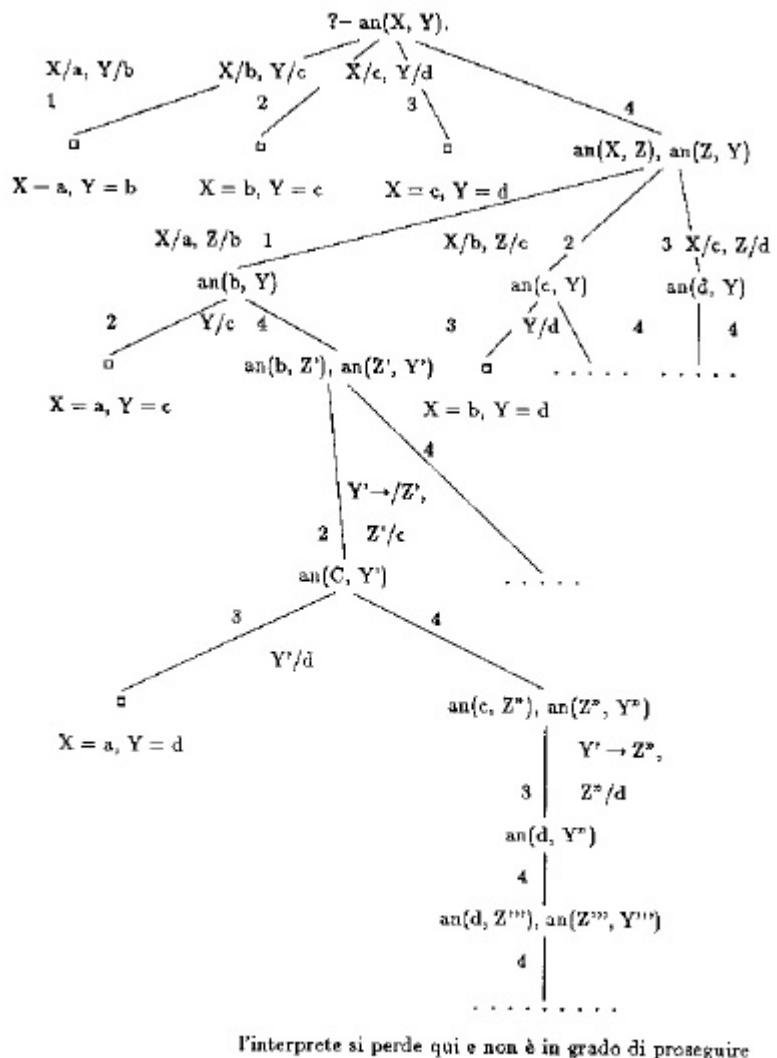


Figura 5.5: Albero relativo alla procedura antenato.

Una possibile soluzione consiste nell'introdurre una relazione aggiuntiva che costituisca un caso particolare della relazione transitiva. Si noti però che non è sempre possibile riformulare le relazioni nei modi sopra indicati. D'altra parte, diversi tentativi di trovare condizioni sulla base delle quali rilevare in modo automatico la possibilità di cicli infiniti non hanno finora condotto ad una soluzione soddisfacente e conclusiva.

Osservazioni sulla ricorsione.

La ricorsione, che è un costrutto fondamentale in Prolog, è anche il costrutto che richiede la maggiore attenzione. Dal punto di vista dichiarativo, il compito più importante è individuare i casi di base e su quale argomento del predicato (se di molteplicità maggiore di 1) effettuare la ricorsione (allo stesso modo in cui, in una dimostrazione induttiva, la cosa più importante è individuare la giusta ipotesi induttiva). Dal punto di vista procedurale, è importante individuare il migliore ordinamento delle sottomete e delle clausole, tenendo conto allo stesso tempo dei problemi di terminazione e di efficienza.

Le procedure di tipo ricorsivo forniscono spesso descrizioni sintetiche ed eleganti delle relazioni dà esse definite, ed ammettono in generale un'interpretazione dichiarativa semplice e trasparente.

Possono però presentare lo svantaggio di richiedere una quantità notevole di memoria, a volte in misura inaccettabile, ed anche di condurre a cicli senza fine. L'inefficienza si ha soprattutto nel caso di clausole con ricorsioni multiple e di clausole nelle quali la meta ricorsiva non è l'ultima.

In alcuni casi un livello accettabile di efficienza, oppure la terminazione, possono essere raggiunti solo trasformando la logica del programma, in modi che possono renderla anche più complessa e meno chiara.

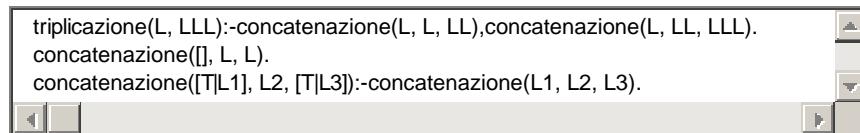
Sequenza.

Lo schema di esecuzione in ordine sequenziale di più operazioni trova in Prolog un'immediata corrispondenza nello scriverle come sottomete nel corpo di una clausola, nell'ordine desiderato, ad esempio, la tipica sequenza tra un'operazione di lettura di dati d'ingresso **X**, l'elaborazione di **X** per produrre dati d'uscita **Y**, e la scrittura di **Y**, si esprime semplicemente come:

procedura(X, Y) :- lettura(X), elaborazione(X1 Y), scrittura(Y).

La regola di computazione standard dell'interprete Prolog ed il meccanismo di chiamata di procedura mediante unificazione fanno sì che venga innanzitutto invocata la procedura **lettura(X)** che, acquisendo i dati d'ingresso, istanzia **X**. Il valore di **X** viene poi passato in ingresso ad **elaborazione(X, Y)**, che è chiamata per seconda ed istanzia **Y** in uscita. Il valore di **Y** viene quindi distribuito a **scrittura(Y)**, chiamata per ultima.

È però da notare subito che, nel corpo delle clausole che non contengono predicati con effetti collaterali, la sequenziazione delle sottomete per la progressiva formazione del risultato finale non necessita di un ordine preciso, se non per motivi di efficienza. Consideriamo per esempio la relazione **triplicazione(L, LLL)**, valida se la lista **LLL** è costituita da tre copie consecutive della lista **L**. Si può definirla nel modo seguente (utilizzando la relazione [concatenazione](#)):



```
triplicazione(L, LLL):-concatenazione(L, L, LL),concatenazione(L, LL, LLL).
concatenazione([], L, L).
concatenazione([T|L1], L2, [T|L3]):-concatenazione(L1, L2, L3).
```

Dichiarativamente ciò equivale ad affermare che la triplicazione della lista **L** è uguale alla concatenazione di **L** con la duplicazione di **L**.

Attivando questa clausola con **L** istanziata, l'esecuzione della prima sottometa istanzia **LL** e trasmette l'istanza trovata alla seconda sottometa, la cui esecuzione istanzia poi **LLL**. In entrambe le sottomete, quindi, la procedura concatenazione viene invocata con i primi due argomenti istanziati. Lo stesso risultato si può ottenere anche definendo la procedura con un diverso ordine delle sottomete:

```

triplicazione_1(L, LLL):-concatenazione(L, LL, LLL), concatenazione(L, L, LL).
concatenazione([], L, L).
concatenazione([T|L1], L2, [T|L3]):-concatenazione(L1, L2, L3).

```

Il contenuto dichiarativo non è alterato da questo diverso ordinamento delle sottomete.

Proceduralmente, invece, **triplicazione_1** opera come segue: la prima sottometa viene ora attivata con il solo primo argomento **L** istanziato, e la sua esecuzione lega **LLL** alla lista **[L|LL]**, cioè istanzia parzialmente **LLL**, in quanto **L** è istanziato mentre **LL** non lo è. La seconda sottometa, attivata con i primi due argomenti (uguali) istanziati, istanzia **LL**, che completa così l'istanziamento di **LLL**.

Sebbene di non grande interesse applicativo, l'esempio mostra con chiarezza come il comportamento tipico della variabile logica, ossia la caratteristica di poter essere istanziata progressivamente, rende ininfluente rispetto al risultato l'ordinamento relativo delle sottomete nel corpo di una clausola. Tale ordinamento non ha dunque alcun significato dal punto di vista dichiarativo, ma può avere una anche notevole influenza sotto il profilo procedurale, cosicché risulta importante nell'attività pratica di programmazione. Consideriamo per esempio la relazione **ordinamento_ingenuo(L, L0)**, che esprime il fatto che la lista **L0** è una versione ordinata della lista **L**:

ordinamento_ingenuo(L, L0):-permutazione(L, L0), ordinata(L0).

La lettura dichiarativa è unica, indipendentemente dall'ordine di selezione: "**L0** è una versione ordinata di **L** se **L0** è una permutazione di **L** ed **L0** è ordinata". Le possibili letture procedurali sono invece due, a seconda di quale sottometa sia selezionata per prima:

1. per ordinare una lista **L**, va generata dapprima una permutazione **L0** di **L**, poi si verifica se **L0** è ordinata; se la verifica è positiva, **L0** è una versione ordinata di **L**;
2. per ordinare una lista **L**, va dapprima generata una lista ordinata **L0**, poi si accerta se **L0** è una permutazione di **L** se la verifica è positiva, **L0** è una versione ordinata di **L**.

È facile rendersi conto che la differenza in efficienza fra le due versioni può essere molto notevole. Per completare il programma, si aggiunga per la procedura [ordinata](#) e per permutazione la seguente definizione:

permutazione([], []).

permutazione(L, [T|C]):-permutazione(L1, C), cancellazione(T, L, L1).

cancellazione(_, [], []).

cancellazione(E, [E|C], C).

cancellazione(E, [T|C1], [T|C2]):-E\== T,cancellazione(E, C1, C2).

dove **cancellazione(E, L1, L2)** significa: "la lista **L2** è uguale alla lista **L1** privata della prima occorrenza dell'elemento **E**". È da notare che cancellazione restituisce immutata in uscita la lista assegnata in ingresso se l'elemento da cancellare non è presente in essa (caso particolare è quello in cui la lista è vuota). Volendo invece che la procedura in questo caso fallisca (in modo più corrispondente alla lettura dichiarativa) è sufficiente eliminare la prima condizione limite.

Il programma che si ottiene è il seguente:

```
ordinamento_ingenuo(L, L0):-permutazione(L, L0), ordinata(L0).
ordinata([]).
ordinata([_]).
ordinata([X,Y|Z]):-X=<Y, ordinata([Y|Z]).
permutazione([], []).
permutazione([L, [T|C]):-permutazione(L1, C), cancellazione(T, L, L1).
cancellazione(_, [], []).
cancellazione(E, [E|C], C).
cancellazione(E, [T|C1], [T|C2]) :- E== T, cancellazione(E, C1, C2).
```



L'ordine tra le sottomete nel corpo di una clausola può influire sull'efficienza della computazione a seconda dell'uso che se ne fa, vale a dire a seconda di come viene chiamata. Consideriamo il programma:

```
nonno_a(X, Y):-genitore(X, Z), genitore(Z, Y).
genitore(maria, giorgio).
genitore(paolo, giorgio).
genitore(piетro, paolo).
```



Se la chiamata è:

? nonno_a(piетro, Y). /* di chi è nonno_a pietro? */

la regola di computazione di Prolog è efficiente, perché il primo argomento della prima sottometa viene subito istanziato, cosicché una sola tra le clausole genitore corrisponde, istanziando il secondo argomento; lo stesso avviene per la seconda sottometa, fornendo la risposta: **Y = giorgio** (pietro è nonno di giorgio). Se invece la chiamata è:

?- nonno_a(Y, giorgio). /* chi è nonno_a di giorgio? */

la regola di computazione da sinistra a destra non è efficiente, perché entrambi gli argomenti della prima sottometa sono ora non istanziati, quindi tutte le clausole genitore devono essere provate; l'unificazione della prima sottometa con ognuna delle prime due clausole comporta un fallimento della seconda sottometa che provoca un ritorno indietro, e solo dopo l'unificazione della prima sottometa con la terza clausola la seconda sottometa riesce, dando la risposta **Y = pietro** (il nonno di giorgio è pietro). Per questo secondo tipo di utilizzo è più efficiente riordinare le sottomete nel corpo della clausola **nonno_a**:

nonno_a(X,Y):-genitore(Z, Y),genitore(X, Z).

ottenendo il programma:

```

nonno_a(X,Y) :- genitore(Z, Y), genitore(X, Z).
genitore(maria, giorgio).
genitore(paolo, giorgio).
genitore(pietro, paolo).

```

Se la chiamata è:

?- nonno_a(piетro, giorgio). /* è vero che pietro è nonno di giorgio? */

allora entrambe le versioni sono ugualmente efficienti.

Si noti che, non essendo presenti clausole ricorsive, non si pone un problema di terminazione (la soluzione viene trovata qualunque sia l'ordine adottato) ma solo di efficienza, legata al numero di alternative che devono essere considerate per trovare la soluzione.

In generale, per stabilire l'ordinamento più efficiente in una congiunzione di mete tra loro dipendenti, cioè aventi variabili in comune occorre considerare qual'è l'uso previsto per quella metà, e corrispondentemente quali argomenti ci si aspetta che siano istanziati al momento della sua chiamata. Usualmente (ma non necessariamente), l'ordinamento che minimizza il numero di alternative è quello per cui si trovano più a sinistra le sottomete che risultano avere il maggior numero di variabili istanziate al momento della chiamata, così da avere il minor numero di clausole che corrispondono.

Quest'ultimo criterio - mettere per prime le mete che hanno un numero minore di clausole corrispondenti nella base di dati – è più generale e si può applicare a parità di numero di variabili delle mete nella congiunzione. Definiti ad esempio i seguenti fatti:

```

greco(eschilo).
greco(platone).
filosofo(platone).

```

il quesito:

?- greco(X), filosofo(X).

comporta un ritorno indietro, mentre il quesito:

?- filosofo(X), greco(X).

riesce direttamente.

È infine da osservare che l'ordine dei predicati nel corpo di una clausola risulta invece essenziale per quei predicati predefiniti che presentano effetti collaterali non reversibili, tipicamente i predicati

di valutazione come **is**. Si consideri per esempio il seguente programma per il calcolo delle radici di un'equazione di secondo grado:

radici(A, B, C, R):-discriminante(A, B, C, Delta),radici_1(A, B, Delta, R).

La meta **discriminante(A, B, C, Delta)** è soddisfatta se **Delta** è il discriminante dell'equazione quadratica i cui coefficienti istanziano le variabili **A**, **B** e **C**:

discriminante(A, B, C, Delta):-BQ is B * B, P is 4*A*C, Delta is BQ-P.

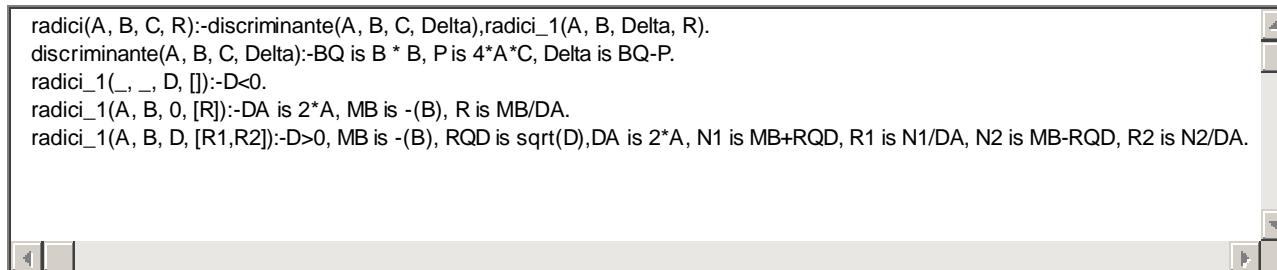
La meta **radici_1(A, B, Delta, L)** è soddisfatta se **L** è la lista delle radici dell'equazione quadratica i cui primi due coefficienti ed il cui discriminante istanziano rispettivamente le variabili **A**, **B** e **Delta**:

radici_1(_, _, D, []):-D<0.

radici_1(A, B, 0, [R]):-DA is 2*A, MB is -(B), R is MB/DA.

radici_1(A, B, D, [R1,R2]):-D>0, MB is -(B), RQD is sqrt(D), DA is 2*A, N1 is MB+RQD, R1 is N1/DA, N2 is MB-RQD, R2 is N2/DA.

Con tali definizioni si ottiene il programma:



```
radici(A, B, C, R):-discriminante(A, B, C, Delta),radici_1(A, B, Delta, R).
discriminante(A, B, C, Delta):-BQ is B * B, P is 4*A*C, Delta is BQ-P.
radici_1(_, _, D, []):-D<0.
radici_1(A, B, 0, [R]):-DA is 2*A, MB is -(B), R is MB/DA.
radici_1(A, B, D, [R1,R2]):-D>0, MB is -(B), RQD is sqrt(D), DA is 2*A, N1 is MB+RQD, R1 is N1/DA, N2 is MB-RQD, R2 is N2/DA.
```

Selezione.

In Prolog l'esigenza - o l'opportunità - di procedere alla rappresentazione di un (sotto)problema, e dunque all'espressione di una meta, in termini di una trattazione per casi, trovano una naturale corrispondenza nella definizione di una procedura mediante più clausole; questa caratteristica generale può poi articolarsi in diverse possibilità.

La prima, e più semplice, è quella di discriminare i diversi casi solo sulla base dei parametri della procedura. Il meccanismo di chiamata di procedura mediante unificazione illustrato nel capitolo 3, in base al quale, assegnata una meta:

?-p(f1,..., fm).

viene verificata la sua corrispondenza strutturale con la testa di ogni clausola i-esima fra le n che la definiscono:

pi(ti1,...,tim).

è sufficiente ad effettuare automaticamente la selezione. Questa può avvenire simultaneamente su un qualunque numero **k** (**1 =< k =< m**) di parametri in ingresso, ed operare in modo semplice o più sofisticato a seconda che i termini siano atomi o strutture.

Si consideri, come esempio, la seguente procedura (parzialmente specificata) per il calcolo di derivate, nella quale sono definite - mediante clausole unitarie - un certo numero di derivate di base, o "pre-calcolate", seguite da una serie di regole generali:

derivata(X, X, 1).

derivata(sin(X), X, cos(X)).

derivata(cos(X), X, -sin(X)).

derivata(log(X), X, 1/X).

...

derivata(...).-...

...

Quando la procedura viene chiamata con il primo argomento corrispondente al primo argomento di una delle clausole unitarie, la ricerca si arresta all'altezza della clausola per la quale la corrispondenza ha avuto luogo, ed il terzo argomento di tale clausola è il risultato richiesto. In caso contrario la ricerca continuerà fino a determinare l'utilizzo di una delle regole, che si incaricherà del calcolo effettivo.

La selezione guidata dalla corrispondenza strutturale può non essere sufficiente per lo scopo: spesso si rendono necessarie ulteriori condizioni, che vengono espresse come sottomete nel corpo delle clausole. Tali sottomete possono essere le prime da sinistra nel corpo della clausola, ed allora fungono da condizioni aggiuntive di applicazione della clausola, che fallisce se non sono soddisfatte. Per esempio:

p(0, Y):-a(Y).

p(X, Y):-X>0, b(Y).

p(X, Y):-X<0, c(Y).

Più in generale, può presentarsi la situazione in cui si abbiano differenti possibilità di continuazione dopo una o più sottomete di una congiunzione. Per esempio, schematicamente:

a :- b, c, d_1.

a :- b, c, d_2.

...

a :- b, c, d_n.

dove si suppone che, in dipendenza dai parametri in uscita dalla procedura e, sono da invocare le procedure **d_1** oppure **d_2** oppure ... **d_n**.

Questa impostazione è tanto meno accettabile quanto più è complessa l'elaborazione attivata dalla sottometa **b**, in quanto questa verrebbe, nel caso generale, effettuata più volte. Occorre perciò individuare metodi alternativi.

Un primo metodo consiste nell'introdurre un punto di scelta che, in riferimento allo schema precedente, può essere così realizzato:

a :- b, punto_di_scelta.

punto_di_scelta:-condizione_1, c, d_1.

punto_di_scelta:-condizione_2, c, d_2.

punto_di_scelta:-condizione_n, c, d_n.

Con questa soluzione la meta a viene frammentata in più parti.

In ogni clausola i-esima di punto_di_scelta, il soddisfacimento della sottometa condizione_i determina l'attivazione di e con parametri in ingresso diversi da quelli che verrebbero ottenuti con le altre clausole.

Si può realizzare una soluzione alternativa definendo la seguente procedura ad una sola clausola:

a:-b, c(S), d_1(S), d_2(S), ..., d_n(S).

con c tale da determinare l'istanziamento della variabile S (che funge così da selettore) ad uno fra n possibili valori predefiniti (per esempio **1, 2, ..., n**), ciascuno corrispondente all'attivazione di una delle procedure **d_1, d_2, ..., d_n**, le quali verranno così definite:

d_1(1):- ... /* codice 1 */

d_1(2):- ... /* codice 2 */

...

d_1(n):- ... /* codice n */

d_2(1):- ... /* codice 1 */

d_2(2):- ... /* codice 2 */

...

d_2(n):- ... /* codice n */

...

```
d_n(1):- ... /* codice 1 */
```

```
d_n(2):- ... /* codice 2 */
```

...

```
d_n(n):- ... /* codice n */
```

Questa situazione evita la frammentazione della procedura a introdotta dalla soluzione precedente, ma introduce nel contempo l'utilizzo artificioso della variabile S, che è estranea alla logica del problema in quanto funge da parametro fittizio d'ingresso per le procedure entro le quali viene utilizzata. Procedimenti di questo tipo vengono qualche volta indicati con il termine di fattorizzazioni.

Le modalità di selezione descritte in questo paragrafo possono essere rese più efficaci mediante un accurato uso del predicato predefinito per il controllo del ritorno indietro, che esamineremo nel prossimo capitolo.

Iterazione.

L'iterazione dei linguaggi convenzionali trova in Prolog diverse corrispondenze ed interpretazioni, basate in parte sulle sue caratteristiche fondamentali quali la ricorsione ed il ritorno indietro, ed in parte su predicati di sistema definiti per questo scopo.

Iterazione per ritorno indietro.

La più semplice forma di iterazione è fornita dallo stesso meccanismo di ritorno indietro, che esamina tutte le alternative nello spazio di ricerca. Un semplice esempio è quello presentato nel seguito.

Assegnato un insieme di punti su un piano, espressi con coordinate X ed Y, per esempio:

```
p(0, 0).
p(0, 10).
p(0, 20).
p(10, 0).
p(10, 10).
p(10, 20).
p(20, 0).
p(20, 10).
p(20, 20).
```

si ottengono tutti i punti che soddisfano una certa relazione geometrica, definita da un opportuno quesito, mediante ricerca sequenziale della base di dati con possibili ritorni indietro. Per esempio, per ottenere tutti i punti di ascissa **10**, il quesito:

```
?- p(10, Y).
```

dà luogo ad un'iterazione semplice, ossia ad una singola scansione della base di dati. Per ottenere tutti i segmenti verticali (esclusi quelli di lunghezza nulla e quelli speculari) di ascissa **10** (vale a dire tutte le coppie di ordinate dei punti estremi), il quesito:

?- **p(10, Y1), p(10, Y2), Y1 < Y2.**

dà luogo ad una doppia iterazione, una più esterna in cui varia **Y1** ed un'altra, interna alla precedente, nella quale, per ogni **Y1**, varia **Y2**.

Per ottenere tutti i segmenti verticali (esclusi quelli di lunghezza nulla e quelli speculari), il quesito:

?- **p(X, Y1), p(X, Y2), Y1 < Y2.**

dà luogo a tre cicli contenuti uno nell'altro, in cui variano **Y2**, **Y1** ed **X**, rispettivamente (dall'interno verso l'esterno).

Si noti che ogni ciclo interno dà luogo ad un ulteriore livello di ramificazione nell'albero di dimostrazione; per esempio il secondo quesito dà luogo all'albero della seguente figura (dove il ciclo esterno risulta nel primo livello di ramificazione, e quello interno nel livello successivo):

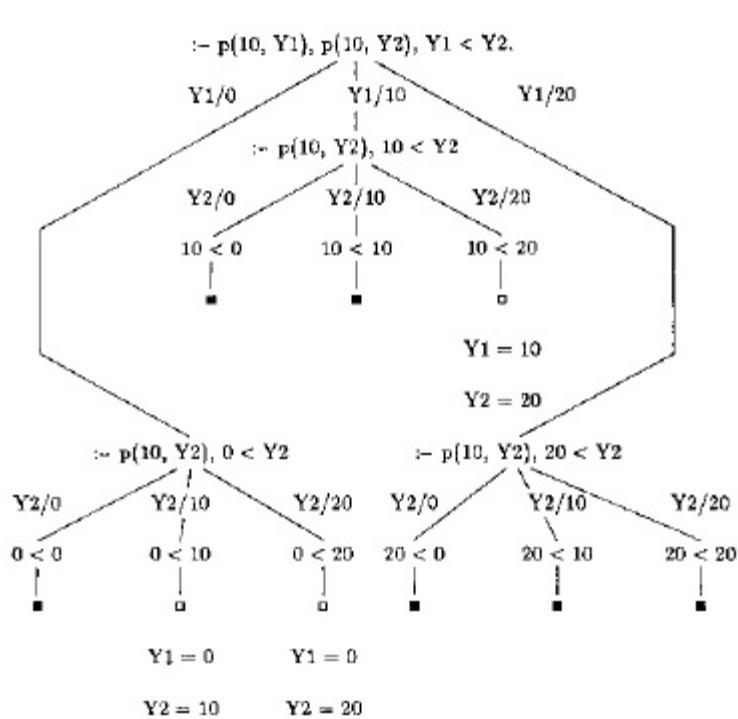


Figura 5.6.

Ponendo i quesiti precedenti, si possono ottenere tutte le soluzioni attivando da tastiera il ritorno indietro dopo ogni soluzione fornita in risposta dall'interprete. Per ottenere la ripetizione automatica, con la quale vengono fornite progressivamente tutte le soluzioni senza interazione da parte dell'utente, si può usare il predicato predefinito fail. È una procedura senza argomenti il cui unico effetto è quello di influenzare il flusso di controllo; infatti essa fallisce sempre, forzando il ritorno indietro. Nell'esempio precedente, il quesito:

?- **p(10, Y), write('Y ='), write(Y), nl, fail.**

visualizza, senza ulteriori interventi da parte dell'utente, tutti i punti di ascissa **10** definiti come fatti nella base di dati.

Iterazione come ricorsione in coda.

Una diversa interpretazione dell'iterazione è quella in cui essa è considerata come quel caso particolare di ricorsione, detto ricorsione in coda, o ricorsione eseguita dal basso verso l'alto. La ricorsione in coda può essere espressa nella forma:

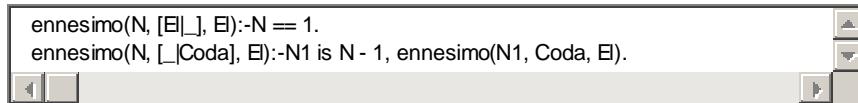
p(X) :- r(X).

p(X) :- q(X, X1), p(X1).

In questa procedura **X** è un parametro d'ingresso che viene modificato in **X1** da **q(X, X1)** e, ad ogni passo ricorsivo attivato da **p(X1)**, viene sottoposto alla condizione **r(X)**, il cui verificarsi causa la terminazione. Tale tipo di ricorsione può essere perciò interpretato come l'iterazione semplice:

UNTIL r REPEAT q

dei linguaggi di programmazione tradizionali, quale il Pascal. Come esempio consideriamo la seguente procedura che, assegnati in ingresso una lista ed un intero positivo **N** non superiore alla sua lunghezza, dà in uscita l'**N**-esimo elemento della lista:

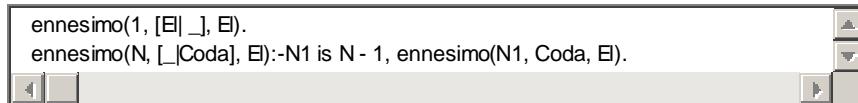
```
ennesimo(N, [El|_], El):-N == 1.  
ennesimo(N, [_|Coda], El):-N1 is N - 1, ennesimo(N1, Coda, El).  

```

È facile riconoscere il processo con cui la lista iniziale viene ripetutamente privata del suo primo elemento, ogni volta decrementando il valore di **N**, fino a quando **N** è uguale ad 1, cioè fino al momento in cui, avendo eliminato i primi **N** - 1 elementi, l'elemento **N**-esimo viene a trovarsi in testa alla lista rimanente, da cui viene prelevato come risultato.

Si noti che, facendo appello al meccanismo di unificazione, la prima clausola può essere espressa più semplicemente come fatto:

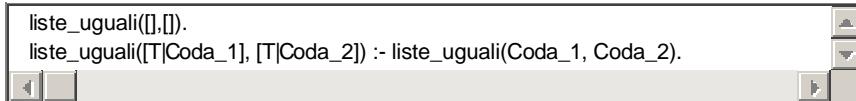
ennesimo(1, [El|_], El).

ottenendo il programma:

```
ennesimo(1, [El|_], El).  
ennesimo(N, [_|Coda], El):-N1 is N - 1, ennesimo(N1, Coda, El).  

```

Più in generale, è spesso possibile esprimere la condizione di terminazione **r** non nel corpo della prima clausola, ma unicamente come caso particolare dell'argomento della testa.

A questo riguardo si può osservare che è sempre opportuno cercare di delegare la maggior quantità possibile di "lavoro" al meccanismo predefinito di unificazione del sistema Prolog. Per esempio, la verifica dell'uguaglianza di due liste può essere compiuta con:



```
liste_uguali([],[]).  
liste_uguali([T|Coda_1], [T|Coda_2]) :- liste_uguali(Coda_1, Coda_2).
```

ma ciò non è necessario; tutto il lavoro può essere compiuto da:



```
liste_uguali(L, L).
```

L'iterazione è di solito più efficiente della ricorsione perché il completamento di un passo iterativo - a differenza di quello di un passo ricorsivo - non richiede di attendere i risultati dei passi successivi. Caratteristica dell'iterazione è pertanto quella di usare una quantità costante di memoria, mentre la ricorsione usa una pila di memoria che cresce ad ogni passo ricorsivo. Nella ricorsione in coda, ogni nuova sottometa **p(X1)** può sostituire la precedente sottometa **p(X)**, e l'esecuzione richiede solo una quantità costante di memoria per la sottometa corrente.

Iterazione con il costrutto "repeat".

Poiché è più efficiente in generale utilizzare il ritorno indietro anziché la ricorsione, è disponibile in Prolog il predicato predefinito repeat, che riesce sempre, generando una ripetizione senza fine.

Esso può venire utilizzato, in combinazione con il predicato fail, per realizzare un ciclo iterativo.

Consideriamo, per esempio, l'iterazione dello schema sequenziale di lettura, elaborazione e scrittura; si può esprimere con la seguente procedura mono-clausola:

esecuzione:-lettura(X),elaborazione(X, Y),scrittura(Y),esecuzione.

che consente l'esecuzione ripetuta di sottomete. In questo caso non è in generale possibile l'ottimizzazione della ricorsione in coda, perché **elaborazione(X, Y)** potrà essere sottoponibile a ritorno indietro.

La chiamata ricorsiva finale può essere evitata con:

esecuzione :- repeat, lettura(X), elaborazione(X, Y), scrittura(Y), fail.

Nei casi in cui la definizione di elaborazione rappresenta un programma complesso, la differenza tra le due soluzioni può risultare in un programma eseguibile al posto di un programma che esaurisce la memoria prima di giungere alla conclusione.

Per concludere si può osservare che caso per caso va ricercato un opportuno bilanciamento fra l'uso della ricorsione, dichiarativamente più efficace ma in alcuni casi poco efficiente, e l'uso dell'iterazione, con i predicati di controllo predefiniti, più efficiente ma priva di significato logico.

Note bibliografiche.

Una discussione delle relazioni fra induzione e ricorsione, seppure nel contesto del Lisp, si trova in Allen (1978). Un'ampia presentazione delle caratteristiche delle computazioni dall'alto verso il basso e dal basso verso l'alto nell'ambito della programmazione logica si trova in Kowalski (1979a), ed anche in Hogger (1984).

Nei lavori di Covington (1985a e 1985b), Nute (1985) e Poole e Goebel (1985) compaiono analisi su alcune fonti di cicli infiniti in Prolog, con l'indicazione di possibili soluzioni.

Warren (1980) ha descritto l'implementazione dell'ottimizzazione della ricorsione in coda nel Prolog/DEC-10. Anche Bruynooghe (1982a) l'ha realizzata nel suo sistema Prolog.

Il programma di calcolo delle radici di un'equazione quadratica compare nell'articolo di McDermott (1980). Le procedure lunghezza_pari e lunghezza_dispari sono riprese da Bratko (1980).

Le procedure triplicazione sono state introdotte da Warren (1980) e poi riprese da Cohen (1985).

Sommario.

Lo studente ha ora acquisito gli elementi principali da considerare nel definire procedure ricorsive ed iterative, con riguardo sia agli aspetti dichiarativi che a quelli procedurali.

6. Controllo del ritorno indietro

Dove si descrive il costrutto extra-logico del taglio quale predicato predefinito di controllo che agisce come meccanismo di potatura dell'albero di ricerca. Si mostrano i principali usi di questo predicato predefinito all'interno di procedure di selezione e di ripetizione; si illustrano gli effetti negativi che usi poco accorti ditale costrutto possono sortire, ed infine si discutono alcune possibilità alternative. Si è visto in precedenza che l'esecuzione di un programma Prolog può essere considerata come (la costruzione e) l'attraversamento dell'albero di ricerca, utilizzando la strategia in profondità da sinistra a destra con ritorno indietro.

In aggiunta alle possibilità di ordinamento delle clausole di una procedura e di sequenziazione delle mete all'interno delle clausole, considerate nel capitolo precedente, il linguaggio Prolog rende disponibile un altro strumento per influire sul controllo: il predicato predefinito senza argomenti "!", detto taglio (cut).

Il predicato predefinito di taglio.

Il taglio è strettamente correlato al meccanismo di ritorno indietro, e consente di alterarne il normale funzionamento, con la conseguenza di escludere ("tagliare", "potare") alcuni rami dell'albero di ricerca della computazione in corso e quindi la possibilità di risoddisfare alcune mete, in precedenza già soddisfatte, in altro modo, ossia con altri istanziamenti.

In generale il taglio può quindi essere usato quando, trovata una soluzione per una certa meta, non si è interessati a considerare altre soluzioni per quella meta o per le sue sottomete, o per escludere dalla ricerca quelle clausole delle quali si sa che non possono portare a soluzioni. Quanto segue si applica indifferentemente sia ai ritorni indietro che avvengono durante l'esecuzione del programma per effetto delle modalità di controllo in esso contenute, sia a quelli provocati dall'utente nella sua interazione col sistema.

Il predicato "!" viene utilizzato come una normale meta nel corpo di una clausola; esso viene immediatamente soddisfatto non appena è incontrato nel corso dell'usuale processo di soddisfacimento delle mete, e non può essere risoddisfatto. Comporta tuttavia un effetto collaterale che modifica da quel punto in poi il normale funzionamento del ritorno indietro, effetto che consiste nel congelare, cioè fissare come definitive, tutte le scelte effettuate dopo la chiamata della meta genitrice (la meta che, unificata con la testa della clausola che contiene il taglio, l'ha attivata). Tutte le soluzioni alternative comprese tra la meta genitrice ed il taglio vengono quindi scartate, nel senso che ne viene impedita la ricerca che in sua assenza sarebbe possibile. Vengono invece mantenute le alternative presenti sui rami più a monte nell'albero di ricerca.

Se, in un momento successivo della computazione, un fallimento di una meta a destra del taglio o da questa richiamata determina un ritorno indietro fino ad esso, non essendo possibile il risoddisfacimento delle mete a sinistra del taglio, si ha l'immediato fallimento della meta che aveva determinato l'utilizzo della clausola che lo contiene. Poiché anche le clausole alternative ad essa non vengono più considerate, il ritorno indietro continua con il tentativo di risoddisfare la meta immediatamente alla sinistra della meta genitrice.

Il predicato di taglio ha quindi l'effetto di rendere deterministica l'esecuzione di un insieme di clausole che sono altrimenti non deterministiche; le mete rese deterministiche sono la meta genitrice, tutte le mete presenti prima del taglio nella clausola che lo contiene, e tutte le sottomete attivate durante l'esecuzione di quelle mete. In quanto orientato al controllo dell'esecuzione, il taglio

introduce un fattore extra-logico nella computazione. Tale fattore può risolversi tanto in un miglioramento di efficienza della computazione quanto in una perdita di corrispondenza di essa col significato dichiarativo delle clausole, a seconda di come viene usato. Essenziale a questo riguardo è considerare se tra i rami potati esistono o no soluzioni.

Nel seguito si mostreranno dapprima esempi schematici per illustrare il meccanismo più in dettaglio, e poi esempi concreti per mostrare i possibili vantaggi e svantaggi nell'utilizzo di questo predicato extra-logico.

Nel primo gruppo di esempi che seguono, si utilizzano per semplicità prediciati senza argomenti, ma tutte le considerazioni indicate valgono parimenti nel caso generale di prediciati con argomenti, cui si applica l'usuale nozione di unificazione. Per visualizzare l'effetto del taglio, si mostrerà l'albero AND-OR completo che si avrebbe per un certo insieme di clausole senza il taglio, e la parte che viene potata per la presenza di esso.

Come primo esempio consideriamo il caso di una meta consistente di un'unica clausola, che contiene un taglio dopo una sottometa a sua volta definita da due clausole alternative:

```
a :- b, !.
b :- c.
b :- d.

[<] [>] [?] [!]
```

Le quattro possibilità dipendenti dalla riuscita o meno di **c** e **d** sono rappresentate dai seguenti alberi, dove è evidenziata la parte tagliata per effetto della presenza del taglio nella prima clausola:

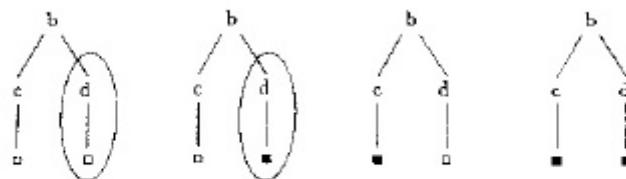


Figura 6.1.

Nel primo caso **b** riesce con la seconda clausola e, per effetto del taglio che elimina la possibilità di risoddisfare **b**, la terza clausola non può venire considerata. Una soluzione risulta pertanto esclusa. Nel secondo caso si ha la stessa esecuzione del primo, ma questa volta l'ignorare la terza clausola, che comunque fallirebbe, comporta lo stesso risultato con minore computazione. Nel terzo e quarto caso, in cui la seconda clausola fallisce, la terza clausola viene considerata con successo e fallimento di **b** (e quindi di **a**), rispettivamente. In questi casi nessuna parte viene tagliata, in quanto non vi sono altre clausole per **b**.

Questo primo esempio è anche indicativo di ciò che succede per effetto del taglio quando a sinistra di esso vi sono più sottomete: per ognuna di esse, la prima clausola che riesce esclude, se ve ne sono, le possibili alternative, non appena il taglio viene eseguito.

Come secondo esempio consideriamo una clausola con una sottometa sia a sinistra che a destra del taglio, ciascuna con due alternative:



I casi principali sono rappresentati dai seguenti alberi (solo parzialmente specificati):

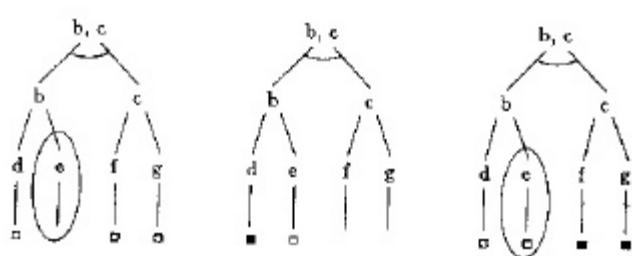


Figura 6.2.

Nel primo albero, **b** riesce con **d**, l'alternativa **e** non viene quindi considerata, e si passa a **c**. Poiché il taglio non influisce sulle mete alla sua destra, è possibile, una volta soddisfatta **c** con **f**, tornare indietro e risoddisfarla con **g**. Nel secondo albero, **b** fallisce con **d** ma riesce con **e**; purché **c** non fallisca (cioè o **f** o **g** o entrambe possano essere soddisfatte) non si ha alcuna potatura. Nel terzo albero, **b** riesce con **d**, e **c** fallisce perché sia **f** che **g** non possono essere soddisfatte. Si ha un fallimento immediato di **a** senza considerare la possibilità di risoddisfare **b** con **e**.

Analogo comportamento si ha se a destra od a sinistra del taglio vi sono più sottomete: la ricerca, mediante ritorno indietro, delle loro soluzioni è esaustiva, ma se falliscono tutte la metà testa della clausola fallisce immediatamente, senza riconsiderare le sottomete alla sinistra del taglio.

Un terzo esempio è quello in cui la clausola che contiene il taglio ha essa stessa delle alternative:



I casi possibili sono rappresentati dai seguenti tre alberi (anch'essi solo parzialmente specificati):

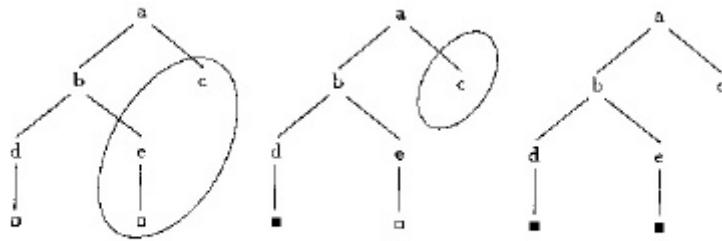


Figura 6.3.

Si vede come, se **b** riesce, la seconda clausola di **a** non viene mai considerata (e quindi, nel caso c possa essere soddisfatta, si perde la corrispondente soluzione). Lo stesso vale qualunque sia il numero di clausole alternative e successive a quella che contiene il taglio.

Per esaminare meglio i casi in cui la clausola che contiene il taglio ha delle alternative, è più opportuno considerarla nel contesto della clausola che ne contiene la chiamata:

```
a :- b, c, d.
c :- e, !, f.
c :- g.

```

Supponiamo che, nell'esecuzione di **a**, sia stata soddisfatta **b** e venga chiamata **c**. Se **e** fallisce, viene provata la terza clausola. Se invece **e** riesce, occorre ora soddisfare **f**, e poi **d**; entrambe possono essere soddisfatte più volte, mediante ritorni indietro. Se il soddisfacimento (od il risoddisfacimento) di **f** fallisce, non viene effettuato alcun tentativo di risoddisfare **e**, quindi **c** fallisce immediatamente, e non viene effettuato alcun tentativo di soddisfarla con la terza clausola. Il sistema tenta invece di risoddisfare la metà **b** e, in caso di successo, di soddisfare **c** e **d**, come se venissero invocate per la prima volta.

Nel caso generale di predici con argomenti, ad esempio:

a(...) :- **b(...), c(...), d(...)**.

c(...) :- **e(...), !, f(...)**.

c(...) :- **g(...)**.

valgono, come si è detto, le considerazioni precedenti, ricordando che, perché una clausola venga attivata, la sua testa deve essere unificabile con la chiamata. Perciò la terza clausola non viene considerata se avviene l'unificazione della chiamata **c(...)** con la testa della seconda clausola, e la metà **e(...)** riesce, attivando il taglio. Oltre al caso in cui la metà **e(...)** fallisce, viene invece tentata l'unificazione della chiamata **c(...)** con la testa della terza clausola, se fallisce l'unificazione con quella della seconda.

Per predici con argomenti, hanno perciò senso anche clausole in cui il taglio sia la prima ed eventualmente unica sottometà, in quanto l'applicabilità della clausola è determinata dalla configurazione degli argomenti d'ingresso.

Il seguente gruppo di esempi specifici di programmi che utilizzano il taglio è volto a illustrare ed evidenziare il duplice aspetto di questo predicato, già accennato inizialmente: da un lato esso è uno strumento utilizzabile per organizzare il controllo in modo da far fronte alle necessità pratiche della programmazione, e da questo punto di vista si mostreranno gli usi più tipici e ricorrenti; dall'altro lato esso influenza la computazione, sia rispetto alla sua efficienza (ottenere l'effetto desiderato con il minore utilizzo possibile di tempo e di memoria), sia rispetto alla sua corrispondenza o meno con il significato logico delle clausole (che influisce sulla comprensibilità del programma). A questo riguardo si evidenzieranno i casi in cui l'uso di questo predicato extra-logico compromette l'interpretazione dichiarativa tipica della programmazione logica. Si noti però che queste due schematizzazioni sono puramente indicative, sia perché ciascuna di esse non necessariamente esaurisce tutte le possibilità, sia perché i due aspetti sono sempre compresenti e strettamente intrecciati. Gli esempi proposti sono quindi da considerare casi di utilizzo di un predicato predefinito il cui effetto sul flusso di controllo del programma è comunque sempre governato dalle regole viste in precedenza.

Il taglio nelle strutture di selezione.

Data una relazione definita da più clausole, la chiamata di procedura con unificazione seleziona un'unica clausola se le forme delle teste sono tra loro mutuamente esclusive. Si consideri ad esempio la seguente procedura:

```
tipo_lista([], 'lista vuota').
tipo_lista([_|_], 'lista non vuota').
```

Se nella meta:

?- tipo_lista(L, A).

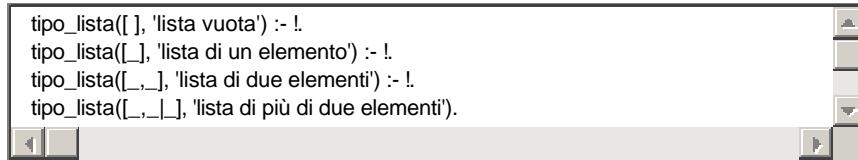
L è istanziato a una lista, **A** viene istanziato all'atomo 'lista vuota' oppure 'lista non vuota' (fornendo così una sorta di messaggio). Se **L** è istanziato alla lista vuota, un eventuale tentativo di risoddisfare la meta con la seconda clausola fallisce, non essendo possibile l'unificazione. D'altra parte è noto a priori (al programmatore, non all'interprete) che il secondo caso non può presentarsi se si è presentato il primo. Il programmatore può "comunicare" all'interprete che in questo caso è inutile cercare di applicare la seconda clausola, ponendo nella prima clausola un taglio, che opera come conferma della scelta fatta:

```
tipo_lista([], 'lista vuota') :- !.
tipo_lista([_|_], 'lista non vuota').
```

Il taglio nella seconda clausola è inutile, dato che non ve ne sono altre; si noti inoltre la maggiore semplicità di questa formulazione rispetto a [quella ricorsiva](#). È evidente che in questo caso il significato dichiarativo della procedura è lo stesso con il taglio o senza; ma è altrettanto evidente che non si ha un grande guadagno di efficienza.

Si generalizza facilmente per più casi:

```
tipo_lista([], 'lista vuota') :- !.  
tipo_lista([_], 'lista di un elemento') :- !.  
tipo_lista([_, _], 'lista di due elementi') :- !.  
tipo_lista([_, _, _], 'lista di più di due elementi').
```



Proseguendo in questo semplice esempio, se nella metà:

?- tipo_lista(L, A).

L è un termine diverso da una lista, la procedura fallisce, fornendo - ma solo implicitamente - tale indicazione. Volendo esplicitare questo caso, si può pensare di completare la (prima) procedura come segue:

```
tipo_lista([], 'lista vuota') :- !.  
tipo_lista([_|_], 'lista non vuota') :- !.  
tipo_lista(_, 'non lista').
```



La situazione ora, apparentemente analoga alla precedente, è in realtà molto diversa; infatti questa volta le tre forme del primo argomento non sono mutuamente esclusive, bensì le prime due sono casi particolari della terza, e quindi ogni istanza che unifichi con la prima o con la seconda unificherebbe anche con la terza. Senza il taglio la procedura, dopo aver dato la prima soluzione '**lista vuota**' oppure '**lista non vuota**', in caso di risoddisfacimento darebbe anche '**non lista**' come seconda "soluzione". Il problema sta nel fatto che si vorrebbe che la terza clausola si applicasse solo nel caso di non applicabilità delle prime due, ma non vi è modo di dare al primo argomento una forma tale da risultare complementare ad esse, cioè tale da escludere la lista. L'uso del taglio ha proprio questo effetto, ma il significato dichiarativo della procedura con il taglio non è più lo stesso di quella che ne è priva.

Nell'esempio precedente la soluzione migliore è ancora la prima considerata: infatti scopo della procedura è discriminare tra il caso vuoto o non vuoto di una lista, e se in ingresso non si ha una lista è "giusto" che essa fallisca. In molti casi però non è possibile discriminare un ingresso solamente sulla base della sua forma (la sola cosa su cui agisce l'unificazione), ma occorrono delle condizioni aggiuntive. Consideriamo ad esempio la relazione **minore(X, Y, Z)**: "**Z** è il minore tra i due numeri **X** e **Y**". La definizione è la seguente:

```
minore(X, X, X).  
minore(X, Y, Y) :- X > Y.  
minore(X, Y, X) :- X < Y.
```



Qui con la forma degli argomenti si può distinguere se i due ingressi sono uguali o diversi, ma in questo secondo caso occorre una condizione aggiuntiva per stabilire qual'è il minore. La definizione ha un significato dichiarativo lampante. Proceduralmente, però, presenta una ridondanza; infatti le clausole sono mutuamente esclusive, ma la seconda e la terza lo sono solo per la condizione che compare come sottometa. Perciò se la seconda clausola è riuscita (sia come chiamata che come sottometa), un eventuale tentativo di risoddisfacimento farebbe riuscire la chiamata della terza clausola, ma fallire la sua sottometa: sarebbe cioè un tentativo prevedibilmente inutile. Questa ridondanza non è molto rilevante in sè, ma diventa più significativa quando la si considera nel contesto di un'altra procedura che la richiama più volte, come nella seguente definizione della relazione **minimo(X, L)**: "X è l'elemento minimo della lista di numeri L":

minimo(M, [M]).

minimo(Min, [T1, T2 | C]) :- minimo(M, [T2 | C]), minore(T1, M, Min).

Nella procedura minore si può evitare la ridondanza prima menzionata utilizzando il taglio come conferma della clausola scelta:

minore(X, X, X) :- !.

minore(X, Y, Y) :- X > Y, !.

minore(X, Y, X).

o, ancora più succintamente:

minore(X, Y, Y) :- X >= Y, !.

minore(X, _, X).

Infine si ottiene il programma:

```
minimo(M, [M]).  
minimo(Min, [T1, T2 | C]) :- minimo(M, [T2 | C]), minore(T1, M, Min).  
minore(X, Y, Y) :- X >= Y, !.  
minore(X, _, X).
```

Il guadagno in efficienza è ottenuto distruggendo la corrispondenza fra interpretazione dichiarativa e procedurale: se si leggono queste due clausole ignorando il taglio, esse non definiscono la relazione desiderata. L'uso del taglio ha infatti sostituito la presenza di una condizione, dichiarativamente essenziale. Inoltre esso ha l'effetto voluto solo con quel particolare ordinamento delle clausole, che diventa quindi cruciale, mentre era indifferente nella procedura senza taglio.

Le considerazioni svolte finora ed illustrate dagli esempi precedenti possono essere generalizzate. L'uso del taglio per confermare la scelta di una regola si presta a realizzare in modo efficiente tutte le procedure che definiscono una relazione articolata in casi; esse consistono cioè di un insieme di clausole, ciascuna delle quali tratta un particolare possibile tipo di ingresso, e non è applicabile in corrispondenza ad altri ingressi. In tali circostanze si può utilizzare il taglio per segnalare

all'interprete Prolog l'inapplicabilità di alternative alla clausola selezionata, in modo da evitare ad esso, nel caso di ritorno indietro, di sprecare tempo e memoria in inutili tentativi.

In questo tipo di utilizzo del predicato di taglio, si possono considerare le sottomete che lo precedono entro una clausola come le condizioni che, oltre alla forma della testa, specificano la classe di ingressi da trattare allo stesso modo. Se tali condizioni sono soddisfatte, la clausola che le contiene va considerata come l'ultima utilizzabile per quella procedura, anche se ve ne sono altre, ed anche se le sottomete successive al taglio dovessero fallire. Se il taglio è la prima, ed eventualmente unica, sottometa presente nella clausola, le condizioni sono rappresentate unicamente dalla forma degli argomenti della testa della clausola, da cui dipende la riuscita della sua attivazione.

Se la procedura non è ricorsiva, questo uso del taglio corrisponde ad organizzare il flusso di controllo della procedura secondo una modalità di selezione tra (due o più) casi, in modo analogo ai costrutti di tipo "if-then-else" e "case" dei linguaggi procedurali strutturati. Gli schemi di controllo sono allora i seguenti.

Per la selezione tra due casi abbiamo:

selezione_tra_due_casi :- condizione, !, operazione_1.

selezione_tra_due_casi :- operazione_2.

dove condizione, **operazione_1** ed **operazione_2** rappresentano una congiunzione di un qualunque numero, eventualmente zero, di sottomete. Se è possibile soddisfare la metà condizione si passa ad eseguire la metà **operazione_1** e la seconda clausola non potrà più essere utilizzata, neanche in corrispondenza all'eventuale fallimento di **operazione_1** o di mete ad essa susseguenti; in caso contrario il ritorno indietro conseguente al fallimento di condizione porta ad utilizzare la seconda clausola, con l'esecuzione della metà **operazione_2**. **selezione_tra_due_casi** realizza perciò lo stesso effetto della struttura "if-then-else".

Per la selezione tra più di due casi si ha un'immediata generalizzazione dello schema precedente:

selezione_tra_n_casi :- condizione_1, !, operazione_1.

selezione_tra_n_casi :- condizione_2, !, operazione_2.

...

selezione_tra_n_casi :- condizione_n, !, operazione_n.

selezione_tra_n_casi :- operazione_n+1.

Anche qui, ogni condizione od operazione può consistere di nessuna, una o più sottomete. Quando una clausola viene attivata e la condizione riesce, l'operazione corrispondente è l'unica eseguita dalla procedura, anche se fallisce. L'ultima clausola gioca il ruolo di "altrimenti" (otherwise), applicandosi nel caso complementare a tutti i precedenti; viene omessa se le condizioni precedenti esauriscono tutti i casi possibili. In tal caso diviene inutile il taglio presente nella n-esima clausola. Nei due esempi seguenti si useranno i predicati predefiniti **atomic**, **var**, **atom**, **number**, **integer**, veri se il loro argomento è rispettivamente una costante, una variabile, un atomo, un numero, un intero; e il predicato predefinito **arg**, vero se il secondo argomento è una struttura.

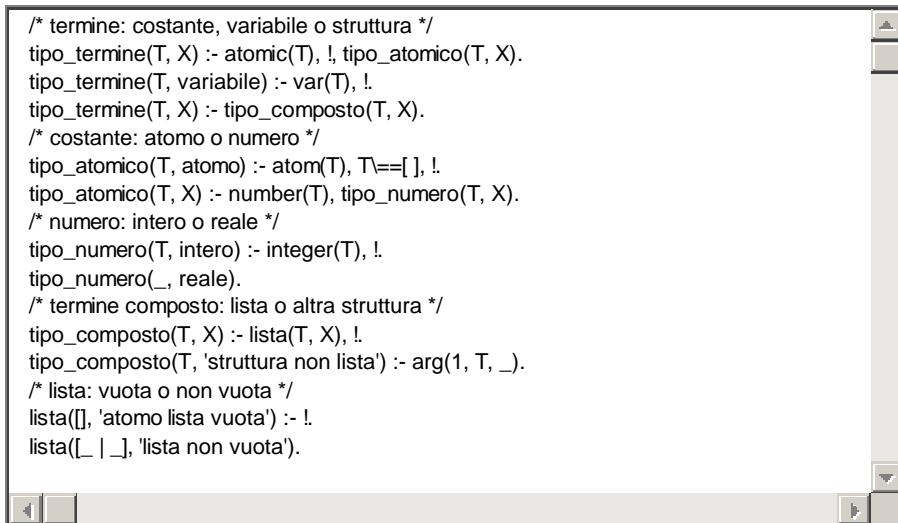
La seguente procedura **termine(T, X)** istanzia **X** ad un atomo che descrive il tipo di termine al quale è istanziato **T** in ingresso:

```
termine(T, costante) :- atomic(T), !.
termine(T, variabile) :- var(T), !.
termine(T, struttura) :- arg(1, T, _).
```



Si possono avere più selezioni interne l'una all'altra, come nell'esempio seguente:

```
/* termine: costante, variabile o struttura */
tipo_termine(T, X) :- atomic(T), !, tipo_atomico(T, X).
tipo_termine(T, variabile) :- var(T), !.
tipo_termine(T, X) :- tipo_composto(T, X).
/* costante: atomo o numero */
tipo_atomico(T, atomo) :- atom(T), T\==[], !.
tipo_atomico(T, X) :- number(T), tipo_numero(T, X).
/* numero: intero o reale */
tipo_numero(T, intero) :- integer(T), !.
tipo_numero(_, reale).
/* termine composto: lista o altra struttura */
tipo_composto(T, X) :- lista(T, X), !.
tipo_composto(T, 'struttura non lista') :- arg(1, T, _).
/* lista: vuota o non vuota */
lista([], 'atomo lista vuota') :- !.
lista([_|_], 'lista non vuota').
```



Un costrutto di tipo "if-then-else".

Alcune implementazioni di Prolog forniscono un costrutto corrispondente ad "if-then-else"; per esempio nel Prolog/DEC-10 e nel CProlog esso si scrive come:

P -> Q ; R.

equivalente a: "if **P** then **Q** else **R**". La sua definizione in Prolog, per i sistemi che non la ammettono come procedura predefinita, è:

P -> Q ; R :- call(P), !, call (Q).

P -> Q ; R :- call (R).

dove le lettere maiuscole stanno per predicati. **call** è un predicato predefinito tale che la meta **call(M)** riesce se **M** è soddisfatta. Nei sistemi che ammettono tale costrutto è anche disponibile il seguente:

P -> Q.

da intendersi equivalente a:

P -> Q ; fail.

che si legge: "se la meta **P** è soddisfatta, cerca di soddisfare la meta **Q**, in caso contrario fallisci". È un caso particolare del costrutto precedente, valido quando non sono previste alternative alla meta **Q**.

Consideriamo, il seguente problema: data una lista **L1** ed un elemento **E**, quest'ultimo va inserito in **L1** se non vi compare già, altrimenti va cancellato da **L1**; sia **L2** la lista che si ottiene in uscita. Una formulazione possibile della relazione è la seguente:

aggiornamento(E, L1, L2) :- appartenessa(E, L1), cancellazione(E, L1, L2).

aggiornamento(E, L1, L2) :- non_appartenessa(E, L1), concatenazione(L1, [E], L2).

dove **non_appartenessa** è definita come segue:

non_appartenessa(_, []).

non_appartenessa(E, [T | C]) :- E\== T, non_appartenessa(E, C).

e cancellazione, è definita come segue:

cancellazione(_, [], []).

cancellazione(E, [E|C], C).

cancellazione(E, [T|C1], [T|C2]):-E\== T,cancellazione(E, C1, C2).

Infine, aggiungendo la definizione di [appartenessa](#) e di [concatenazione](#), si ottiene:

```
aggiornamento(E, L1, L2) :- appartenessa(E, L1), cancellazione(E, L1, L2).
aggiornamento(E, L1, L2) :- non_appartenessa(E, L1), concatenazione(L1, [E], L2).
non_appartenessa(_, [ ]).
non_appartenessa(E, [T | C]) :- E\== T, non_appartenessa(E, C).
cancellazione(_, [], []).
cancellazione(E, [E|C], C).
cancellazione(E, [T|C1], [T|C2]):-E\== T,cancellazione(E, C1, C2).
appartenessa(E, [E|_]).
appartenessa(E, [_|C]):-appartenessa(E, C).
concatenazione([], L, L).
concatenazione([T|L1], L2, [T|L3]):-concatenazione(L1, L2, L3).
```

Il programma presenta chiaramente l'inefficienza dovuta al doppio controllo di appartenessa: una volta effettuato con esito negativo nella prima clausola, viene rieseguito nella seconda. Il problema

può essere superato utilizzando un taglio oppure usando il costrutto "if-then-else" discusso in precedenza.

Vi è un'altra possibilità di eliminare la necessità di procedure separate per la descrizione di relazioni complementari quali **appartenenza** e **non_appartenenza**, che si fonda sulla computazione esplicita del risultato del controllo:

aggiornamento_1(E, L1, L2) :- responso_appartenenza(E, L1, Risposta), aggiornamento_2(E, L1, L2, Risposta).

aggiornamento_2(E, L1, L2, sì) :- cancellazione(E, L1, L2).

aggiornamento_2(E, L1, L2, no) :- concatenazione(L1, [E], L2).

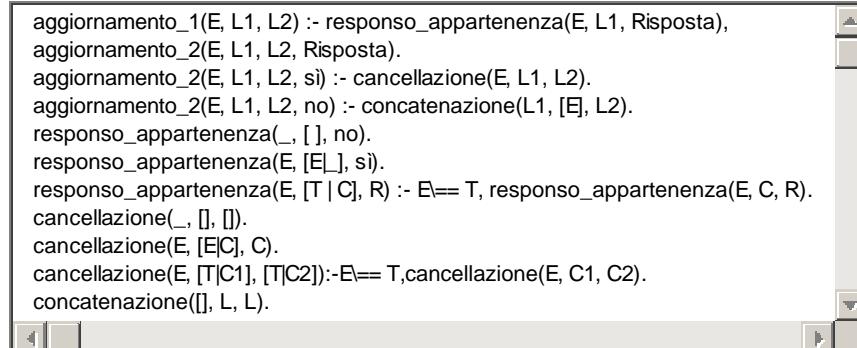
responso_appartenenza(_, [], no).

responso_appartenenza(E, [E|_], sì).

responso_appartenenza(E, [T | C], R) :- E\== T, responso_appartenenza(E, C, R).

La procedura **responso_appartenenza** computa una risposta **sì** o **no** e la passa ad **aggiornamento_2**, che provvede ad invocare l'una o l'altra delle procedure di **cancellazione** e **concatenazione**.

Infine si ottiene:



The screenshot shows the source code of a Prolog program. The code consists of several clauses defined in the following order:

- aggiornamento_1(E, L1, L2) :- responso_appartenenza(E, L1, Risposta), aggiornamento_2(E, L1, L2, Risposta).**
- aggiornamento_2(E, L1, L2, sì) :- cancellazione(E, L1, L2).**
- aggiornamento_2(E, L1, L2, no) :- concatenazione(L1, [E], L2).**
- responso_appartenenza(_, [], no).**
- responso_appartenenza(E, [E|_], sì).**
- responso_appartenenza(E, [T | C], R) :- E\== T, responso_appartenenza(E, C, R).**
- cancellazione(_, [], []).**
- cancellazione(E, [E|C], C).**
- cancellazione(E, [T|C1], [T|C2]) :- E\== T, cancellazione(E, C1, C2).**
- concatenazione([], L, L).**

Il programma ha il pregio di essere efficiente, logicamente corretto ed indifferente all'ordinamento delle clausole della procedura. Il metodo utilizzato ha inoltre carattere generale, ma richiede un argomento aggiuntivo nel predicato a cui è applicato, e diviene poco soddisfacente per problemi che richiedono un ampio numero di verifiche su differenti relazioni, a causa della necessità di scrivere molte clausole atte a trattare tutti i casi possibili.

Il taglio nelle strutture ripetitive.

Consideriamo ora l'uso del predicato di taglio all'interno di procedure ricorsive, iniziando con l'esempio seguente. La procedura **somma_fino_a(N, S)**, chiamata con **N** istanziato ad un numero positivo, istanzia **S** alla somma dei numeri da **1** a **N**. Può essere realizzata con le clausole:

```

somma_fino_a(1, 1) :- !, /* la somma dei numeri da 1 a 1 è 1 */
    somma_fino_a(N, S) :- N1 is N-1, somma_fino_a(N1, S1), S is S1 + N. /* la somma

```

Anche qui, nel definire la relazione, l'intenzione è quella di dedicare la prima clausola al caso in cui risulta $N = 1$, e la seconda a tutti gli altri casi. Ma, poiché la forma dell'argomento nella seconda clausola comprende come caso particolare quella della prima (non sono mutuamente esclusive), un eventuale ritorno indietro causerà il tentativo di usare la seconda clausola anche nel caso $N = 1$. Il taglio posto nella prima clausola ha quindi lo scopo di escludere questa possibilità. Esso non cambia il significato dichiarativo della procedura e ne migliora l'efficienza, come si può vedere più chiaramente utilizzandola in un contesto, ad esempio con la meta:

?- somma_fino_a(3, S), fittizio(non_esiste).

Quando la meta **fittizio(non_esiste)** fallisce, il ritorno indietro comporterebbe, in assenza del taglio, la ricerca di un'altra soluzione, ricerca inutile in quanto un'altra soluzione non esiste. Il taglio nella prima clausola, che è la condizione di terminazione della ricorsione, rende la procedura deterministica, in questo caso congruentemente con il fatto che per qualunque valore d'ingresso maggiore di zero la relazione ammette un'unica soluzione.

Nell'esempio precedente, la procedura opera dall'alto verso il basso, perché è dato un elemento superiore, che è il valore d'ingresso. La procedura, diminuendo progressivamente tale valore, incontra necessariamente la condizione limite, che assicura la terminazione. Se invece un valore superiore non può essere prefissato, ma solo descritto mediante una condizione, la procedura deve operare dal basso verso l'alto. Consideriamo per esempio la ricerca del minimo numero naturale X che soddisfi una certa condizione $p(X)$. Si può usare il seguente schema di programma:

naturale_minimo(X):- naturale(X), p(X), !.

naturale(0).

naturale(N) :- naturale(N1), N is N1 +1.

Il quesito:

?- naturale_minimo(X).

attiva la sottometa **naturale(X)** che, con la prima clausola, istanzia X a 0. Se **p(0)** fallisce, il ritorno indietro porta ad ottenere, con la seconda clausola, un diverso valore di X : prima 1, poi 2 e così via, sino a che la condizione **p(X)** risulta soddisfatta. A questo punto, l'esecuzione del taglio impedisce, in un eventuale ritorno indietro, la ricerca di un altro numero in grado di soddisfare **p**.

La procedura **naturale** svolge il ruolo di generatore di numeri naturali, uno alla volta a partire da 0, sotto il controllo della procedura **naturale_minimo** che li utilizza, sottponendoli alla verifica costituita dall'altra procedura **p**.

Consideriamo due casi specifici di questo schema di programma. Come primo caso, la relazione **troncamento(R, N)**: **N** è il massimo intero non superiore al reale positivo **R** (ossia è il troncamento di **R**; in alcune implementazioni questo è un predicato prede finito):

```
troncamento(R, N) :- R > 0, naturale(N), R1 is R - N, R1 > 0 , R1 < 1, !.
naturale(0).
naturale(N) :- naturale(N1), N is N1 +1.
```

Il taglio ha correttamente l'effetto di notificare al sistema che la prima istanza di troncamento trovata è l'unica possibile, e dunque è inutile porsi alla ricerca di soluzioni alternative che non solo non esistono, ma la cui ricerca porterebbe il programma a divergere. Naturalmente questa procedura è inefficiente, e questo evidenzia il fatto che il procedimento di generazione e verifica (generate and test) è problematico se usato indiscriminatamente; nell'esempio, lo stesso scopo può essere ottenuto con la procedura:

```
troncamento_1(R, 0) :- R > 0, R < 1.
troncamento_1(R, N) :- R > 0, R1 is R - 1, troncamento_1(R1, N1), N is N1+1.
```

che funziona più efficientemente e non ha bisogno di tagli.

Diverso è invece il seguente caso, nel quale il predicato **numero_primo(N)** esprime la proprietà che il numero naturale **N** è un numero primo:

numero_primo(N) :- naturale(N), primo(N).

con il predicato **primo** definito opportunamente. Questa procedura sarebbe evidentemente poco significativa se si limitasse a trovare il primo numero primo, mentre ha senso come generatrice di successivi numeri primi, e perché ciò avvenga non deve contenere il taglio alla fine della clausola. Senza il taglio la procedura può essere risoddisfatta indefiniteamente, in quanto ammette infinite soluzioni (a parte ovviamente i limiti fisici della memoria). Il controllo della sua terminazione è quindi rinviato al livello superiore del contesto di utilizzo, ad esempio:

procedura_che_usa_un_numero_primo(N) :- numero_primo(N), utilizzo(N), !.

come d'altra parte si è già fatto con la procedura **naturale** usata senza taglio, con terminazione controllata dall'altra procedura **naturale_minimo**. In generale, quindi, lo schema di procedura:

generazione_e_prova_primo(X) :- generazione(X), prova(X), !.

ha come interpretazione procedurale: "trova il primo **X** che soddisfa la proprietà **prova(X)**", mentre lo schema di procedura:

generazione_e_prova_di_tutti(X) :- generazione(X), prova(X).

ha come interpretazione procedurale: "trova tutti gli **X** che soddisfano la proprietà **prova(X)**". Il comportamento è con esecuzione dal basso verso l'alto con terminazione definita internamente o a cura del chiamante, rispettivamente, a seconda dei contesti di utilizzo.

I casi precedenti riguardavano procedure ad uscita multipla (cioè nelle quali, senza taglio, per un dato ingresso si possono ottenere, mediante ritorno indietro, più soluzioni) ma con un solo modo di utilizzo (un solo modo utile di istanziare le variabili nella chiamata). Consideriamo ora esempi di procedure che, oltre che ad uscita multipla, sono a più modi d'uso. Per esempio la procedura:

appartenenza(E, [E | _]).

appartenenza(E, [_ | C]) :- appartenza(E, C).

già considerata in precedenza, può essere usata sia per verificare (invocandola con entrambi gli argomenti istanziati) l'appartenenza di un elemento dato ad una lista data, sia per fornire (chiamandola con il primo argomento non istanziato) tutti gli elementi di una lista data, producendone uno per ogni ritorno indietro, fino ad esaurimento.

Per il primo modo d'uso la procedura può essere resa più efficiente con un taglio sulla condizione limite, che evita la ricerca di un'altra soluzione successiva alla prima:

appartenenza(E, [E | _]) :- !.

appartenenza(E, [_ | C]) :- appartenza(E, C).

mentre per il secondo modo d'uso deve essere usata la versione senza taglio se si vogliono ottenere tutte le soluzioni.

La lettura dichiarativa del predicato **appartenenza(E, L)** è: "l'elemento **E** appartiene alla lista **L**" indipendentemente da come sono istanziati gli argomenti. Il significato procedurale dipende invece dagli istanziamenti degli argomenti nella chiamata (cioè dai modi d'uso), ed è diverso nelle due versioni. La versione senza taglio ha il significato procedurale: "verifica l'appartenenza di un elemento ad una lista se entrambi sono assegnati, oppure fornisci uno alla volta gli elementi di una lista assegnata". La versione con il taglio ha invece il significato procedurale: "verifica l'appartenenza di un elemento ad una lista se entrambi sono assegnati, oppure fornisci il primo elemento di una lista assegnata". In tutti i casi nei quali si è interessati ad una sola soluzione, pur sapendo che possono esservene altre, e non volendo escludere che la procedura definita possa generarle tutte in corrispondenza ad altre chiamate, il taglio va posto non nelle clausole di definizione della procedura, bensì all'interno delle clausole della procedura generatrice, in congiunzione con le mete relative alla procedura interessata. Per esempio, definita la procedura:

gioca(marina, tennis).

gioca(gianni, pallavolo).

gioca(paolo, ping_pong).

volendo conoscere il nome di una persona che sta giocando è sufficiente invocare la procedura:

chi_gioca(Persona) :- gioca(Persona, _), !.

Aggiungendo i fatti precedenti si ottiene:

```
chi_gioca(Persona) :- gioca(Persona, _), !.  
gioca(marina, tennis).  
gioca(gianni, pallavolo).  
gioca(paolo, ping_pong).
```

Questo uso del taglio presenta il vantaggio di rendere più visibile il punto di chiamata di una procedura che richiede una sola soluzione, di non impedire alla procedura di generare tutte le soluzioni quando venga chiamata direttamente, e - infine - di evitare di porre un taglio su ogni clausola della procedura, cosa particolarmente sconveniente soprattutto quando le clausole sono molte. Alcune versioni di Prolog, come l'LM-Prolog, rendono disponibile un predicato predefinito rivolto a questo scopo, chiamato per esempio **once**, che ha l'effetto di richiedere una sola soluzione della chiamata di procedura che compare come suo argomento. Come si è visto in precedenza, una procedura ricorsiva può contenere in generale più clausole ricorsive e più condizioni limite; anche in questo ambito, ciascuna clausola può essere preposta al trattamento di uno dei casi possibili, che si presentano non solo in funzione dell'ingresso ma anche in funzione delle modifiche operate sull'ingresso dalle precedenti computazioni originate dalla ricorsione. Il predicato di taglio può essere utilizzato anche all'interno di clausole ricorsive, per migliorare l'efficienza della procedura. Consideriamo ad esempio la procedura **cancellazione_1(E, L1, L2)**, con il significato dichiarativo: "la lista **L2** è uguale alla lista **L1** privata di ogni occorrenza dell'elemento **E**":

```
cancellazione_1(_, [], []).  
cancellazione_1(E, [E | C], L) :- cancellazione_1(E, C, L).  
cancellazione_1(E, [T | C1], [T | C2]) :- E== T, cancellazione_1(E, C1, C2).
```

La prima clausola ricorsiva viene utilizzata quando il primo elemento della lista corrente coincide con l'elemento da sottrarre alla lista. La seconda clausola ricorsiva quando ciò non avviene. Ad ogni passo ricorsivo che non conduca ad ottenere la lista vuota viene quindi sfruttata la presenza di un punto di scelta. Poiché le tre clausole sono mutuamente esclusive, si può rendere più efficiente la procedura con l'utilizzo di due tagli:

```
cancellazione_2(_, [], []) :- !.  
cancellazione_2(E, [E | C], L) :- !, cancellazione_2(E, C, L).  
cancellazione_2(E, [T | C1], [T | C2]) :- E== T, cancellazione_2(E, C1, C2).
```

A differenza dettaglio sulla prima clausola, che opera una volta sola al termine della ricorsione, quello sulla seconda consente l'eliminazione di un'alternativa per ogni passo ricorsivo effettuato, e quindi un guadagno di efficienza più significativo.

Con considerazioni analoghe a quelle viste nel caso non ricorsivo, la presenza del taglio nella clausola precedente consente di eliminare la condizione aggiuntiva della terza clausola:

```
cancellazione_3(_, [], []) :- !.  
cancellazione_3(E, [E | C], L) :- !, cancellazione_3(E, C, L).  
cancellazione_3(E, [T | C1], [T | C2]) :- cancellazione_3(E, C1, C2).
```

Si ha un ulteriore guadagno di efficienza, più significativo che nel caso non ricorsivo; allo stesso modo però esso fa venir meno la corrispondenza del significato procedurale con quello dichiarativo.

Si può aggiungere che se l'elemento da cancellare non è contenuto nella lista, questa viene restituita immutata. Per fare in modo che in tal caso la procedura fallisca basta modificare la prima clausola in modo che anch'essa, come le altre due, non unifichi con la lista vuota. Con ritorno indietro però si ottengono ora tante risposte quante sono le occorrenze nella lista dell'elemento da cancellare, ognuna con un'occorrenza in meno; per ottenere per prima la soluzione senza alcuna occorrenza bisogna invertire l'ordine delle prime due clausole, e per avere solo quella è necessario aggiungere un taglio. La procedura diventa quindi:

```
cancellazione_4(E, [E | C], L) :- cancellazione_4(E, C, L), !.  
cancellazione_4(E, [E | C], C).  
cancellazione_4(E, [T | C1], [T | C2]) :- E== T, cancellazione_4(E, C1, C2).
```

Determinismo dichiarativo e procedurale.

Si sono discusse in precedenza due caratteristiche importanti delle procedure Prolog:

- il non-determinismo: per un singolo ingresso possono essere restituite più uscite (ovvero un quesito può avere più soluzioni, od una soluzione può essere ottenuta in più modi); il non determinismo è simulato dall'interprete Prolog mediante la strategia di discesa in profondità con ritorno indietro;
- l'invertibilità, o multi-direzionalità, dei parametri: non vi è una distinzione prefissata tra parametri di ingresso e parametri di uscita negli argomenti di una procedura, bensì (in linea di principio) i valori ignoti possono essere calcolati a partire da quelli noti in qualunque modo richiesto. Ciò rende la procedura utilizzabile in diversi modi.

Queste caratteristiche consentono di esprimere in modo chiaro e sintetico definizioni che richiederebbero costruzioni più lunghe e complesse nei linguaggi di programmazione tradizionali. Tuttavia esse comportano un costo, in tempo di esecuzione ed occupazione di memoria, che risulta superfluo in tutti quei casi nei quali le procedure definite nel programma non necessitano di meccanismi così generali. In tali casi è preferibile fare ricorso a caratteristiche più restrittive di minor costo computazionale. La possibilità teorica di usare una procedura in tutte le possibili

combinazioni di ruoli di ingresso e di uscita dei parametri è in realtà limitata da alcuni fattori, quali i seguenti:

- spesso le relazioni definite dalle procedure sono funzionali, ossia non ammettono più di una soluzione per ogni valore degli argomenti d'ingresso (ovvero l'uscita è funzione dell'ingresso), quindi non necessitano di un ritorno indietro generalizzato;
- in molti programmi accade che alcuni predicati in essi definiti sono usati sempre con alcuni degli argomenti come parametri d'ingresso ed altri come parametri d'uscita, e non richiedono l'invertibilità dei ruoli;
- molti dei predicati predefiniti (tipicamente quelli di valutazione aritmetica) possono essere usati solo in una direzione prefissata; questa direzionalità viene necessariamente ereditata dalle procedure che li chiamano, direttamente o indirettamente;
- la semplice strategia standard del Prolog può condurre a cicli senza fine per alcune combinazioni di ruolo dei parametri di procedure, che invece operano correttamente per altre combinazioni; quindi l'inversione dei ruoli non è sempre innocua, perché può comportare la non terminazione delle procedure.

Per queste ragioni un'attenzione particolare meritano i predicati che sono o possono essere resi deterministici; e - come si è visto - il taglio ha appunto l'effetto di rendere deterministica una procedura. È opportuno perciò approfondire questo aspetto, distinguendo innanzi tutto fra determinismo dichiarativo (o logico) e determinismo procedurale.

Un predicato è logicamente deterministico quando esprime una relazione funzionale, tale cioè che per ogni dato valore in ingresso ammette al più un valore in uscita. Ad esempio, nel caso di un predicato a due argomenti **p(X, Y)**, **Y** è una funzione di **X** se per ogni istanza **a** di **X** esiste al più un'istanza **t** di **Y** tale che **p(s, t)** è conseguenza logica delle clausole che definiscono **p**. In generale, un predicato **p(X1, X2,..., Xn)** è funzionale rispetto ad un argomento o ad un sottoinsieme degli argomenti se per ogni loro istanza esiste al più una n-pla contenente quell'istanza che è conseguenza logica delle clausole definitorie. E chiaro che un predicato funzionale rispetto ad un argomento può non essere tale rispetto agli altri. Un predicato è proceduralmente deterministico quando la sua definizione nel programma o le mete con le quali viene richiamato all'interno di esso sono tali che ogni meta fornisce al più una soluzione; essa può riuscire o fallire, ma non può attivare il ritorno indietro per la ricerca di soluzioni alternative. Sulla base della strategia standard di Prolog, una meta logicamente deterministica non è necessariamente deterministica proceduralmente. Ad esempio, la procedura:

p(a).

p(X) :- p(X).

è logicamente deterministica, poiché **a** è l'unica istanza di **p**, ma non è proceduralmente deterministica perché questa istanza può essere prodotta infinite volte. Il predicato di taglio, quando è usato per rendere proceduralmente deterministico un predicato che è tale logicamente, ne mantiene il significato dichiarativo e ne migliora l'efficienza procedurale. Questo è il caso, per esempio, della procedura [somma_fino_a\(N, S\)](#) che è logicamente deterministica (per ogni numero positivo **N** esiste un solo valore della somma **S** dei numeri da **1** ad **N**), e con il taglio diviene tale anche proceduralmente, in quanto esso evita l'inutile ricerca di un'altra soluzione. Il determinismo non dipende, tuttavia, solo da come un predicato è definito, ma anche da come è chiamato: una particolare chiamata di un predicato (con una specifica configurazione di ingresso e di uscita degli argomenti) può essere funzionale (rispetto a quegli ingressi) anche se tale non è in generale (per altre configurazioni). Consideriamo per esempio la procedura [concatenazione](#). La meta:

?- concatenazione([a,b,c], [d,e], Z).

è logicamente deterministica, perché l'unica istanza possibile per **Z** è **[a,b,c,d,e]**. Proceduralmente, dopo la prima soluzione l'interprete Prolog può tornare indietro a cercarne un'altra; per evitare questa inutile ricerca si può usare il taglio nella prima clausola:



```
concatenazione([], L, L) :- !.  
concatenazione([T | L1], L2, [T | L3]) :- concatenazione(L1, L2, L3).  
[ ] [ ] [ ] [ ]
```

Tuttavia in questo modo si rende la procedura operazionalmente deterministica per ogni meta. Ad esempio, la meta:

?- concatenazione(X, Y, [a,b,c]).

otterrà l'unica risposta:

X=[] Y=[a, b, c]

mentre dichiarativamente non è deterministica, poiché esistono le ulteriori soluzioni:

X=[a] Y=[b,c];

X=[a,b] Y=[c];

X=[a,b,c] Y=[]

In questo caso il taglio ha reso l'interpretazione procedurale non più corrispondente a quella dichiarativa, distruggendo la completezza della risoluzione, che non può più trovare le altre soluzioni. Per la procedura **concatenazione** sono logicamente deterministiche le mete nelle quali almeno due dei tre argomenti vengono forniti in ingresso, in corrispondenza ai seguenti usi:

1. con tutti e tre gli argomenti in ingresso, per verificare la relazione;
2. con **L1** e **L2** in ingresso, e **L3** in uscita, per ottenere la lista concatenata;
3. con **L1** (o **L2**) e **L3** in ingresso, per ottenere in uscita in **L2** (o in **L1**) la lista complementare.

Sono invece non deterministiche le mete nelle quali almeno due degli argomenti sono d'uscita, per esempio:

4. con **L1** e **L2** in uscita, per ottenere la scomposizione della lista **L3** in ingresso.

Il caso 4. è l'opposto del caso 2., e mostra il carattere "reversibile" della procedura, ossia la possibilità di scambiare i ruoli dei parametri di ingresso e di uscita. Mentre per i primi tre modi d'uso può essere utilizzata la versione con il taglio, questa non è utilizzabile per il modo d'uso 4., che richiede di lasciare aperta la ricerca di soluzioni alternative. In generale si può osservare che qualunque predicato è logicamente deterministico quando tutti gli argomenti vengono utilizzati

come parametri d'ingresso; inoltre, se è logicamente deterministico per una certa assegnazione dei ruoli d'ingresso e di uscita dei parametri, è tale anche per ogni altra assegnazione ricavata dalla precedente facendo diventare d'ingresso uno o più parametri d'uscita. Per esempio, il predicato **p** definito da:

p(a, b, c).

p(d, b, c).

p(b, b, d).

p(c, d, d).

è logicamente deterministico se il primo parametro è d'ingresso e gli altri due d'uscita, quindi è tale anche se o il secondo, o il terzo, od entrambi sono d'ingresso insieme con il primo. Non è invece logicamente deterministico negli altri casi. Si è accennato prima che lo scambio di ruoli dei parametri può portare una procedura a divergere. E il caso ad esempio delle procedure [inversione](#) e [inversione_1](#), che calcolano con successo, nel secondo argomento, l'inverso di una lista assegnata come primo argomento, ma entrano in un ciclo infinito, in caso di ritorno indietro, con una meta nella quale la lista è data nel secondo anziché nel primo argomento. Per evitare questo si può definire, per **inversione**:

```
invers(L1, L2) :- inversione(L1, L2), !.
inversione([], []). /* l'inversa della lista vuota è la lista vuota */
inversione([T|C], L2):- inversione(C, L1), concatenazione(L1, [T], L2). /* l'inversa di
una lista non vuota è uguale all'inversa della coda concatenata con la lista il cui
unico elemento è la testa.*/

```

e per **inversione_1** analogamente, oppure, sfruttando il fatto che questa ha una procedura ausiliaria:

```
invers_1(L1, L2) :- inv(L1, [], L2), !.
inversione_1(L1,L2):-inv(L1, [],L2).
inv([],L, L).
inv([T|C], L1, L2):-inv(C, [T|L1], L2).
```

Si è visto inoltre, nell'[Interpretazione procedurale](#), come non sia sempre possibile trovare un ordinamento dei predicati nel corpo delle clausole, e delle clausole stesse tra loro, che si riveli appropriato per ogni possibile uso della procedura. Diversi aspetti concorrono pertanto a rendere opportuna l'introduzione di una dichiarazione di modo (di uso), ovvero un assegnamento di modo (o di direzione) di ingresso o di uscita degli argomenti. Ogni argomento di un predicato può assumere uno fra tre modi possibili, come segue:

- "<" (modo d'ingresso): quando il predicato è invocato, l'argomento è un termine chiuso (ovvero completamente istanziato, cioè istanziato ad un termine diverso da una variabile e non contenente variabili non istanziate);

- ">" (modo uscita): quando il predicato è invocato, l'argomento è una variabile libera;
- "<>" (modo indifferente): quando il predicato è invocato, l'argomento può essere un termine qualsiasi, cioè completamente o parzialmente istanziato, oppure una variabile libera.

Il modo di una chiamata di un predicato consiste allora nell'assegnamento di uno dei tre possibili modi a ciascuno dei suoi argomenti; se, in un programma, un predicato è chiamato sempre con lo stesso modo, questo verrà detto semplicemente il modo del predicato. Per esempio (ma senza riferimento ad una particolare sintassi), la dichiarazione:

concatenazione(>, >, <)

indica che la procedura **concatenazione** verrà sempre chiamata con i primi due argomenti non istanziati ed il terzo istanziato, manifestando quindi l'intenzione di utilizzarla per generare in uscita (nei primi due argomenti) tutte le possibili separazioni di una lista (assegnata in ingresso al terzo argomento).

I modi dei predicati di sistema sono predefiniti, e valgono per qualunque programma nel quale compaiono; per esempio il predicato predefinito **is** ha il modo:

is(<>, <)

ossia può venire utilizzato sia per istanziare la variabile libera che eventualmente figura come primo argomento, sia per verificare l'uguaglianza dei due argomenti, ma mai per generare istanze di variabili a secondo argomento.

La conoscenza del modo dei predicati (o delle loro chiamate) in un programma può essere utilizzata per vari scopi:

- per verificare che una procedura sia utilizzata coerentemente con l'eventuale dichiarazione di modo per essa fornita; i sistemi che implementano la dichiarazione di modo, il primo dei quali è stato il Prolog/DEC-10, possono fornire una segnalazione d'errore in caso contrario;
- per determinare, sulla base di condizioni sufficienti, i casi nei quali la mancanza di verifica di occorrenza nell'unificazione è innocua, ed i casi nei quali - al contrario - dev'essere effettuata per evitare soluzioni non corrette o cicli infiniti;
- per indirizzare l'interprete all'utilizzo di una regola di selezione diversa da quella standard, applicando il criterio di selezionare per prime le mete più istanziate, discusso nella [Strutturazione del controllo](#). Alcuni sistemi, come l'IC-Prolog, implementano questa possibilità, dando all'utente la facoltà di richiederla mediante annotazioni di controllo nel programma;
- per consentire l'ottimizzazione della ricorsione in coda, che - come si è visto - si può applicare a procedure delle quali è noto che sono deterministiche;
- come efficace strumento di documentazione dell'utilizzo delle procedure in un programma.

Superamento tendenziale dell'uso del taglio.

In questo capitolo si è mostrato che vi sono usi pericolosi ed usi innocui del predicato predefinito di taglio, a seconda che esso alteri o meno il contenuto dichiarativo delle procedure ed escluda o meno dalla ricerca alcune delle loro possibili soluzioni; a volte vengono chiamati tagli rossi i primi, e tagli verdi i secondi. L'utilizzo di questo predicato di controllo del ritorno indietro è spesso utile o indispensabile per i fini pratici della programmazione logica; tuttavia esso incoraggia uno stile di programmazione che inficia le caratteristiche dichiarative che le sono proprie. Possibili soluzioni

tendenziali, introdotte sperimentalmente in alcune recenti versioni di Prolog, consistono nel considerare il taglio come una primitiva di basso livello, utilizzata internamente e mascherata da primitive di più alto livello che lo sostituiscono nell'utilizzo da parte dell'utente. Tra queste vi sono il costrutto "if-then-else" per le situazioni di mutua esclusione delle clausole, la dichiarazione di modi per le procedure deterministiche, ed il predicato once per indicare esplicitamente che si desidera una sola soluzione (la prima) anche se ve ne possono essere altre.

Note bibliografiche.

Van Emden (1982) ha proposto la distinzione fra tagli verdi (innocui) e tagli rossi (pericolosi). Tale distinzione è ripresa da Sterling e Shapiro (1986).

I problemi della multidirezionalità e della molteplicità d'uso delle procedure sono discussi, ad esempio, negli articoli di Kowalski (1974 e 1979b) e di McDermott (1980).

La nozione di funzionalità di una relazione è discussa in Debraig e Warren (1986) ed in Nakamura (1986).

Le nozioni di modo e di dichiarazione di modo per indicare le possibilità di utilizzo di un predicato in un programma Prolog sono state introdotte da Warren (1977). Una discussione sulla generazione automatica delle dichiarazioni di modo per programmi Prolog compare in Mellish (1981), ed una breve sintesi del metodo si trova in Mellish (1985). L'utilizzo dei modi in IC-Prolog è illustrato in Clark e McCabe (1980).

La dichiarazione di determinismo è stata introdotta in Nilsson (1984), dove si dimostra che ogni programma logico contenente il taglio può essere tradotto in un programma equivalente privo di esso utilizzando la dichiarazione di determinismo (ma anche questa può essere innocua o pericolosa). Il costrutto **once** è disponibile in LM-Prolog, e viene descritto da Carlsson e Kahn (1985).

Sommario.

Lo studente è ora a conoscenza delle possibilità offerte dal Prolog per il controllo del ritorno indietro, e degli aspetti da curare per cercare di bilanciare la necessità di ottenere procedure efficienti con quella di mantenere il loro significato dichiarativo.

Può ora esercitarsi nello scrivere procedure Prolog con un certo grado di sofisticazione per quanto riguarda la gestione del controllo.

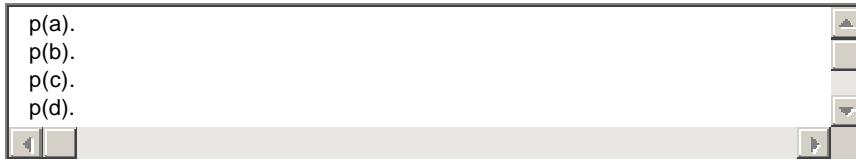
7. Strutturazione dei dati

Dove si considera la caratteristica delle clausole di potere rappresentare procedure sia di computazione che di memorizzazione esplicita di dati. E dove si esaminano i diversi aspetti della duplice possibilità di strutturare i dati con relazioni o con termini.

Rappresentazione di dati con relazioni e con termini.

Nella programmazione logica i dati possono essere rappresentati fondamentalmente in due modi: per mezzo di termini o per mezzo di relazioni. Una procedura Prolog costituita da clausole unitarie di base (i fatti) costituisce una rappresentazione diretta di una relazione, in quanto fornisce esplicitamente le n-ples che appartengono alla relazione. Tale definizione mediante elencazione di tutte le istanze vere della relazione è detta definizione estensionale. Ad esempio, nel caso di relazione unaria, la procedura:

```
p(a).  
p(b).  
p(c).  
p(d).
```



definisce tutti gli oggetti (**a**, **b**, **c** e **d**) che godono della proprietà **p**. Da un punto di vista dichiarativo tali oggetti costituiscono un insieme, ma proceduralmente formano una sequenza, in quanto le clausole sono scandite dal sistema Prolog nell'ordine sequenziale in cui sono scritte, ad esempio a fronte di quesiti come:

?- p(c).

?- p(h).

che chiedono se un oggetto **c** o **h** ha o meno la proprietà **p**, o come:

?- p(X).

che chiede quali oggetti godono della proprietà **p**.

I dati rappresentati come fatti possono tipicamente essere elaborati mediante scansione sequenziale con ritorno indietro, con una procedura ad esempio del tipo:

elabora_p :- p(X), elabora(X), fail.

elabora_p.

con **elabora(X)** opportunamente definito.

In linea di principio, qualunque struttura di dati può essere espressa estensionalmente. Per esempio si può rappresentare la lista **[1,4,9,16]** con i seguenti fatti:

elemento(lista, 1, 1).

elemento(lista, 2, 4).

elemento(lista, 3, 9).

elemento(lista, 4, 16).

dove la relazione **elemento(L, I, E)** significa: "E è l'I-esimo elemento della lista di nome L", ovvero: "nella lista L l'elemento E occupa la posizione I".

In questo esempio si è aggiunto un argomento che serve a dare un nome alla lista; questo è utile o necessario quando la compresenza nella base di dati di più liste rappresentate con clausole richiede un modo per distinguerle.

La rappresentazione della lista precedente può essere completata con il fatto:

lunghezza(lista, 4).

che definisce la **lunghezza** della lista considerata, ottenendo il programma:

```
elemento(lista, 1, 1).
elemento(lista, 2, 4).
elemento(lista, 3, 9).
elemento(lista, 4, 16).
lunghezza(lista, 4).
```

Quesiti del tipo:

?- elemento(lista, 2, E).

?- elemento(lista, I, 9).

che chiedono il valore o la posizione di un elemento, costituiscono un accesso diretto alla struttura di dati, nel senso che invocano soltanto la clausola che corrisponde. Si ha un reale accesso diretto efficiente nelle implementazioni di Prolog che forniscono l'indicizzazione delle clausole (clause indexing).

Indicizzazione delle clausole.

Nei casi in cui una relazione viene rappresentata estensionalmente mediante parecchi fatti, o per procedure dotate di molte regole, ciascuna delle quali corrisponde ad un particolare caso da trattare, è opportuno limitare - per quanto possibile - il numero di clausole che il sistema deve esplorare per trovare quella (o quelle) in grado di dare luogo ad una corrispondenza con la meta corrente.

A questo scopo è stato realizzato in alcuni sistemi (per la prima volta nella versione compilata del Prolog/DEC-10) un meccanismo di indicizzazione delle clausole secondo il funtore principale del

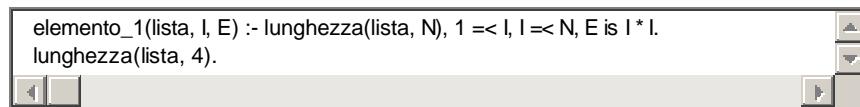
primo argomento della loro testa, in quanto si presume che questa sia la posizione generalmente adottata per l'ingresso principale di una procedura.

In tal modo le sole clausole da considerare per una possibile corrispondenza sono quelle che hanno lo stesso funtore, od una variabile, nella posizione indicizzata, a condizione che il primo argomento nella chiamata sia effettivamente un ingresso, cioè sia una costante, un termine strutturato od una variabile già istanziata: spesso si presenta addirittura una sola alternativa.

La lista delle clausole potenzialmente in grado di corrispondere viene trovata non più con una ricerca sequenziale, bensì mediante una tecnica di hash coding che richiede un tempo indipendente dai numero di clausole della procedura.

L'indicizzazione delle clausole presenta l'ulteriore pregio di aumentare l'efficienza del sistema migliorando la ricerca di situazioni deterministiche, in maniera da risparmiare informazione sui possibili punti di ritorno indietro.

L'utilizzo dell'opzione di indicizzazione delle clausole è in grado di avvicinare le prestazioni delle versioni interpretate del linguaggio a quelle del codice compilato. È chiaro che il ricorso a questa possibilità diventa di particolare efficacia quando il numero di clausole per una data procedura assume proporzioni significative. L'alternativa alla rappresentazione esplicita (estensionale) di una relazione è la sua definizione mediante una regola in grado di generarne le istanze; questo tipo di definizione è detto intensionale. Ad esempio, la stessa [lista precedente](#) può essere definita da:



```
element_1([1,2,3], 2, E) :- lunghezza([1,2,3], N), 1 =< I, I =< N, E is I * I.  
lunghezza([1,2,3], 4).
```

Il precedente quesito:

?- element_1([1,2,3], 2, E).

comporta in questo caso non più una ricerca tra un insieme di fatti stabiliti, bensì una computazione per calcolare l'elemento della lista desiderato.

Si noti che tutti i fatti considerati nella rappresentazione estensionale sono conseguenza logica della corrispondente definizione intensionale. Ciascuno di essi può infatti essere derivato con un singolo quesito avente i primi due argomenti istanziati; oppure essi possono essere tutti ottenuti consecutivamente, aggiungendo alle clausole precedenti una procedura di generazione di numeri positivi (opportunamente controllata) e ponendo il quesito con il secondo e terzo argomento non istanziati.

Usando invece la rappresentazione con termini, la stessa lista precedente può essere rappresentata globalmente mediante il singolo termine **[1,4,9,16]**. Per la risposta a quesiti come quelli precedenti occorre in questa rappresentazione definire specifiche procedure di accesso agli elementi, come la procedura [ennesimo](#). Ad esempio, il quesito:

?- ennesimo(4, [1,4,9,16], E).

decomponga progressivamente il termine che denota la lista, finché trova l'elemento richiesto.

La procedura **ennesimo** riceve la lista in ingresso; alternativamente, la lista può essere un termine di una clausola, ad esempio:

lista(lista, [1,4,9,16]).

e la procedura di accesso farà riferimento ad essa:

ennesimo_elemento(Nome, N, E) :- lista(Nome, L), ennesimo(N, L, E).

ottenendo il programma:

```
lista(lista, [1,4,9,16]).  
ennesimo_elemento(Nome, N, E) :- lista(Nome, L), ennesimo(N, L, E).  
ennesimo(1, [El|_], El).  
ernesimo(N, [_|Coda], El):-N1 is N - 1, ennesimo(N1, Coda, El).  
  
[<|>] [^|v]
```

Il comportamento è analogo al caso precedente.

In generale, i dati rappresentati con termini sono tipicamente elaborati con procedure ricorsive ausiliarie per l'accesso alle componenti, che comportano un maggiore lavoro computazionale di scomposizione e ricomposizione dei termini stessi. Per converso la rappresentazione con termini è più compatta, e particolarmente adatta per strutture di dati definibili ricorsivamente, come ad esempio gli alberi, per i quali la rappresentazione a clausole è meno conveniente. Quest'ultima si presta bene invece alla rappresentazione di grafi.

Come l'esempio precedente sulle liste ha mostrato, la rappresentazione intensionale si comporta come una procedura che genera dati d'uscita, e quella estensionale come una memorizzazione esplicita dei dati. La capacità delle clausole di esprimere entrambe le soluzioni indica come la tradizionale distinzione tra procedure e dati non va intesa di per sé, ma piuttosto in relazione agli scopi e ai modi di utilizzo delle rappresentazioni. La scelta dipende cioè dal problema, a seconda - per esempio - della quantità dei dati, della loro struttura regolare od irregolare, del tipo di elaborazioni da effettuare, della preferenza verso l'ottimizzazione dell'uso della memoria o del tempo di esecuzione, e così via. È utile perciò considerare le opzioni possibili e le loro combinazioni, piuttosto che una contrapposizione fra i due approcci.

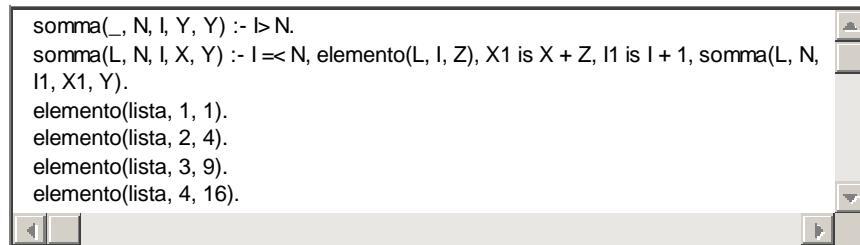
Nel caso in cui su una lista siano da effettuare operazioni che non si esauriscono nella ricerca di un elemento ma richiedono di conservare memoria dell'elaborazione man mano che questa viene compiuta, nella rappresentazione della lista come relazione la ricerca sequenziale con ritorno indietro non è sufficiente allo scopo, in quanto il ritorno indietro distrugge gli istanziamenti fatti precedentemente; occorre allora operare con procedure ricorsive, che mantengono memoria da una chiamata all'altra. Per esempio, per sommare tutti gli elementi di una lista occorre una procedura come la seguente:

somma(L, N, I, Y, Y) :- I>N.

```
somma(L, N, I, X, Y) :- I =< N, elemento(L, I, Z), X1 is X + Z, I1 is I + 1, somma(L, N, I1, X1, Y).
```

dove il significato di **somma(L, N, I, X, Y)** è: "per ogni **I** compreso tra 1 e la lunghezza **N** della lista **L**, **X** è la somma di tutti gli elementi di **L** che precedono l'**I**-esimo, e **Y** è il risultato del sommare **X** alla somma degli elementi dall'**I**-esimo all'**N**-esimo compresi

Nell'esempio della [lista precedente](#):



```
somma(_, N, I, Y, Y) :- I > N.
somma(L, N, I, X, Y) :- I =< N, elemento(L, I, Z), X1 is X + Z, I1 is I + 1, somma(L, N, I1, X1, Y).
elemento(lista, 1, 1).
elemento(lista, 2, 4).
elemento(lista, 3, 9).
elemento(lista, 4, 16).
```

la procedura va attivata con la meta:

```
?- somma(lista, 4, 1, 0, Y).
```

che inizializza i valori dei primi quattro argomenti. L'argomento **X** funge da accumulatore delle somme parziali; dopo **N** corrispondenze della chiamata con la seconda clausola, **I** diventa maggiore di **N** e viene attivata la prima clausola, che istanzia **Y** al valore di **X**.

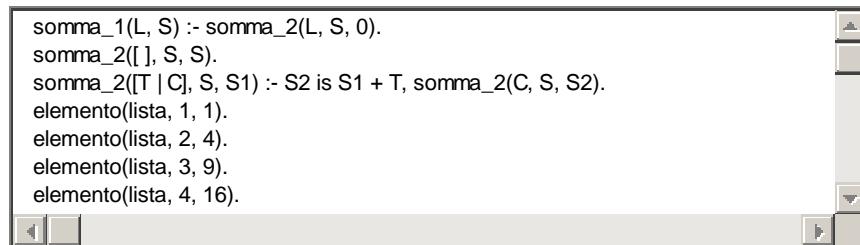
Se invece la lista è rappresentata da un termine, la seguente procedura ricorsiva che somma gli elementi non necessita dell'argomento aggiuntivo utilizzato prima come accumulatore:

```
somma_1(L, S) :- somma_2(L, S, 0).
```

```
somma_2([], S, S).
```

```
somma_2([T | C], S, S1) :- S2 is S1 + T, somma_2(C, S2, S1).
```

dove **somma_1(L, S)** ha il significato: "**S** è la somma degli elementi della lista **L**", e va utilizzata con **L** istanziato al termine che rappresenta la lista. Nell'esempio precedente:



```
somma_1(L, S) :- somma_2(L, S, 0).
somma_2([], S, S).
somma_2([T | C], S, S1) :- S2 is S1 + T, somma_2(C, S2, S1).
elemento(lista, 1, 1).
elemento(lista, 2, 4).
elemento(lista, 3, 9).
elemento(lista, 4, 16).
```

```
?- somma_1([1,4,9,16], X).
```

X = 30

Un altro esempio delle due possibili rappresentazioni, con termini o con relazioni, è dato dal caso dei vettori (arrays) multidimensionali. Il termine:

[[10, 20, 30], [40, 50, 60], [70, 80, 90]]

tratta la matrice come lista di righe, ognuna delle quali è rappresentata dalla lista degli elementi. La relazione:

elem(E, I, J, M)

esprime che **E** è l'elemento di riga **I** e colonna **J** della matrice **M**. Per la [matrice precedente](#) si ha:

```
elem(10, 1, 1, m).  
elem(20, 1, 2, m).  
elem(30, 1, 3, m).  
elem(40, 2, 1, m).  
elem(50, 2, 2, m).  
elem(60, 2, 3, m).  
elem(70, 3, 1, m).  
elem(80, 3, 2, m).  
elem(90, 3, 3, m).
```



Una possibilità intermedia consiste nell'usare una relazione per rappresentare le righe, ed un termine per gli elementi della riga: la relazione **riga(I, L, M)** esprime che **L** è la lista degli elementi della riga **I** della matrice **M**. Nell'[esempio precedente](#) la rappresentazione diventa:

```
riga(1, [10,20,30], m).  
riga(2, [40,50,60], m).  
riga(3, [70,80,90], m).
```



Naturalmente per ognuno dei casi precedenti è possibile, simmetricamente, la rappresentazione della matrice per colonne.

La rappresentazione con termini può essere adeguata per elaborazioni sequenziali di tutti gli elementi di ogni riga e di tutte le righe della matrice; la rappresentazione a clausole può essere più conveniente nel caso di matrici sparse.

In generale, la definizione intensionale è possibile e conveniente quando i dati sono omogenei e presentano caratteristiche di regolarità, esprimibili appunto mediante una regola generale; la memorizzazione esplicita difatti è più appropriata quando i dati sono disomogenei o hanno una struttura irregolare. Si può adottare una combinazione delle due rappresentazioni nel caso di regole con eccezioni. Come semplice esempio, consideriamo il predicato **aula(Materia, Anno, Aula)**. L'insieme dei seguenti fatti:

```

aula(algebra, primo, 312).
aula(analisi, primo, 312).
aula(geometria, primo, 312).
aula(fisica, primo, 400).

```

può essere rappresentato sinteticamente da:

```

aula(fisica, primo, 400).
aula(M, primo, 312):- corso(M), M =\= fisica.
corso(algebra).
corso(analisi).
corso(geometria).

```

Strutturazione dei termini.

Quando si usano termini, questi possono essere strutturati in diversi modi (se ne è fatto accenno nel **capitolo 1**). Ad esempio, i seguenti sono due modi alternativi di rappresentare un fatto circa un corso di **analisi** tenuto il lunedì dalle **9** alle **11** da **mario rossi** nell'aula **312** del **settore_didattico**:

```

corso(analisi, lunedì, 9, 11, mario, rossi, settore_didattico, 312).
corso(analisi, orario(lunedì, 9, 11), docente(mario,rossi), luogo(settore_didattico, 312)).

```

Nel primo fatto **corso** è un predicato con 8 argomenti semplici, nel secondo con 4 argomenti, di cui 3 composti. La seconda definizione è più compatta e naturale, ma l'accesso alle componenti è tanto più complesso quanto più esse sono innestate in profondità. Definendo regole ausiliarie, è possibile passare da una rappresentazione all'altra:

```

corso(analisi, orario(lunedì, 9, 11), docente(mario,rossi), luogo(settore_didattico, 312)).

corso(Titolo, orario(Giorno, Ora_inizio, Ora_fine), docente(Nome, Cognome), luogo(Palazzo, Aula)) :- 
    corso(Titolo, Giorno, Ora_inizio, Ora_fine, Nome, Cognome, Palazzo, Aula).

corso(analisi, lunedì, 9, 11, mario, rossi, settore_didattico, 312).

corso(Titolo, Giorno, Ora_inizio, Ora_fine, Nome, Cognome, Palazzo, Aula) :-

```

corso(Titolo, orario(Giorno, Ora_inizio, Ora_FINE), docente(Nome, Cognome), luogo(Palazzo, Aula)).

Si noti incidentalmente che si è usata qui per la prima volta la possibilità, legittima in Prolog anche se non va a vantaggio della chiarezza, di avere nello stesso programma funtori con ugual nome e molteplicità diverse, che costituiscono a tutti gli effetti funtori diversi.

È anche possibile definire regole per estrarre singole componenti, come per esempio:

titolo(Titolo) :- corso(Titolo, _, _, _, _, _, _).

nella prima rappresentazione, oppure:

titolo(Titolo) :- corso(Titolo, _, _, _).

nella seconda rappresentazione. Similmente è possibile isolare sottoinsiemi di componenti, come nella procedura:

corso(Titolo, Docente) :- corso(Titolo, _, _, _, _, Docente, _, _).

oppure:

corso(Titolo, docente(Nome, Cognome)) :- corso(Titolo, _, _, _, Nome, Cognome, _).

nella prima rappresentazione, oppure:

corso(Titolo, Docente) :- corso(Titolo, _, Docente, _).

nella seconda rappresentazione.

Nell'esempio precedente si ha:

```
corso(analisi, lunedì, 9, 11, mario, rossi, settore_didattico, 312).
corso(analisi, orario(lunedì, 9, 11), docente(mario, rossi), luogo(settore_didattico, 312)).
corso(Titolo, Docente) :- corso(Titolo, _, _, _, _, Docente, _, _).
corso(Titolo, docente(Nome, Cognome)) :- corso(Titolo, _, _, _, Nome, Cognome, _).
corso(Titolo, Docente) :- corso(Titolo, _, Docente, _).
titolo(Titolo) :- corso(Titolo, _, _, _, _, _, _).
titolo(Titolo) :- corso(Titolo, _, _, _).
```

Regole del tipo suddetto rendono più flessibile la rappresentazione adottata, in quanto consentono l'accesso alle componenti per nome anziché per posizione, e rendono non necessaria la conoscenza della posizione prestabilita che gli argomenti hanno nel termine.

Rappresentazione ennaria e binaria di relazioni.

In generale ogni relazione ennaria può essere riespressa come congiunzione di **n + 1** relazioni binarie. Per esempio, la relazione:

corso(algebra, rossi, primo).

(che descrive il corso di nome algebra, docente rossi e anno primo), può essere alternativamente espressa dall'insieme delle seguenti relazioni binarie:

è_un(c1, corso).

nome(c1, algebra).

docente(c1, rossi).

anno(c1, primo).

Nella formulazione binaria si introduce una costante che nomina la ennupla della relazione (**c1** nell'esempio), e una costante che sostituisce il nome di predicato ennario (sopra indicati entrambi con **corso**). Ognuna delle **n** relazioni binarie (ciascuna con un diverso proprio nome di predicato) esprime la correlazione del nome dato alla ennupla della relazione ennaria con ognuno dei suoi argomenti, e così pure con la costante che sostituisce il nome del predicato ennario. Una diversa ennupla della stessa relazione ennaria, come:

corso(sistemi, bianchi, secondo).

sarà indicata con un nome diverso dal precedente nella corrispondente congiunzione di relazioni binarie; nell'esempio:

è_un(c2, corso).

nome(c2, sistemi).

docente(c2, bianchi).

anno(c2, secondo).

Si noti che la rappresentazione ennaria e quella binaria sono diverse anche per **n = 2**.

Con le calusole precedenti si ottiene il programma:

```
corso(algebra, rossi, primo).
corso(sistemi, bianchi, secondo).
è_un(c1, corso).
è_un(c2, corso).
nome(c1, algebra).
nome(c2, sistemi).
docente(c1, rossi).
docente(c2, bianchi).
anno(c1, primo).
anno(c2, secondo).
```

Se nelle ennuple della relazione almeno un argomento assume valori tutti distinti tra loro si possono omettere le costanti che nominano le singole ennuple e la costante che sostituisce il nome del predicato ennario, cosicché sono sufficienti n relazioni binarie. Nell'esempio:



```
è_un(algebra, corso).
è_un(sistemi, corso).
docente(algebra, rossi).
docente(sistemi, bianchi).
anno(algebra, primo).
anno(sistemi, secondo).
```

Se la relazione è unaria, è sempre sufficiente una sola relazione binaria, che esprime come l'originario nome di predicato, che diviene una costante, è in relazione con il suo argomento; ad esempio:

mortale(X).

diventa:

è_un(X, mortale).

La rappresentazione binaria ha il vantaggio che argomenti non noti o non interessanti della originaria relazione ennaria possono essere ignorati. Per esempio, supponendo che il corso simulazione sia un corso complementare che può essere seguito in qualsiasi anno, nella rappresentazione ennaria occorre comunque indicare il terzo argomento:

corso(simulazione, verdi, X).

oppure:

corso(simulazione, verdi, _).

mentre nella rappresentazione binaria viene semplicemente omesso:

è_un(simulazione, corso).

docente(simulazione, verdi).

Si può dire che la rappresentazione binaria consente di trattare una relazione come se avesse un numero variabile di argomenti. Questa caratteristica rende più facile aggiungere nuova informazione. Supponiamo ad esempio di volere specificare anche la sigla del corso; nella rappresentazione ennaria occorre cambiare il predicato, in quanto cambia la sua molteplicità, mentre in quella binaria basta aggiungere altre relazioni:

sigla(algebra, 1021).

sigla(sistemi, 1022).

sigla(simulazione, 1040).

La rappresentazione binaria consente anche di parlare di ennuple della relazione, ciò che può tornare utile in diversi casi. Per esempio, si possono correlare delle ennuple tra loro:

propedeutico(c1, c2). /* il corso c1 è propedeutico al corso c2 */

oppure correlare alle ennuple altri individui, ad esempio:

studente(brambilla, c1). /* brambilla è uno studente del corso c1 */

ed in generale costruire oggetti strutturati a partire da oggetti non strutturati senza modificare i riferimenti che già esistono.

È da osservare tuttavia che in realtà è il trattare le ennuple come individui (con un nome) che fornisce i vantaggi della rappresentazione binaria illustrati negli ultimi due esempi. È possibile anche nella rappresentazione ennaria aggiungere un argomento che nomina la ennupla; ad esempio:

corso(c1; algebra, rossi, primo).

corso(c2, sistemi, bianchi, secondo).

propedeutico(c1, c2).

studente(brambilla, c1).

Più importante è il fatto che la rappresentazione binaria consente di descrivere i dati mediante leggi generali più di quanto non sia possibile con quella ennaria. Per esempio l'asserzione "bianchi tiene tutti i corsi di sistemi" può essere espressa con la regola:

docente(X, bianchi) :- nome(X, sistemi). /* se un corso è di sistemi, allora lo tiene bianchi */

La stessa regola non può essere espressa direttamente (in clausole di horn) usando il predicato ennario **corso**. Occorre appunto definire le relazioni ausiliarie, come visto in precedenza.

Nella rappresentazione binaria risulta perciò più agevole, ripetto a quella ennaria, esprimere la base di dati sia con fatti che con regole; queste ultime possono raggruppare quel sottoinsieme di fatti che nella rappresentazione ennaria hanno un argomento con lo stesso valore.

Naturalmente la rappresentazione binaria ha lo svantaggio, rispetto a quella ennaria, di una maggiore prolissità, in quanto i nomi degli oggetti sono ripetuti in ogni clausola; questo comporta una maggiore lunghezza delle regole e dei quesiti. Per esempio, dati un insieme di fatti corrispondenti rispettivamente alle seguenti due rappresentazioni:

corso(Corso, Giorno, Aula, Nome_studente, Nome_docente).

e:

è_un(Istanza, Corso).

nome(Istanza, Corso).

giorno(Istanza, Giorno).

aula(Istanza, Aula).

studente(Istanza, Nome_studente).

docente(Istanza, Nome_docente).

il quesito "esiste uno studente tale che un docente gli insegna due diversi corsi nella stessa aula 7" viene espresso rispettivamente nei due modi seguenti:

?- corso(C1, G1, A, Ns, Nd), corso(C2, G2, A, Ns, Nd).

?- è_un(I1, corso), nome(I1, C1), giorno(I1, G1), aula(I1, A), studente(I1, Ns), docente(I1, Nd), è_un(I2, corso), nome(I2, C2), giorno(I2, G2), aula(I2, A), studente(I2, Ns), docente(I2, Nd).

Astrazione procedurale e astrazione sui dati.

Quando i dati sono rappresentati con termini, risultano generalmente interconnessi con le procedure che operano su essi; è possibile mantenere una certa indipendenza dei dati dalle procedure di più alto livello del programma, ma ciò richiede una particolare cura nel realizzare le opportune astrazioni procedurali.

Quando i dati sono rappresentati con relazioni, risultano automaticamente separati dalle procedure preposte ad operare su essi; queste possono persino mancare del tutto, essendo sempre possibile porre nella metà le operazioni richieste. In questo caso il programma definisce un più alto livello di astrazione sul problema, rappresentando le informazioni in modo del tutto indipendente dalle operazioni che si vorranno compiere su esse, che potranno cambiare a seconda degli scopi.

La rappresentazione con relazioni fornisce un modo naturale ed immediato per realizzare astrazioni sui dati, ed aiuta a ridurre un problema ai suoi aspetti più essenziali.

Riprendiamo l'esempio di una base di dati costituita da un insieme di punti su un piano, espressi con coordinate **X** e **Y**; supponiamo che i punti dati siano:

```
p(0, 0).  
p(10, 0).  
p(20, 0).  
p(0, 10).  
p(10, 10).  
p(20, 10).  
p(0, 20).  
p(10, 20).  
p(20, 20).
```

Si possono porre diversi tipi di quesiti, come i seguenti. Per ottenere tutti i segmenti verticali (esclusi quelli di lunghezza nulla e quelli speculari):

?- p(X, Y1), p(X, Y2), Y1<Y2.

Per ottenere quadrati:

?- p(X1, Y1), p(X2, Y1), D is X2 - X1, D > 0, p(X1, Y2), D is Y2 - Y1, p(X2, Y2).

Per ottenere triangoli isosceli con base orizzontale:

?- p(X1, Y1), p(X2, Y2), DHB is X2 - X1, DHB > 0, DV is Y2 - Y1, DV > 0, p(X3, Y1), DHB is X3 - X2.

In linea di principio si può porre ogni genere di quesito, sia per ottenere tutte le figure di un certo tipo, che per ottenere figure specifiche, ad esempio figure speculari, figure con certi punti prefissati, figure con certi lati di lunghezza assegnata, o figure con i lati in una relazione specificata.

Avendo descritto solo i fatti, il programma non si impegna in alcun modo con operazioni predeterminate, lasciando totalmente libero l'utilizzatore di richiedere ciò a cui volta per volta può essere interessato. Un tale programma è molto diverso da quello ottenibile con un linguaggio procedurale, dove il programma realizza un insieme prestabilito di operazioni sui dati; operazioni diverse non possono essere ottenute, se non modificando il programma (ciò che in genere risulta laborioso).

Naturalmente, per converso, è lasciato all'utente anche l'onere, oltre che la libertà, di specificare i quesiti di suo interesse; per figure complesse tali quesiti possono essere di non immediata formulazione. Poiché, come si è visto in precedenza, caratteristico delle clausole è il fatto che un qualunque quesito può essere trasformato in una procedura, possono essere definite nel programma, insieme ai fatti, le procedure corrispondenti a quelli che si possono ipotizzare come i quesiti più frequenti; per i due ultimi esempi precedenti si possono introdurre le procedure:

```
quadrati(X1, X2, Y1, Y2) :- p(X1, Y1), p(X2, Y1), D is X2 - X1, D > 0, p(X1, Y2), D is Y2 - Y1, p(X2, Y2).
isosceli(X1, X2, X3, Y1, Y2) :- p(X1, Y1), p(X2, Y2), DHB is X2 - X1, DHB>0, DV is - Y1, DV>0, p(X3, Y1), DHB is X3 - X2.
p(0, 0).
p(10, 0).
p(20, 0).
p(0, 10).
p(10, 10).
p(20, 10).
p(0, 20).
```

ed i corrispondenti quesiti risultano semplificati:

?- quadrati(X1, X2, Y1, Y2).

?- isosceli(X1, X2, X3, Y1, Y2).

Un'altra possibilità, per facilitare il compito dell'utente, è aggiungere al programma alcune procedure di base, in termini delle quali formulare quesiti più articolati. Si possono introdurre ad esempio le definizioni di linee orizzontali e verticali, e considerare le figure come liste di linee orizzontali e verticali:

```

p(0, 0).
p(10, 0).
p(20, 0).
p(0, 10).
p(10, 10).
p(20, 10).
p(0, 20).
p(10, 20).
p(20, 20).

/* Una lista di linee è orizzontale se non vi sono linee o se la prima linea è orizzontale e il resto delle linee sono orizzontali */
orizzontali([]).
orizzontali([Testa | Resto_della_lista]) :- orizz(Testa), orizzontali(Resto_della_lista). /* Una linea è orizzontale se non contiene locazioni; oppure contiene una sola locazione; oppure contiene due locazioni con punti di uguale coordinata Y e diversa coordinata X, e la seconda locazione costituisce una linea orizzontale con il resto della lista */
orizz([]).
orizz([_]).
orizz([loc(p(X1, Y), D1), loc(p(X2, Y), D2) | Resto]) :- p(X1, Y), p(X2, Y), X2 =\= X1, D1 is X2 - X1, orizz([loc(p(X2, Y), D2) | Resto]). /* Analogamente per le linee verticali */
verticali([]).
verticali([Testa | Resto_della_lista]) :- vert(Testa), verticali(Resto_della_lista).
vert([]).
vert([_]).
vert([loc(p(X, Y1), D1), loc(p(X, Y2), D2) | Resto]) :- p(X, Y1), p(X, Y2), Y1 =\= Y2, D1 is Y2 - Y1, vert([loc(p(X, Y2), D2) | Resto]). /* ass(X, Y): Y è il valore assoluto di X */
ass(A, A) :- A >= 0.
```

Quesiti che utilizzano tali definizioni sono ad esempio i seguenti. Per ottenere figure a L comprese le immagini speculari, con verticali di lunghezza doppia delle orizzontali:

?- orizzontali([[loc(P1, DO),loc(P2, 0)]]), ass(DO, DOA), verticali([[loc(P1, DV),loc(P3, 0)]]), ass(DV, DVA), DVA is DOA *2.

Per ottenere figure a T che si intersecano in p(10, 20):

?- orizzontali([[loc(P1, DO),loc(p(10, 20), DO), loc(P3, 0)]]), verticali([[loc(P4, DV),loc(p(10, 20))]]), DO>0, DV>0.

Di nuovo, per facilitare la ricerca di figure a T si può inserire nel programma la procedura seguente:

due_direzioni(L1, L2) :- orizzontali(L1), verticali(L2).

due_direzioni(L1, L2) :- orizzontali(L2), verticali(L1).

ottenendo:

```

p(0, 0).
p(10, 0).
p(20, 0).
p(0, 10).
p(10, 10).
p(20, 10).
p(0, 20).
p(10, 20).
p(20, 20).

/* Una lista di linee è orizzontale se non vi sono linee o se la prima linea è orizzontale e il resto delle linee sono orizzontali */
orizzontali([]).

orizzontali([Testa | Resto_della_lista]) :- orizz(Testa), orizzontali(Resto_della_lista). /* Una linea è orizzontale se non contiene locazioni; oppure contiene una sola locazione; oppure contiene due locazioni con punti di uguale coordinata Y e diversa coordinata X, e la seconda locazione costituisce una linea orizzontale con il resto della lista */

orizz([]).

orizz([_]).

orizz([loc(p(X1, Y), D1), loc(p(X2, Y), D2) | Resto]) :- p(X1, Y), p(X2, Y), X2 =\= X1, D1 is X2 - X1, orizz([loc(p(X2, Y), D2) | Resto]). /* Analogamente per le linee verticali */

verticali([]).

verticali([Testa | Resto_della_lista]) :- vert(Testa), verticali(Resto_della_lista).

vert([]).

vert([_]).

vert([loc(p(X, Y1), D1), loc(p(X, Y2), D2) | Resto]) :- p(X, Y1), p(X, Y2), Y1 =\= Y2, D1 is Y2 - Y1, vert([loc(p(X, Y2), D2) | Resto]). /* ass(X, Y): Y è il valore assoluto di X */

ass(A, A) :- A >= 0.

```

Tra le due soluzioni estreme - da un lato quella in cui il programma definisce solo fatti e tutta l'elaborazione è rimandata alla formulazione del quesito (massima generalità, difficoltà di uso), e dall'altro lato quella in cui il programma predefinisce completamente (come nella programmazione tradizionale) le sole operazioni possibili sui dati (programma particolare, di facile uso) - le clausole consentono una gamma continua di possibilità intermedie. In esse le procedure aggiuntive ai fatti, orientate alla classe dei problemi considerati più che ad un procedimento di soluzione predeterminato per uno specifico problema, costituiscono "interfacce" che ne facilitano l'utilizzo da parte dell'utente, senza tuttavia vincolano ad operazioni particolari, in quanto rimane comunque possibile accedere direttamente ai fatti mediante appositi quesiti.

Di nuovo, e più in generale, le clausole consentono di riconsiderare, e ricomporre, molte delle dicotomie classiche della programmazione tradizionale, quali le opposizioni di procedure a dati, di programmi a basi di dati, di astrazioni procedurali ad astrazioni sui dati.

Note bibliografiche.

La prima formulazione, nell'ambito della programmazione logica, della rappresentazione di una relazione ennaria in termini di relazioni binarie si trova nell'articolo di Deliyanne e Kowalski (1979). Una panoramica più generale su logica e strutturazione dei dati è fornita in Gallaire e Minker (1978).

Un resoconto dettagliato sulle modalità tipiche di rappresentazione interna dei termini nelle implementazioni del linguaggio è contenuto in Warren (1977), mentre gli schemi di indicizzazione per la memorizzazione delle clausole e l'accesso ad esse sono esposti in Clark e McCabe (1980). Entrambi gli argomenti sono trattati anche da Warren e Pereira (1977) nel loro confronto tra Prolog e Lisp.

L'esempio riguardante punti e figure e relative procedure è ripreso dall'articolo di Swinson (1980).

Sommario.

Lo studente ha ora un quadro di alcune opzioni possibili nel rappresentare i dati relativi ad un problema, e nell'individuare quella combinazione tra rappresentazione con termini e con clausole, e tra dati e procedure, più consona alla natura ed agli scopi della sua applicazione.

8. Sviluppo, documentazione, riusabilità

Dove si descrive uno schema di documentazione di un programma logico, e si esaminano aspetti relativi alla sua modularizzazione e riusabilità. La programmazione logica non fa venir meno, ma anzi esalta, l'opportunità di seguire uno stile di programmazione ed una metodologia di sviluppo dei programmi. Avvicinare la comprensione statica del testo del programma al comportamento dinamico che esso evoca durante l'esecuzione è stato l'obiettivo della programmazione strutturata: la dualità di interpretazione dichiarativa e procedurale propria della programmazione logica favorisce questo avvicinamento, e dove le implementazioni di Prolog se ne discostano è possibile cercare di supplire con una metodologia di sviluppo e di documentazione. Obiettivi di una tale metodologia sono quelli classici dell'ingegneria del software nella sua versione più aggiornata: ridurre l'incidenza degli errori di programmazione e favorire la produzione di programmi più economici, più facilmente comprensibili e modificabili, e possibilmente portabili e riusabili.

Modularità.

Il fondamento per la buona documentazione di un programma in qualsiasi linguaggio è la possibilità di descrivere il significato dell'intero programma mediante la descrizione dei significati delle sottoparti, e delle loro interrelazioni. Perciò le possibilità di documentazione sono correlate a quelle di modularizzazione del programma. Nel seguito si esaminano sotto questo aspetto le caratteristiche favorevoli e meno favorevoli del Prolog, indicando uno schema di documentazione che mira ad integrare gli aspetti più carenti. Fra le caratteristiche di Prolog che favoriscono la modularità del programma si possono indicare le seguenti:

- la clausola come campo di validità lessicale delle variabili: ogni variabile è locale ad una clausola, e non esistono variabili globali (a meno dell'uso dei predicati assert, retract e loro consimili);
- il funzionamento "ad assegnamento singolo" delle variabili: una variabile istanziata fa riferimento ad un particolare oggetto del programma, per tutto il corso di una dimostrazione di risposta ad un quesito, a meno di ritorni indietro;
- la clausola costituisce una unità di significato, in quanto gode di un'interpretazione dichiarativa associata;
- la procedura, come insieme di clausole che definiscono una relazione, è necessariamente una unità funzionale autosufficiente;
- l'unificazione costituisce un meccanismo flessibile di interfacciamento tra procedure.

Vi sono tuttavia altre caratteristiche, o loro risvolti, che non vanno a vantaggio della modularità:

- la procedura è una unità logica, ma non ha in Prolog un costrutto che la evidenzia e la racchiude in una unità di programma: le clausole aventi lo stesso predicato di testa possono essere sparse ovunque nel programma;
- la presenza di tagli in una procedura può alterare il significato dichiarativo delle clausole, che non possono più essere interpretate singolarmente; può inoltre restringerne l'utilizzo ad uno o ad alcuni fra tutti gli usi possibili;
- il comportamento tipico della variabile logica di funzionare da parametro d'ingresso o d'uscita di una procedura a seconda del suo uso, se da un lato conferisce flessibilità alla procedura, dall'altro lato può determinare una anomalia di interfacciamento tra procedure quando - pur essendovi corrispondenza tra chiamata e procedura - questa viene usata con un passaggio di parametri in una direzione non prevista;

- a differenza dei nomi di variabili, che - essendo locali alle clausole - non pongono problemi di interferenza, i nomi delle procedure sono globali, ossia visibili ed utilizzabili da ogni parte del programma. Ciò ostacola la possibilità di sviluppare, od anche soltanto di capire, alcune sottoparti del programma sulla base di una conoscenza solo esterna, funzionale, delle altre sottoparti, in quanto occorre conoscere i nomi di tutte le procedure esistenti per evitare di scriverne altre con lo stesso nome, che interferirebbero con le precedenti. In particolare, per quelle relazioni che vengono definite unicamente come ausiliarie ad altre relazioni e non vengono mai utilizzate al di fuori di esse, non vi è modo di indicarle come "locali", e dunque di nessun interesse per il resto del programma;
- il programma risulta in una base di dati unica, nella quale non si individua una strutturazione in sottoparti più aggregate della singola procedura e dotate di un significato complessivo proprio.

Gli ultimi due aspetti sono riassumibili nella mancanza in Prolog di un concetto e di un costrutto come quello di modulo. Il concetto di modulo può essere inteso come un raggruppamento di un insieme di procedure, alcune delle quali hanno significato per diverse parti del programma, mentre altre sono definite unicamente in funzione di supporto alle prime ed hanno quindi una utilità unicamente locale. La realizzazione di tale concetto richiede un costrutto che dia facoltà all'utente di raggruppare insieme alcune procedure, e poi stabilire quali nomi di procedura devono essere visibili ("esportati") all'esterno del modulo e quali nascosti. Le procedure esportate da un modulo possono essere utilizzate ("importate") da altri moduli del programma.

Si consideri, a titolo esemplificativo, il caso della procedura [inversione_1](#), che utilizza la procedura ausiliaria **inv** che presumibilmente non viene invocata da nessun'altra parte del programma. Le due procedure possono allora essere confinate in un modulo:

```
/*
```

MODULO liste.

ESPORTA: inversione_1

```
*/
```

```
inversione_1(L1, L2) :- inv(L1, [], L2).
```

```
inv([], L, L).
```

```
inv([T | C], L1, L2) :- inv(C, [T | L], L2).
```

La notazione precedente può essere utilizzata come documentazione del programma, ma chiaramente di per sé non rende le procedure locali "opache" al resto del programma, dunque non elimina i problemi relativi all'uso ed alla gestione dei nomi di procedure.

Alcuni sistemi Prolog (il primo dei quali è stato l'ungherese MProlog) implementano il suddetto tipo di modularità, ossia la possibilità di partizionare la base di dati del programma in moduli, che comunicano con gli altri moduli attraverso un'interfaccia: ogni modulo è introdotto nel programma da una dichiarazione di modulo che ne specifica il nome, e contiene una specificazione di interfaccia, che dichiara i nomi dei predicati definiti nel modulo e visibili al suo esterno (predicati esportati) e quelli dei predicati utilizzati entro il modulo ma definiti al suo esterno (predicati importati). Tutti i predicati definiti all'interno di un modulo che non sono dichiarati esportati nella

specificazione di interfaccia risultano invisibili all'esterno del modulo, cioè non possono essere invocati quale meta da alcun predicato esterno. Il sistema supporta poi in modo congruente la gestione dello sviluppo, della consultazione e dell'esecuzione dei programmi modularizzati.

Uno schema di documentazione.

Un possibile schema di documentazione è il seguente:

/*

PROGRAMMA: <nome>

< descrizione del problema>

MODULO <nome>

ESPORTA: <lista dei predicati esportati>

IMPORTA: <lista dei predicati importati>

USA: <lista dei predicati predefiniti usati >

*/

/*

PROCEDURA: <nome procedura> (<lista agomenti>)

D: <interpretazione dichiarativa>

P: <interpretazione procedurale>

T: <tagli>

BD: < base dati>

C: < commento generale>

*/

< clausole commentate>

La documentazione del programma inizia con commenti che ne introducono il nome e descrivono (in forma libera) il problema trattato, cosa fa il programma e come usarlo. Seguono uno o più moduli. Ciascun modulo contiene una parte iniziale di commenti che introducono il nome del modulo, l'elenco dei predicati di utente esportati ed importati, e l'elenco dei predicati di sistema

utilizzati; i predicati sono indicati nella forma <nome del predicato/molteplicità>. Seguono una o più procedure definite nel modulo. Ciascuna procedura contiene una parte iniziale di commenti.

- il primo commento, che comincia con **PROCEDURA**, ne introduce il nome (lo stesso predicato usato come testa delle clausole della procedura) e la lista degli argomenti. Il secondo commento, che inizia con **D**, descrive cosa rappresentano gli argomenti, e quale relazione tra essi è definita dalla procedura. Ad esempio, nella procedura **appartenenza(Elemento, Lista)**, **Lista** è una lista di termini qualsiasi (ed **Elemento** è uno di essi), mentre nella procedura **somma(Lista, Somma)**, **Lista** può essere solo una lista di numeri.
- Nel commento che inizia con **P** sono elencati (se più di uno, numerati progressivamente: **P1**, **P2**, ...) e descritti uno per uno i diversi usi della procedura, con una <**lista argomenti annotati**> per ciascuno di essi, insieme con l'indicazione descrittiva del tipo di utilizzo (per verifica, per generazione, od altro), e se fornisce una o più soluzioni.
- La <**lista argomenti annotati**> riprende la <**lista argomenti**> facendo precedere ciascuno di essi dal simbolo "<", per indicare che è un parametro d'ingresso (dev'essere istanziato quando la procedura viene chiamata), oppure dal simbolo ">" per indicare che è un parametro d'uscita (dev'essere non istanziato), oppure dal simbolo "<>" per indicare che è un parametro che può essere usato in qualunque modo.
- Il commento che inizia con **T** indica la presenza o meno di tagli e, nel caso siano presenti, se questi alterano o meno l'interpretazione dichiarativa.
- Il commento che inizia con **BD** indica se la procedura modifica il contenuto della base di dati. In caso positivo, indica se la modifica vale solo durante il tempo di esecuzione o permane, al termine.
- Il commento che inizia con **C** chiarisce altri elementi generali che si può ritenere opportuno specificare ulteriormente.

Seguono le clausole che definiscono le procedure, nelle quali i commenti saranno disposti secondo convenienza, tenendo conto di un bilanciamento tra loro significatività e sinteticità della documentazione.

In riferimento a quanto sopra, si noti che:

- si è evidentemente ipotizzato che tutte le clausole costituenti la procedura siano raggruppate nel testo, il che non è richiesto in Prolog ma costituisce un requisito minimale per la documentazione della procedura;
- non è necessario un commento di chiusura del modulo o della procedura, in quanto essi risultano delimitati naturalmente dal primo commento del modulo o della procedura successivi;
- le parti che non si applicano al caso in esame vengono omesse. Se non si introducono moduli, i corrispondenti commenti vengono omessi; in questo caso si possono riportare nelle singole procedure i commenti che descrivono i predicati utente (per esempio cominciando con **U:**) ed eventualmente i predicati di sistema (**S:**) richiamati nella procedura. Se questa non contiene tagli, o non altera la base di dati, o non utilizza altre procedure, le corrispondenti righe di commento sono tralasciate; in generale, i primi tre tipi di commenti saranno però sempre presenti.
- lo schema di documentazione esemplificato vale per le versioni di Prolog che non contemplano i costrutti di modulo e di dichiarazione di modo; altrimenti, questi evidentemente sostituiscono i corrispondenti commenti.

Convenzioni di scrittura dei programmi.

La leggibilità di un programma Prolog può essere favorita seguendo alcuni accorgimenti nella scrittura dei programmi: la presentazione del programma non è meno importante del suo contenuto. Lo stile di programmazione è anche questione di gusti e di preferenze soggettive, dunque risulta passibile di variabilità; ciò che conta in ogni caso è che le convenzioni adottate siano congruenti ed uniformi in tutto il programma.

È consigliabile utilizzare nomi significativi per predicati, funtori e variabili (si ricordi che in Prolog non ci sono limiti a priori per la lunghezza di tali nomi). Per i predicati è preferibile la forma "dichiarativa" piuttosto che quella "imperativa"; ad esempio si consigliano: concatenazione, permutazione, inserimento, ..., piuttosto che concatena (o appendi), permuta, inserisci. Per le variabili, i nomi possono essere descrittivi del loro ruolo; si possono usare nomi standard abbreviati per i ruoli più ricorrenti, per esempio: **T** e **C** per "testa" e "coda" di una lista, **E** o **El** per "elemento" e così via. Variabili che compaiono una sola volta in una clausola possono essere anonime. Il programma:

```
proc(X, Y, Z) :- proc_1(X, Y), comp(X, Y, Z).

proc_1(X, Y) :- atom(X), atom(Y), !.

proc_1(X, Y) :- nl, write('Errore.'), nl.

comp(X, Y, Z) :- elab(X, X1), elab(Y, Y1), comp(X1, Y1, Z).
```

è meno leggibile del seguente, che pure ha gli stessi requisiti funzionali e lo stesso comportamento in esecuzione:

```
procedura(Ingresso_1, Ingresso_2, Risultato) :- verifica_degli_ingressi(Ingresso_1,
Ingresso_2), computazione(Ingresso_1, Ingresso_2, Risultato).

verifica_degli_ingressi(Ingresso_1, Ingresso_2), atom(Ingresso_1), atom(Ingresso_2), !.

verifica_degli_ingressi(_, _) :- nl, write('Errore.'), nl.
```

```
computazione(Ingresso_1, Ingresso_2, Risultato) :- elaborazione(Ingresso_1, Parziale_1),
elaborazione(Ingresso_2, Parziale_2), raccolta_parziali(Parziale_1, Parziale_2, Risultato).
```

È conveniente che i simboli "/*" e "*/" relativi all'inizio ed alla fine di un commento compaiano sempre in corrispondenza ai primi due caratteri delle rispettive righe, in maniera da consentire una visibilità immediata dell'estensione del commento. Le teste delle clausole (o le intere clausole, se sono asserzioni) possono essere scritte a partire dalla terza posizione della riga, tutte allineate fra loro. Le mete del corpo di ogni clausola vanno allineate più all'interno rispetto alla metà di testa. È consigliabile non disporre più di due o tre mete su una stessa riga, ed in ogni caso vanno poste su righe diverse invocazioni di procedura che si riferiscono a funzionalità tra loro logicamente diverse. Per esempio è preferibile scrivere:

```
meta :- ingresso_1, ingresso_2,  
computazione,
```

uscita_1, uscita_2, uscita_3.

anziché:

meta :- ingresso_1, ingresso_2, computazione,

uscita_1, uscita_2, uscita_3.

o, peggio ancora:

meta :- ingresso_1,

ingresso_2,

computazione,

uscita_1,

uscita_2,

uscita_3.

I tagli possono essere posti come ultima meta di una riga se non alterano il contenuto dichiarativo della procedura; se ciò accade, invece, possono figurare come unica sottometa della riga per evidenziare tale effetto della loro presenza. Le procedure vanno presentate avendo cura di ordinarle da quelle di livello più alto a quelle destinate alle funzionalità ausiliarie. Tutte le clausole relative ad una procedura vanno raggruppate consecutivamente, separate mediante una riga vuota dalle clausole della procedura successiva;

Eventuali procedure ausiliarie i cui parametri siano un sovrainsieme dei parametri della procedura chiamante possono avere lo stesso nome di predicato di quest'ultima seguito dai suffissi "**_1**", "**_2**", e così via; questo accorgimento diminuisce la probabilità di introdurre altrove nel programma ulteriori procedure con lo stesso nome di quelle ausiliarie. Tali suffissi possono anche essere impiegati per diverse versioni di procedure che realizzano una stessa funzionalità.

Le dichiarazioni di operatori possono essere poste all'inizio del programma, a meno che una certa dichiarazione non ne sostituisca un'altra relativa allo stesso operatore, nel qual caso comparirà nel punto in cui la precedente cessa di valere; alternativamente, possono essere localizzate nei punti che precedono le clausole nelle quali vengono usate.

Riusabilità e prototipazione.

La modularità del programma non è utile solo per una sua efficace documentazione: essa favorisce anche la riusabilità di parti di programma (procedure e/o moduli che contengono più procedure). Diversi aspetti sono coinvolti nella possibilità di riutilizzare moduli: alcuni dipendono dal programmatore, altri dal linguaggio. Nel seguito si discutono tali aspetti, e gli accorgimenti che è possibile prendere per facilitare la riusabilità di procedure Prolog.

Per essere convenientemente usabile in un programma in fase di realizzazione, un modulo già esistente deve essere di facile interpretazione, e facilmente inseribile nel contesto in cui serve. Ciò comporta che deve essere possibile comprendere la funzionalità del modulo dall'esterno, senza

leggerne il codice, e che non devono esserci ambiguità sul significato sia dichiarativo che procedurale e sulle interfacce (i parametri).

Entrambi i suddetti aspetti sono in relazione con il modo in cui il programma è documentato: lo schema di documentazione proposto precedentemente ne tiene conto. Ad esempio, l'indicazione di quali predicati di sistema un modulo usa è utile in vista della portabilità: secondo la versione di Prolog disponibile, il modulo potrà essere utilizzato così come, se risulta che i predicati predefiniti che utilizza sono implementati anche in quella particolare versione, oppure andrà modificato, o - se possibile - gli stessi predicati dovranno essere definiti.

Queste considerazioni riguardano l'utilizzo di una biblioteca (library) di moduli, una volta che questa sia disponibile. Un aspetto complementare è quello relativo alla realizzazione di una tale biblioteca di moduli da riusare. Oltre agli aspetti di documentazione già indicati, la questione principale in questo caso è quella della scelta di quali moduli inserire in biblioteca. Una tale scelta deve bilanciare due fattori contrastanti: se da un lato la presenza di molte funzionalità aumenta virtualmente la possibilità di trovare quella che serve, dall'altro lato rende meno agevole il suo reperimento effettivo. Alcuni criteri pratici sono i seguenti:

- la biblioteca può essere suddivisa in sezioni, secondo le varie classi di funzionalità;
- le funzionalità presenti in ogni sezione non devono essere né troppo specializzate (altrimenti diminuisce la probabilità che risultino utili) né troppo elementari (in tal caso diminuisce la loro utilità, ossia il risparmio ottenuto rispetto al costruirle);
- le varianti di una stessa funzionalità devono essere, nella documentazione, chiaramente differenziate rispetto all'uso; se sono possibili varianti che usano differenti tecniche implementative, ma le cui implicazioni rispetto all'utilizzo non sono chiaramente definibili, una sola di esse sarà presente in biblioteca;
- poiché l'utente potrà utilizzare i moduli non solo inserendoli direttamente nel suo programma, ma anche componendoli egli stesso nelle funzionalità più complesse desiderate, i moduli della biblioteca che sono già composizioni di procedure più elementari singolarmente presenti in biblioteca (in quanto le richiamano) devono essere composizioni significative, cioè non ricavabili in modo immediato ed evidente dalle procedure componenti.

È da osservare che in generale la realizzazione di una funzione mediante composizione di procedure già definite può risultare meno efficiente di quella ottenibile con un'implementazione apposita. Per un esempio molto semplice di questo fatto, si rivedano le due realizzazioni della relazione inversione di liste. La prima, che utilizza la procedura concatenazione, è meno efficiente della seconda, realizzata appositamente senza far ricorso ad altre relazioni di uso generale.

Tenendo conto delle suddette considerazioni, si può attuare un processo di sviluppo di un programma logico procedendo in due fasi distinte. In un primo momento ci si occupa prevalentemente, se non esclusivamente, della funzionalità del sistema in sviluppo. In questa fase è conveniente utilizzare il più possibile i moduli già esistenti in biblioteca per ottenere rapidamente (in virtù del corrispondente risparmio di tempo) un prototipo del programma. I moduli vengono selezionati essenzialmente in base al loro contenuto dichiarativo; il contenuto procedurale può essere tenuto presente per verificarne l'adeguatezza al contesto. Anche la composizione di moduli viene eseguita da un punto di vista funzionale, senza particolare riguardo all'efficienza.

Una volta terminato e verificato complessivamente il prototipo così ottenuto, particolarmente per quanto concerne la sua aderenza ai requisiti funzionali desiderati, si può procedere ad una seconda fase, il cui obiettivo principale è quello di migliorare l'efficienza del programma. In questa fase il

contenuto procedurale dei moduli servirà non solo all'ottimizzazione delle singole procedure, ma soprattutto ad un'ottimizzazione globale che può anche comportare modifiche relative alla loro composizione.

È da notare che i due punti di vista considerati - quello di usare una biblioteca di moduli e quello di costruirla - non solamente non sono in contrasto tra loro, ma anzi possono essere utilmente compresenti. Chi sviluppa un programma può man mano costruirsi la propria biblioteca di moduli che userà in più parti del programma in fase di elaborazione; oppure, partendo da una biblioteca preesistente, durante lo sviluppo del proprio programma può sia utilizzarla che espanderla e migliorarla. La possibilità di utilizzare moduli è praticabile nei sistemi Prolog che rendono disponibile il costrutto di modulo, ma non è supportata dal normale meccanismo di consultazione previsto dal linguaggio (predicato predefinito **consult**). Esso consente di distribuire i moduli in files diversi (eventualmente usando come nome del file lo stesso nome del modulo) e di richiamarli nella base di dati quando occorre, ma non mantiene la modularità, in quanto il contenuto di un file, una volta immesso nella base di dati, non conserva memoria dell'appartenenza ad uno specifico modulo.

Ciò comporta ad esempio che, se in due moduli consultati da due files diversi vi sono procedure comuni, queste risultano duplicate nella base di dati. Viceversa, non è possibile un aggiornamento selettivo della base di dati, in quanto la riconsultazione (predicato predefinito **reconsult**) cambia tutte le clausole corrispondenti a quella selezionata, senza riguardo al modulo di provenienza, con le conseguenze che da ciò derivano nell'esecuzione del programma in termini di alterazione del flusso di controllo e dunque di esito della computazione.

Supponiamo per esempio di avere in un file un modulo contenente le procedure **concatenazione** ed **appartenenza**. In un altro file vi sia il seguente modulo per la rappresentazione di un insieme mediante liste:

```
/*
```

MODULO insiemi.

ESPORTA: sottoinsieme, cancellazione.

IMPORTA: appartenenza (da liste).

```
*/
```

```
sottoinsieme([ ], _).
```

```
sottoinsieme([ T | C], I) :- appartenenza(T, I), sottoinsieme(C, I).
```

```
cancellazione(_, [], []).
```

```
cancellazione(E, [E | C], C).
```

```
cancellazione(E, [T | C1], [T | C2]) :- E\== T, cancellazione(E, C1, C2).
```

Un terzo file contenga il seguente modulo per la realizzazione di code:

```
/*
```

MODULO code.

ESPORTA: aggiunta_elemento, cancellazione.

IMPORTA: concatenazione (da liste).

*/

aggiunta_elemento(X, Q1, Q2) :- concatenazione(Q1, [X], Q2).

cancellazione(_, [], []).

cancellazione(E, [E | C], L):- cancellazione(E, C, L).

cancellazione(E, [T | C1], [T | C2]) :- E\== T, cancellazione(E, C1, C2).

La definizione della procedura cancellazione è diversa nei due moduli perché nel primo si sfrutta la conoscenza che negli insiemi gli elementi sono tutti distinti (quindi la prima procedura cancella una sola occorrenza dell'elemento), mentre nel secondo questa ipotesi non vale (perciò la seconda procedura cancella ogni occorrenza dell'elemento). La consultazione dei tre files per l'uso dei tre moduli nello stesso programma comporta la compresenza nella base di dati delle clausole relative alle due differenti definizioni di cancellazione, ciò che durante l'esecuzione può dar luogo a comportamenti diversi da quelli desiderati.

Note bibliografiche.

Bruynooghe (1982b) ha studiato per primo il problema dello sviluppo della documentazione e delle possibili ridondanze in un programma Prolog.

L'idea di utilizzare le dichiarazioni di modo come documentazione, ed il suggerimento di posizionare i tagli su righe separate, compaiono in O'Keefe (1983). Alcuni aspetti relativi alla realizzazione di biblioteche di moduli in Prolog vengono esaminati in Feuer (1983) ed in Furukawa, Nakajima e Yonezawa (1983). Una breve raccolta di procedure Prolog di utilità generale è stata proposta da Dundy e Welham (1977). Sulla modularità in MProlog si veda Szeregi (1982). La nozione ed il costrutto di modulo sono anche presenti in micro-Prolog, ed illustrati in Clark e McCabe (1984). Aspetti d'uso di Prolog per lo sviluppo prototipale di software sono discussi per esempio in Venken e Bruynooghe (1984).

Sommario.

Lo studente ha potuto considerare in questo capitolo alcuni fra i diversi aspetti della problematica di sviluppo di un programma Prolog, particolarmente riguardo alla sua documentazione e modularizzazione.

Questi aspetti diventano tanto più rilevanti quanto più articolato e complesso è il programma da costruire. Essi vengono esemplificati nel seguito del corso, dove si procede con esempi più ampi; ai casi più significativi è applicato lo schema di documentazione proposto, in modo flessibile rispetto alle esigenze di spazio.

9. Strutture di dati e programmi

Dove si espongono le possibilità di rappresentazione in Prolog delle principali strutture logiche di dati classiche della programmazione: sequenze, pile, code, insiemi, matrici, alberi e grafi.

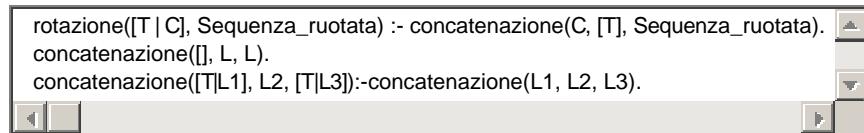
E dove si illustrano procedure tipiche per la loro elaborazione (come inserimenti, cancellazioni e ordinamenti), cominciando ad applicare lo [schema di documentazione](#) esposto. Si è visto nella [Strutturazione dei dati](#) come i dati possono essere rappresentati in Prolog estensionalmente (con relazioni) o intensionalmente (con termini), e si sono discusse le conseguenze di queste diverse possibilità riguardo alle procedure di accesso ed elaborazione. In questo capitolo, la rappresentazione estensionale è impiegata solo nel caso dei grafi, mentre le altre strutture logiche di dati più ricorrenti nella programmazione tradizionale vengono rappresentate mediante termini, definendo poi opportune procedure per realizzare le modalità di elaborazione più tipiche di ciascuna di esse.

Rappresentando i dati mediante termini, si possono utilizzare liste o altre strutture. L'uso delle liste risulta più conveniente quando la struttura di dati è una sequenza di elementi, indipendentemente dalla loro natura, o comunque una collezione di oggetti da sottoporre a scansione o ad ordinamento. Con opportuni accorgimenti, le liste possono essere impiegate anche per gli insiemi e le loro operazioni. Nel caso di alberi è invece più adeguato l'utilizzo di termini appositamente strutturati.

Sequenze.

Una sequenza, cioè una collezione ordinata di elementi omogenei (non necessariamente distinti), è rappresentata da una lista in modo naturale. Le procedure seguenti, definite in termini di sequenze, possono quindi essere viste anche come ulteriori esempi di operazioni su liste.

L'accesso diretto agli elementi di una sequenza, in base alla loro posizione, è assicurato dalla procedura [ennesimo](#). La procedura [concatenazione](#) è alla base di diverse operazioni su sequenze; un esempio di sua applicazione è dato dalla relazione **rotazione(S1, S2)**, soddisfatta se la sequenza **S2** si ottiene dalla sequenza **S1** scambiando il suo primo elemento con l'ultimo, ossia "ruotandola" di una posizione verso sinistra:



```
rotazione([T | C], Sequenza_ruotata) :- concatenazione(C, [T], Sequenza_ruotata).
concatenazione([], L, L).
concatenazione([T|L1], L2, [T|L3]) :- concatenazione(L1, L2, L3).
```

Il quesito:

?- rotazione([a,b,c,d,e],S), rotazione(S,[c,d,e,a,b]).

termina con successo, istanziando **S** a **[b, e, d, e, a]**.

La relazione di adiacenza fra due elementi di una sequenza è espressa dalla procedura seguente:

```
/*
```

PROCEDURA: adiacenti(E1, E2, S).

D: gli elementi E1 ed E2 della sequenza S sono tra loro adiacenti.

P1: adiacenti(<, <, <): verifica la relazione.

P2: adiacenti(>, <, <): ricerca in una sequenza l'elemento che precede un elemento assegnato, per tutte le occorrenze di quest'ultimo, tramite ritorno indietro.

P3: adiacenti(<, >, <): ricerca in una sequenza l'elemento che segue un elemento assegnato, per tutte le occorrenze di quest'ultimo, tramite ritorno indietro.

P4: adiacenti(>, >, <): genera, tramite ritorno indietro, le coppie di elementi fra loro adiacenti.

C: Si noti come, in questo caso, la condizione limite sia espressa per sequenze di due o più elementi.

```
*/
```

```
adiacenti(E1, E2, [E1,E2|_]).
```

```
adiacenti(E1, E2, [_|Coda]) :- adiacenti(E1, E2, Coda).
```

Per la sostituzione di un elemento di una sequenza si può usare la procedura:

```
/*  
PROCEDURA: sostituzione(E1, S1, E2, S2).  
D: la sequenza S2 è uguale alla sequenza S1 con ogni occorrenza dell'elemento E2 in S2 al posto dell'elemento E1 in S1.  
P1: sostituzione(<, <, <, <): verifica la relazione.  
P2: sostituzione(<, <, <, >): fornisce S2 con E2 sostituito ad ogni occorrenza di E1 in S1.  
P3: sostituzione(<, >, <, <): fornisce S1 con E1 sostituito ad ogni occorrenza di E2 in S2.  
T: rosso, per rendere mutuamente esclusive le due clausole ricorsive.  
*/  
sostituzione(_, [], _, []).  
sostituzione(E1, [E1|S1], E2, [E2|S2]) :- !, sostituzione(E1, S1, E2, S2).  
sostituzione(E1, [E|S1], E2, [E|S2]) :- sostituzione(E1, S1, E2, S2).
```

Un'estensione dei programmi di cancellazione di elementi è costituita da procedure che sottraggono una sottosequenza da una sequenza, come la seguente, che sottrae una sottosequenza iniziale (prefisso):

```

/*
PROCEDURA: sottrazione_prefisso(S1, S2, S3).
D: S3 è la sequenza S2 privata del prefisso S1.
P1: sottrazione_prefisso(<, <, <): verifica la relazione.
P2: sottrazione_prefisso(<, <, >): ricerca la sequenza risultante dalla sottrazione della sequenza prefisso alla sequenza in ingresso.
P3: sottrazione_prefisso(>, <, <): ricerca la sequenza prefisso.
C: fallisce se la sequenza a primo argomento non è un prefisso della sequenza a secondo argomento.
*/
sottrazione_prefisso([], S, S).
sottrazione_prefisso([Testa|Coda_prefisso], [Testa|Coda_ingresso], S) :- sottrazione_prefisso(Coda_prefisso, Coda_ingresso, S).

```

La seguente procedura risolve il problema della fusione di sequenze di interi ordinate in senso ascendente:

```

/*
PROCEDURA: fusione(S1, S2, S3).
D: S3 è la sequenza risultante dalla fusione ordinata delle sequenze S1 e S2 ordinate in senso crescente.
P1: fusione(<, <, <): verifica la relazione.
P2: fusione(<, <, >): genera la sequenza risultante dalla fusione.
T: verdi, per eliminare la ridondanza dovuta alla necessaria presenza di due condizioni limite. Senza tagli, invece, tale ridondanza può essere eliminata sostituendo la prima condizione limite con fusione([], [T|C], [T|C]).
*/
fusione([], Sequenza, Sequenza) :- !.
fusione(Sequenza, [], Sequenza) :- !.
fusione([T1|C1], [T2|C2], [T1|C]) :- T1 < T2, fusione(C1, [T2|C2], C).
fusione([T1|C1], [T2|C2], [T2|C]) :- T1 >= T2, fusione([T1|C1], C2, C).

```

Una sequenza di elementi su cui è definita una relazione d'ordine (numeri, per esempio) viene detta unimodale se consiste di una sottosequenza ascendente seguita da una sottosequenza discendente, presentando così un singolo picco. Sono compresi i casi particolari in cui una delle due sottosequenze, od entrambe, sono vuote o di un solo elemento. La procedura più direttamente corrispondente alla definizione del problema è la seguente:

```

unimodale(Lista) :- concatenazione(L1, L2, Lista), ascendente(L1), discendente(L2).
ascendente([]).
ascendente([_]).
ascendente([X, Y | Z]) :- X < Y, ascendente([Y | Z]).
discendente([]).
discendente([_]).
discendente([X, Y | Z]) :- X > Y, discendente([Y | Z]).
concatenazione([], L, L).
concatenazione([T | L1], L2, [T | L3]) :- concatenazione(L1, L2, L3).

```

Essa è caratterizzata dai diversi tentativi da parte di concatenazione, mediante ritorno indietro, di separare la sequenza in ingresso in due parti, la prima ascendente e la seconda discendente. Ciò comporta due fonti di inefficienza: innanzitutto, quando la sequenza non è unimodale, la procedura non è in grado di riconoscerlo sino a che non ha tentato tutte le decomposizioni possibili; in secondo luogo, si ha un completo riesame delle sottosequenze componenti.

Nasce dunque la necessità di sostituire la procedura formulata con un'altra versione che fallisca più efficientemente nel caso in cui non vi sia soluzione. In generale è difficile superare il problema con

un semplice miglioramento della strategia di controllo; si impone, invece, il cambiamento della logica, per esempio con il seguente programma:

```
unimodale_1([ ]).  
unimodale_1([_|_]).  
unimodale_1([X, Y | Z]) :- X < Y, unimodale_1([Y | Z]).  
unirnodale_1([X, Y | Z]) :- X > Y, discendente([Y | Z]).  
discendente([_|_]).  
discendente([_|_]).  
discendente([X, Y | Z]) :- X > Y, discendente([Y | Z]).
```

in cui si ha il passaggio del controllo a **discendente** non appena è stata trovata una coppia discendente. Se la metà non può essere soddisfatta, l'esecuzione termina non appena viene trovata una coppia non ordinata, senza proseguire con la ricerca di altre separazioni: la procedura non dà mai luogo a computazioni inutili (si noti anche l'assenza di chiamate a **concatenazione** è ad **ascendente**). Un'altra possibilità è costituita dall'utilizzo di un interruttore (switch) il cui scopo è l'esplicita determinazione del cammino di controllo:

```
unimodale_2(L) :- unimodale_3(L, _).  
unimodale_3([], _).  
unimodale_3([_|_], _).  
unimodale_3([X, Y | Z], su) :- X < Y, unimodale_3([Y | Z], su).  
unimodale_3([X, Y | Z], giù) :- X > Y, unimodale_3([Y | Z], giù).
```

Alla prima chiamata il secondo argomento, che rappresenta l'interruttore, è libero, mentre nelle successive è istanziato ad una costante che serve come stato per determinare quale clausola applicare. Nell'ipotesi di avere assegnato in ingresso una sequenza con un picco in uno dei suoi elementi centrali, durante la prima fase l'interruttore si trova nello stato **su**, cosicché la scansione in avanti della sequenza viene compiuta dalla prima clausola ricorsiva; quando viene incontrata la prima coppia discendente di numeri, l'interruttore viene disistanziato e reistanziato a **giù**, e viene attivata, per il completamento della scansione, la seconda clausola ricorsiva. Si noti la compattezza del programma, dovuta all'assenza di procedure specifiche volte a verificare se una sottosequenza è ascendente o discendente.

In generale l'utilizzo di interruttori favorisce la diminuzione del numero di procedure di un programma, ma lo rende limitato alle particolari sequenze di controllo previste.

Pile e code.

La caratteristica precipua della struttura di dati detta pila (stack) è di costituire una sequenza di elementi nella quale le operazioni di inserimento di un nuovo elemento (push) e di estrazione di un elemento (pop) avvengono alla stessa estremità, cosicché il primo elemento estratto è sempre l'ultimo inserito; tale elemento costituisce la cima (top) della pila e può essere reso disponibile senza eliminarlo, mediante un'operazione chiamata appunto top. Questa caratteristica della pila, congiunta con quella di multidirezionalità delle procedure Prolog, consente di usare un'unica,

semplice procedura per realizzare tutte e tre le tipiche operazioni di accesso ad una pila (rappresentata mediante una lista):

```
/*
PROCEDURA: pila(E, L1, L2).
D: la lista L2 è uguale alla concatenazione dell'elemento E con la lista L1.
P1: pila(<, <, >): come operazione push che inserisce l'elemento a primo argomento nella pila rappresentata dalla lista a secondo argomento, fornendo la pila modificata nel terzo argomento.
P2: pila(>, >, <): come operazione pop che estrae dalla pila data come lista a terzo argomento l'elemento in cima, fornendo nel secondo argomento la pila modificata e nel primo l'elemento estratto. Fallisce se la pila (lista) in ingresso è vuota. Ammette come casi particolari i due usi seguenti.
P2a: pila(_, >, <): come operazione pop che non rende disponibile l'elemento estratto.
P2b: pila(>, _, <): come operazione top che rende osservabile la cima (fornita nel primo argomento) ma non modifica la pila (se si usa ancora il terzo argomento per le operazioni successive).
*/
pila(Cima, Pila, [Cima | Pila]).
```

Ad esempio, il seguente quesito:

?- pila(a, [b, c], S), pila(E, S1, S), pila(E1, _, S1).

fornisce la risposta:

S = [a, b, c]

E = a

S1 = [b, c]

E1 = b

in cui la prima riga è causata dalla prima sottometta che opera come push, la seconda e la terza sono effetto della seconda sottometta che agisce come pop, e l'ultima è dovuta alla terza sottometta utilizzata come top. Le liste possono essere facilmente impiegate anche per rappresentare la struttura di dati coda (queue). La procedura di **inserimento** di un elemento fornito in ingresso, come primo elemento della lista (l'ultimo della coda da essa rappresentata), è molto semplice; Invece per la procedura che estrae e rende disponibile l'ultimo elemento della lista in ingresso (il primo inserito nella coda da essa rappresentata) è necessaria una ricorsione:

```
inserimento(Elemento, Lista_ingresso, [Elemento | Lista_ingresso]).
/*
PROCEDURA: estrazione(E, L1, L2).
D: la lista L2 è uguale alla lista L1 meno il suo ultimo elemento E.
P1: estrazione(<, <, <): verifica la relazione.
P2: estrazione(>, <, >): fornisce l'ultimo elemento della lista in ingresso e la nuova lista.
C: fallisce se la lista in ingresso è vuota.
*/
estrazione(El, [El], []).
estrazione(El, [T | Coda_ngr], [T | Coda_uscita]) :- estrazione(El, Coda_ngr, Coda_uscita).
```

Come esempio dell'effetto delle due procedure, il quesito:

?- inserimento(a, [], L1), inserimento(b, L1, L2), inserimento(c, L2, L3), estrazione(E, L3, L).

ha come risposta (trascurando **L1**, **L2**, **L3**):

E = a

L = [c, b]

Insiemi.

In Prolog gli insiemi possono essere rappresentati mediante liste, presupponendo che queste non contengano elementi duplicati; in caso contrario le procedure che realizzano le usuali operazioni sugli insiemi, definite in questo paragrafo, possono presentare anomalie di comportamento. La procedura seguente può essere usata per eliminare i duplicati da liste che eventualmente ne contengano, rendendo disponibili liste accettabili in ingresso dalle procedure per la manipolazione di insiemi:

```
/*
PROCEDURA: setaccio(L1, L2).
D: L2 è la lista L1 priva di tutte e sole le occorrenze degli eventuali elementi duplicati.
P1: setaccio(<, <): verifica la relazione.
P2: setaccio(<, >): ricerca la lista che si ottiene cancellando da L1 tutte le occorrenze degli elementi duplicati.
U: cancellazione_1/3.
*/
setaccio([ ], [ ]).
/* Se L1 è una lista non vuota la cui testa è T, L2 deve cominciare anch'essa con T, ma la sua coda dev'essere una versione setacciata della coda di L1, dopo che tutte le ulteriori occorrenze di T sono state cancellate */
setaccio([T | C1], [T | C2]) :- cancellazione_1(T, C1, C), setaccio(C, C2).
cancellazione_1(_, [], []).
cancellazione_1(E, [E | C], L) :- cancellazione_1(E, C, L).
cancellazione_1(E, [T | C1], [T | C2]) :- E\== T, cancellazione_1(E, C1, C2).
```

S'intende che un insieme può essere vuoto, e può essere un insieme semplice od un insieme di insiemi (a qualunque livello di innestamento); pertanto, dove sono menzionati un elemento ed un insieme, essi sono da intendere come elemento semplice ed insieme semplice, oppure insieme ed insieme di insiemi, e così via (a qualunque livello di innestamento).

Rappresentando un insieme con una lista, si possono usare le procedure [lunghezza](#) ed [appartenenza](#). La relazione **lunghezza(I, N)** può qui essere interpretata come cardinalità **N** dell'insieme **I**. Utilizzandola, si possono ad esempio estrarre da un insieme di insiemi i soli singoletti (insiemi di un solo elemento):

```

/*
PROCEDURA: singoletti(I1, I2).
D: I2 è l'insieme dei singoletti dell'insieme I1.
P1: singoletti(<, <): verifica la relazione.
P2: singoletti(<, >): ricerca i singoletti.
T: rosso, per simulare la struttura "if-then-else".
U: lunghezza/2
*/
singoletti([], []).
singoletti([T|C1], [T|C2]) :- lunghezza(T, 1), !, singoletti(C1, C2).
singoletti([_|C1], C2) :- singoletti(C1, C2).
lunghezza([], 0).
lunghezza([_|C], N):-lunghezza(C, N1), N is N1 + 1.

```

La procedura [appartenenza](#) di un elemento ad una lista mette in relazione quest'ultima con i suoi elementi, ma non consente di accedere agli elementi di eventuali sottoliste che possono trovarsi entro la lista data. Di conseguenza, il quesito:

?- appartenenza(3, [1,2, [3], 4]).

ottiene risposta negativa, mentre termina con successo il quesito:

?- appartenenza([3], [1,2, [3],4]).

La seguente procedura consente di raggiungere elementi di sottoliste di una lista data ad un arbitrario livello di innestamento. La relazione **contenuto_in(X, L)** è soddisfatta se X è un elemento di L o di una sottolista di L, o di una sottolista di una sottolista di L, e così via:

```

contenuto_in(X, L) :- appartenenza(X, L).
contenuto_in(X, L) :- appartenenza(Y, L), contenuto_in(X, Y).
appartenenza(E, [E|_]).
appartenenza(E, [_|C]):-appartenenza(E, C).

```

Una versione che non fa uso di procedure ausiliarie, e fornisce mediante ritorno indietro tutti gli elementi di una stessa sottolista sino al livello più elementare, è invece:

```

contenuto_in_1(T, [T | _]).
contenuto_in_1(T1, [[T2 | C] | _]) :- contenuto_in_1(T1, [T2 | C]).
contenuto_in_1(T, [_ | C]) :- contenuto_in_1(T, C).

```

che estende la procedura appartenenza con una terza clausola preposta a gestire il caso in cui il primo elemento della lista è a sua volta una lista di almeno un elemento.

Si possono sperimentare le due versioni della procedura formulando ad esempio il quesito:

```
?- contenuto_in(X, [[1,2],[4],5,[ ],[6,7,[8,9]],10]).
```

e il quesito:

```
?- contenuto_in_1(X, [[1,2],[4],5,[ ],[6,7,[8,9]],10]).
```

A partire dalla relazione [appartenenza](#) si può definire la relazione **sottoinsieme(S1, S2)**, da usare con entrambi gli argomenti istanziati, vera se **S1** è un sottoinsieme (proprio o improprio) di **S2**:

```
sottoinsieme([ ], _).  
sottoinsieme([Elemento | Elementi], Insieme) :- appartenessa(Elemento, Insieme), sottoinsieme(Elementi, Insieme).  
appartenessa(El, [El|_]).  
appartenessa(El, [_|C]) :- appartenessa(El, C).
```

Usando **sottoinsieme** si possono realizzare altre procedure per confrontare insiemi, come la seguente, utilizzabile con i primi due argomenti istanziati ed il terzo libero (i tagli realizzano due strutture di selezione innestate):

```
confronto_insiemi(I1, I2, Relazione) :- sottoinsieme(I1, I2), !, confronto_insiemi_1(I1, I2, Relazione). confronto_insiemi(I1, I2, Relazione) :- sottoinsieme(I2, I1), !, confronto_insiemi(_ , _, inconfrontabili).  
confronto_insiemi_1(I1, I2, uguali) :- sottoinsieme(I2, I1), !.  
confronto_insiemi_1(_ , _, sottoinsieme).  
sottoinsieme([ ], _).  
sottoinsieme([Elemento | Elementi], Insieme) :- appartenessa(Elemento, Insieme), sottoinsieme(Elementi, Insieme).  
appartenessa(El, [El|_]).  
appartenessa(El, [_|C]) :- appartenessa(El, C).
```

Si noti che, ad esempio, il quesito:

```
?- confronto_insiemi([a, b], [b, a], R).
```

fornisce la risposta:

R = uguali

che è corretta per gli insiemi, ma non lo sarebbe per le sequenze. La procedura seguente consente di stabilire se due insiemi sono tra loro disgiunti (il taglio simula la struttura "if-then-else"):

```
disgiunti([], _).  
disgiunti(_ , []).  
disgiunti([Testa | _], Insieme) :- appartenessa(Testa, Insieme), !, fail.  
disgiunti([_|Coda], Insieme) :- disgiunti(Coda, Insieme).  
appartenessa(El, [El|_]).  
appartenessa(El, [_|C]) :- appartenessa(El, C).
```

Per l'unione e l'intersezione di due insiemi si possono utilizzare le seguenti procedure, con i primi due argomenti istanziati ed il terzo libero:

```

union([ ], X, X).
union([T | C], I, Unione) :- appartenenza(T, I), !, union(C, I, Unione).
union([T | C], I, [T | C_union]) :- union(C, I, C_union).
appartenenza(El, [El|_]).
appartenenza(El, [_|C]) :- appartenenza(El, C).

```

```

intersezione([ ], _, [ ]).
intersezione([T | C1], I, [T | C2]) :- appartenenza(T, I), !, intersezione(C1, I, C2).
intersezione([_|C], I1, I2) :- intersezione(C, I1, I2).
appartenenza(El, [El|_]).
appartenenza(El, [_|C]) :- appartenenza(El, C).

```

La relazione **sottoinsieme** può essere espressa mediante **intersezione** osservando che **I1** è un sottoinsieme di **I2** se **I1** è l'intersezione di **I1** ed **I2**:

```

sottoinsieme_1(I1, I2) :- intersezione(I1, I2, I1).
intersezione([ ], _, [ ]).
intersezione([T | C1], I, [T | C2]) :- appartenenza(T, I), !, intersezione(C1, I, C2).
intersezione([_|C], I1, I2) :- intersezione(C, I1, I2).
appartenenza(El, [El|_]).
appartenenza(El, [_|C]) :- appartenenza(El, C).

```

Infine, la procedura seguente fornisce nel terzo argomento l'insieme differenza tra l'insieme dato a primo argomento e quello a secondo argomento:

```

differenza(A, [ ], A).
differenza(A, [T_B | C_B], I) :- cancellazione(T_B, A, I1), differenza(I1, C_B, I).
cancellazione(_, [], []).
cancellazione(E, [E|C], C).
cancellazione(E, [T|C1], [T|C2]) :- E == T, cancellazione(E, C1, C2).

```

Si noti che la ricorsione è effettuata, per maggiore efficienza, sulla lista da sottrarre e non su quella da sotoporre a sottrazione, lasciando alla procedura **cancellazione** la gestione del caso in cui la lista da sottrarre ha tutti (o alcuni) elementi che non compaiono nella lista da sotoporre a sottrazione: se si usa la procedura cancellazione a tre clausole, quest'ultima viene data in uscita invariata (o senza gli elementi comuni), mentre con quella a due clausole differenza fallisce. A partire dalle procedure precedenti si possono facilmente definire altre operazioni su insiemi, come la procedura:

```

sottrazione([I1, I2, In1, In2]) :- intersezione(I1, I2, I), differenza(I1, I, In1), differenza(I2, I, In2).
intersezione([], _, []).
intersezione([T | C1], I, [T | C2]) :- appartenenza(T, I), !, intersezione(C1, I, C2).
intersezione([_|C], I1, I2) :- intersezione(C, I1, I2).
appartenenza(E, [E|_]).
appartenenza(E, [_|C]) :- appartenenza(E, C).
differenza(A, [], A).
differenza(A, [T_B | C_B], I) :- cancellazione(T_B, A, I1), differenza(I1, C_B, I).
cancellazione(_, [], []).
cancellazione(E, [E|C], C).
cancellazione(E, [T|C1], [T|C2]) :- E == T, cancellazione(E, C1, C2).

```

che fornisce a terzo e quarto argomento gli insiemi di primo e secondo argomento privati degli eventuali elementi comuni; ad esempio, con il quesito:

?- sottrazione([a,b], [b,c], X, Y).

si ottiene la risposta:

X = [a]

Y = [c]

Matrici.

Una maniera naturale ed immediata per rappresentare una matrice in Prolog consiste nell'associare ad ogni riga della matrice una lista dei suoi elementi, e nel raggruppare le liste così ottenute entro una lista globale: una matrice è così una lista di righe, ed ogni sottolista della lista che rappresenta la matrice risulta ordinatamente associata ad una sua riga.

Come primo esempio di una procedura che opera su matrici è naturale considerare il problema dell'accesso ad un suo elemento. L'invocazione della procedura seguente termina con successo se **Ei** è l'elemento di posto (**I, J**), **I**-esima riga e **J**-esima colonna, della matrice **Mat**:

```

elemento_matrice(Mat, I, J, El) :- ennesimo(I, Mat, Riga), ennesimo(J, Riga, El).
ennesimo(1, [El|_], El).
ennesimo(N, [_|Coda], El) :- N1 is N - 1, ennesimo(N1, Coda, El).

```

È un tipico caso di trasmissione sequenziale dei parametri: la lista **Riga**, ottenuta in uscita dalla prima chiamata della procedura **ennesimo**, viene passata in ingresso alla seconda chiamata, che fornisce l'elemento **El** desiderato. Utilizzando la rappresentazione di matrici mediante liste è possibile sviluppare con facilità procedure per la loro analisi, come la seguente:

```

/*
PROCEDURA: conteggio_elementi_matrice(M, X, Y, Z).
D: X, Y e Z sono rispettivamente il numero di elementi positivi, nulli e negativi presenti nella matrice numerica M.
P: conteggio_elementi_matrice(<, >, >, >): fornisce i numeri X, Y e Z.
*/
/* inizializza contatori totali */
conteggio_elementi_matrice(Matrice, X, Y, Z) :- conteggio_1(Matrice, X, 0, Y, 0, Z, 0).

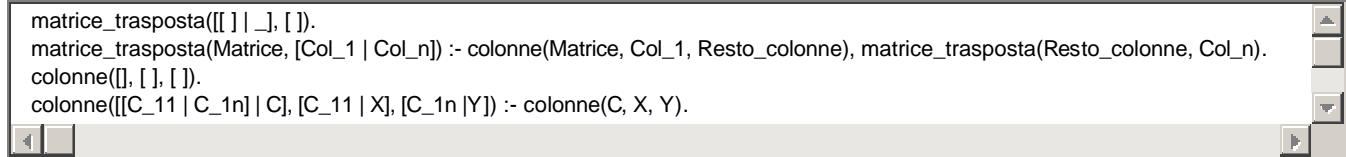
```

```

/* i contatori totali assumono il valore finale */
conteggio_1([], X, X, Y, Y, Z, Z).
/* calcola il valore dei contatori parziali relativi alla riga corrente e aggiorna i contatori totali */
conteggio_1([Riga | Righe], X, X1, Y, Y1, Z, Z1) :- conteggio_2(Riga, X2, Y2, Z2), X3 is X1+X2, Y3 is Y1+Y2, Z3 is Z1+Z2,
conteggio_1(Righe, X, X3, Y, Y3, Z, Z3).
/* inizializza i contatori parziali */
conteggio_2(Riga, X, Y, Z) :- conteggio_3(Riga, X, 0, Y, 0, Z, 0).
/* I contatori parziali assumono il valore finale */
conteggio_3([], X, X, Y, Y, Z, Z).
/* aggiorna i contatori parziali di riga */
conteggio_3([El | Elementi], X, X1, Y, Y1, Z, Z1) :- El>0, X2 is X1 + 1, conteggio_3(Elementi, X, X2, Y, Y1, Z, Z1).
conteggio_3([0 | Elementi], X, X1, Y, Y1, Z, Z1) :- Y2 is Y1+1, conteggio_3(Elementi, X, X1, Y, Y2, Z, Z1).
conteggio_3([El | Elementi], X, X1, Y, Y1, Z, Z1) :- El<0, Z2 is Z1 + 1, conteggio_3(Elementi, X, X1, Y, Y1, Z, Z2).

```

Per calcolare la trasposta di una matrice è sufficiente trasformarla da lista delle righe in lista delle colonne. La procedura ausiliaria **colonne** consente di separare la matrice in due parti: la prima colonna ed il resto delle colonne; a tal punto si applica ricorsivamente il procedimento di trasposizione al resto della matrice così ottenuto.



```

matrix_transposta([], [], []).
matrix_transposta(Matrice, [Col_1 | Col_n]) :- colonne(Matrice, Col_1, Resto_colonne), matrix_transposta(Resto_colonne, Col_n).
colonne([], [], []).
colonne([[C_11 | C_1n] | C], [C_11 | X], [C_1n | Y]) :- colonne(C, X, Y).

```

La procedura è utilizzata nel seguente programma per la moltiplicazione di due matrici:

```

/*
PROCEDURA: moltiplicazione_matrici(M1, M2, M3).
D: Ma è la matrice prodotto delle matrici M1 ed M2.
P: moltiplicazione_matrici(<, <, >): fornisce in M3 il prodotto di M1 per M2.
C: Si assume, ma non si verifica, che tutti gli elementi delle due matrici siano numeri. La procedura è deterministica, come anche tutte le procedure richiamate.
*/
/* Per moltiplicare due matrici, si traspone la seconda e si formano tutti i prodotti interni */
moltiplicazione_matrici(M1, M2, M_prod) :- matrice_trasposta(M2, M2_trasp), prodotti_interni(M1, M2_trasp, M_prod).
/* Calcolata M2_trasp (la matrice trasposta di M2), si distribuiscono tutte le possibili coppie di righe di M1 con colonne di M2_trasp alla relazione di prodotto interno mediante le due seguenti procedure ausiliarie, la cui struttura interna è identica */
/* Moltiplica tutte le righe della matrice a primo argomento per la matrice a secondo argomento */
prodotti_interni([], _, []).
prodotti_interni([[M1 | Mn], N, [R1 | Rn]] :- prodotti_interni_1(M1, N, R1), prodotti_interni(Mn, N, Rn).
/* Moltiplica tutte le colonne della matrice a secondo argomento per la riga a primo argomento */
prodotti_interni_1(_, [], []).
prodotti_interni_1([M, [N1 | Nn], [R1 | Rn]] :- prodotto_interno(M, N1, R1), prodotti_interni_1(M, Nn, Rn).
/* Somma i prodotti fra gli elementi di ugual posto nelle liste dei primi due argomenti */
prodotto_interno([], [], 0).
prodotto_interno([M1 | Mn], [N1 | Nn], R) :- prodotto_interno(M1, N1, R1), prodotti_interni_1(M, Nn, Rn), R is R1+N1.
matrice_trasposta([], [], []).
matrice_trasposta([Matrice, [Col_1 | Col_n]]) :- colonne(Matrice, Col_1, Resto_colonne), matrice_trasposta(Resto_colonne, Col_n).
colonne([], [], []).
colonne([[C_11 | C_1n] | C], [C_11 | X], [C_1n | Y]) :- colonne(C, X, Y).

```

Volendo accettare se le matrici in ingresso possono essere moltiplicate fra loro, ossia se sono rispettivamente del tipo ($m * n$) ed ($n * m$), si può sostituire la procedura di più alto livello con **moltiplicazione_matrici_1**:

```

verifica_matrici(A, B, si) :- lunghezza(A, M), primo_elemento(A1, A), lunghezza(A1, N), lunghezza(B, N), primo_elemento(B1, B),
lunghezza(B1, M), !.
verifica_matrici(_, _, no) :- write('Ingresso sbagliato.'), nl.
primo_elemento(E, [E | _]).

```

```

/* Calcolata M2_trasp (la matrice trasposta di M2), si distribuiscono tutte le possibili coppie di righe di M1 con colonne di M2_trasp alla relazione di prodotto interno mediante le due seguenti procedure ausiliarie, la cui struttura interna è identica */
/* Moltiplica tutte le righe della matrice a primo argomento per la matrice a secondo argomento */
prodotti_interni([ ], _, [ ]).
prodotti_interni([M1 | Mn], N, [R1 | Rn]) :- prodotti_interni_1(M1, N, R1), prodotti_interni(Mn, N, Rn).
/* Moltiplica tutte le colonne della matrice a secondo argomento per la riga a primo argomento */
prodotti_interni_1(_, [ ], [ ]).
prodotti_interni_1([M, [N1 | Nn], [R1 | Rn]] :- prodotto_interno(M, N1, R1), prodotti_interni_1(M, Nn, Rn).
/* Somma i prodotti fra gli elementi di ugual posto nelle liste dei primi due argomenti */
prodotto_interno([ ], [ ], 0).
prodotto_interno([M1 | Mn], [N1 | Nn], R) :- prodotto_interno(Mn, Nn, X), R is X+M1*N1.
matrice_traspota([[ ] | _, [ ]].
matrice_traspota(Matrice, [Col_1 | Col_n]) :- colonne(Matrice, Col_1, Resto_colonne), matrice_traspota(Resto_colonne, Col_n).
colonne([], [ ], [ ]).
colonne([[C_11 | C_1n] | C], [C_11 | X], [C_1n | Y]) :- colonne(C, X, Y).

```

verifica_matrici riesce in ogni caso; se dopo la sua chiamata il terzo argomento è istanziato a **sì**, si procede con il calcolo del prodotto (ed il taglio previene gli effetti di eventuali ritorni indietro), in caso contrario l'elaborazione si arresta.

Il seguente programma realizza la somma di due matrici. La segnalazione di ingresso non corretto (se le due matrici da sommare non hanno lo stesso numero di righe e lo stesso numero di colonne) viene attivata dalla seconda clausola di **somma_matrici** anziché dalla procedura di verifica:

```

somma_matrici(M1, M2, M_somma) :- verifica_per_somma(M1, M2), !, somma_matrici_1(M1, M2, M_somma).
somma_matrici(_, _, [ ]) :- write('Ingresso sbagliato.'), nl.
verifica_per_somma(M1, M2) :- lunghezza(M1, N), primo_elemento(X, M1), lunghezza(X, L), lunghezza(M2, N), primo_elemento(Y, M2), lunghezza(Y, L).
primo_elemento(E, [E | _]).
somma_matrici_1([ ], [ ], [ ]).
somma_matrici_1([M1 | M1n], [M2 | M2n], [MS1 | MSn]) :- somma_matrici_2(M1, M2, MS1), somma_matrici_1(M1n, M2n, MSn).
somma_matrici_2([ M1 | M1n], [M2 | M2n], [MS1 | MSn]) :- MS1 is M1 + M2, somma_matrici_2(M1n, M2n, MSn).

```

Tecniche di ordinamento.

Si è già vista la procedura **ordinamento ingenuo**, che genera una permutazione della lista assegnata in ingresso e controlla che la lista risultante si trovi nell'ordine desiderato. Se questa verifica fallisce, il ritorno indietro forza la generazione di una permutazione differente; il procedimento continua sino alla generazione della permutazione ordinata. Essa è un classico esempio di procedura iterativa del tipo "generazione e verifica", con contenuto dichiarativo molto chiaro ma caratterizzata da un elevato grado di inefficienza.

Esaminiamo, nel seguito, altre possibili procedure per l'ordinamento di liste di numeri in ordine crescente; esse hanno tutte lo stesso contenuto dichiarativo ("cosa" la procedura fa), mentre hanno differenti contenuti procedurali ("come" lo fa).

Ordinamento per inserimento.

L'algoritmo di ordinamento per inserimento prevede che ogni elemento della lista da ordinare venga estratto da essa e inserito, nella posizione appropriata, in una nuova lista, che in tal modo risulterà ordinata:

```

/*
PROCEDURA: ordinamento_per_inserimento(L1,L2).
D: L2 è la lista L1 ordinata
P1: ordinamento_per_inserimento(<, <): verifica la relazione.
P2: ordinamento_per_inserimento(<, >): fornisce la lista ordinata.
U: inserimento/3.

```

C: l'ordine delle due sottomete della seconda clausola è essenziale, perché i primi due parametri di inserimento devono essere istanziati.

```
/*
/* La lista vuota è ordinata */
ordinamento_per_inserimento([], []).
/* Una lista non vuota è ordinata ordinandone la coda e inserendone la testa nella posizione appropriata della coda ordinata */
ordinamento_per_inserimento([Testa | Coda], Lista_ordinata) :- ordinamento_per_inserimento(Coda, Lista), inserimento(Testa, Lista, Lista_ordinata).
/*
La procedura inserimento è la seguente:
PROCEDURA: inserimento(E, L1, L2).
D: La lista ordinata L2 è uguale alla lista ordinata L1 con in più l'elemento E inserito nell'ordine.
P1: inserimento(<, <, <): verifica la relazione.
P2: inserimento(<, <, >): produce la nuova lista ordinata.
T: rosso, per simulare la struttura "if-then-else".
*/
/* Se l'elemento da inserire è maggiore della testa, va inserito nella coda */
inserimento(E, [T | C], [T | C1]) :- T < E, !, inserimento(E, C, C1).
/* Altrimenti, è inserito come primo elemento */
inserimento(E, Lista, [E | Lista]).
```

Ordinamento a bolla d'aria.

La tecnica di ordinamento a bolla d'aria si fonda sull'osservazione che una successione di elementi non è ordinata se contiene una coppia non ordinata: l'algoritmo migliora, ad ogni passo, l'ordinamento della successione in esame, sottponendo a verifica le coppie di elementi adiacenti della lista da ordinare e scambiandoli nel caso in cui si trovino fuori ordine. Il procedimento viene ripetuto sino a che non si rendono più necessari ulteriori scambi.

```
/*
PROCEDURA: ordinamento_a_bolla(L1, L2).
D: L2 è la lista L1 ordinata.
P1: ordinamento_a_bolla(<, <): verifica la relazione.
P2: ordinamento_a_bolla(<, >): ricerca la lista ordinata.
T: rosso, per simulare la struttura "if-then-else".
U: concatenazione/3.
*/
ordinamento_a_bolla(Lista_ingr, Lista_ord) :- concatenazione(X, [M, N | Y], Lista_ingr), N < M, !, concatenazione(X, [N, M | Y], Lista),
ordinamento_a_bolla(Lista, Lista_ord).
ordinamento_a_bolla(Lista, Lista).
concatenazione([], L, L).
concatenazione([T|L1], L2, [T|L3]):=concatenazione(L1, L2, L3).
```

L'uso di concatenazione consente di selezionare, in maniera non deterministica, gli elementi della lista in ingresso: la prima chiamata genera tutte le coppie di elementi adiacenti nella lista in ingresso, mentre la seconda ricostruisce la lista modificata, che diviene l'argomento della chiamata ricorsiva. Se la prima clausola non è più applicabile, allora tutte le coppie di elementi adiacenti sono ordinate, e la condizione limite, che deve necessariamente stare dopo la clausola ricorsiva, fornisce il risultato desiderato. Altrettanto cruciale è l'ordinamento delle sottomete nella prima clausola, in quanto dapprima vengono ripetutamente isolate le coppie di elementi contigui (l'operazione fallisce se la lista è troppo corta), e si esamina se esse sono ordinate; non appena viene trovata la prima coppia non ordinata, ha luogo la chiamata ricorsiva. Questa versione di **ordinamento_a_bolla** non è molto efficiente in tempo ed in spazio, poiché X, segmento iniziale della lista, viene copiato due volte ad ogni livello di ricorsione.

Ordinamento per divisione e composizione.

Il programma di ordinamento per divisione e composizione si ispira ad una strategia del tipo "divisione e conquista", che muove dall'idea di manipolare liste di dimensioni rilevanti separandole in liste più piccole: ciò richiede la separazione della lista in ingresso in due sottoliste, le cui

lunghezze differiscono al più di una unità, sulle quali ripetere l'operazione di ordinamento, per poi procedere alla loro ricomposizione nella lista ordinata che costituisce l'uscita del programma.

```
/*
PROCEDURA: ordinamento_e_composizione(L1,L2).
D: L2 è la lista L1 ordinata.
P1: ordinamento_e_composizione(<, <): verifica la relazione.
P2: ordinamento_e_composizione(<, >): ricerca la lista ordinata.
U: fusione/3 , lunghezza/2.
C: il compito della procedura fusione è più semplice di quello dell'ordinamento di una lista in quanto può utilizzare la conoscenza del fatto che le due liste in ingresso sono già ordinate.
*/
ordinamento_e_composizione([], []).
ordinamento_e_composizione([E|L], [E|L]).
ordinamento_e_composizione([T1, T2 | C], X) :- separazione([T1, T2 | C], Y1, Y2), ordinamento_e_composizione(Y1, Z1),
ordinamento_e_composizione(Y2, Z2), fusione(Z1, Z2, X).
/* separazione(L, L1, L2) è vera se L1 ed L2 differiscono in lunghezza al più di un elemento e se concatenazione(L1, L2, L) è vera */
separazione(L, L1, L2) :- lunghezza(L, N), N is N//2, separazione_lista(N1, L, L1, L2).
separazione_lista(0, X, [], X).
separazione_lista(N, [T | X], [T | Y], Z) :- 0<N, N1 is N - 1, separazione_lista(N1, X, Y, Z).
lunghezza([], 0).
lunghezza([_|C], N):-lunghezza(C, N1), N is N1 + 1.
/*
PROCEDURA: fusione(S1, S2, S3).
D: S3 è la sequenza risultante dalla fusione ordinata delle sequenze S1 e S2 ordinate in senso crescente.
P1: fusione(<, <, <): verifica la relazione.
P2: fusione(<, <, >): genera la sequenza risultante dalla fusione.
T: verdi, per eliminare la ridondanza dovuta alla necessaria presenza di due condizioni limite. Senza tagli, invece, tale ridondanza può essere eliminata sostituendo la prima condizione limite con fusione([], [T|C], [T|C]).
*/
fusione([], Sequenza, Sequenza) :- !.
fusione(Sequenza, [], Sequenza) :- !.
fusione([T1|C1], [T2|C2], [T1|C]) :- T1<T2, fusione(C1, [T2|C2], C).
fusione([T1|C1], [T2|C2], [T2|C]) :- T1>=T2, fusione([T1|C1], C2, C).
```

Una potenziale inefficienza insita nel programma consiste nel fatto che si rende necessario calcolare la lunghezza di una lista in corrispondenza a ciascuna chiamata ricorsiva. Ciò non è in realtà necessario, poiché la relazione di separazione trova le lunghezze delle liste **L1** ed **L2**, che vengono ricorsivamente ordinate. La procedura **ordinamento_e_composizione** può essere modificata per costituire una relazione fra una coppia (**X**, **L**) ed una lista **Y**, dove **Y** è la versione ordinata di **X**, ed **L** è la lunghezza di **X**: basta modificarne la regola ricorsiva, e modificare la procedura separazione, mentre rimangono invariate le altre procedure:

```
ordinamento_e_composizione_1([], 0, []).
ordinamento_e_composizione_1([X], 1, [X]):- !.
ordinamento_e_composizione_1((X, L), Y) :- separazione_1((X, L), Y1, Y2), ordinamento_e_composizione_1(Y1, Z1),
ordinamento_e_composizione_1(Y2, Z2), fusione(Z1, Z2, Y).
separazione_1((X, Y), (X1, Y1), (X2, Y2)) :- Y1 is Y//2, Y2 is Y - Y1, separazione_lista(Y1, X, X1, X2).
separazione_lista(0, X, [], X).
separazione_lista(N, [T | X], [T | Y], Z) :- 0<N, N1 is N - 1, separazione_lista(N1, X, Y, Z).
/*
PROCEDURA: fusione(S1, S2, S3).
D: S3 è la sequenza risultante dalla fusione ordinata delle sequenze S1 e S2 ordinate in senso crescente.
P1: fusione(<, <, <): verifica la relazione.
P2: fusione(<, <, >): genera la sequenza risultante dalla fusione.
T: verdi, per eliminare la ridondanza dovuta alla necessaria presenza di due condizioni limite. Senza tagli, invece, tale ridondanza può essere eliminata sostituendo la prima condizione limite con fusione([], [T|C], [T|C]).
*/
fusione([], Sequenza, Sequenza) :- !.
fusione(Sequenza, [], Sequenza) :- !.
fusione([T1|C1], [T2|C2], [T1|C]) :- T1<T2, fusione(C1, [T2|C2], C).
fusione([T1|C1], [T2|C2], [T2|C]) :- T1>=T2, fusione([T1|C1], C2, C).
```

Quicksort.

La stessa strategia di fondo della "divisione e conquista" scomporre la lista in due parti, ordinarie ricorsivamente ed unirle1 per ottenere la lista ordinata - può essere utilizzata per programmi di ordinamento del tutto diversi, se si opta per tecniche di "divisione" a loro volta diverse.

Nel caso dei programmi di ordinamento per inserimento e di divisione e composizione, la separazione di lista è semplice, mentre è più complessa l'unione delle liste risultanti. Viceversa, se la lista viene separata in due parti non qualsiasi, ma tali che tutti gli elementi della prima siano inferiori a tutti quelli dell'altra, tale partizionamento più elaborato consente di avvalersi di questa proprietà in fase di fusione delle liste risultanti dalle due chiamate ricorsive: è dunque possibile sostituire la chiamata della procedura [fusione](#) con quella meno onerosa di [concatenazione](#).

```
/*
PROCEDURA: quicksort(L1, L2). L2 è la lista L1 ordinata.
P1: quicksort(<, <): verifica la relazione.
P2: quicksort(<, >): ricerca la lista ordinata.
U: concatenazione/3.
*/
quicksort([], []).
quicksort([T | C], L_ordinata) :- nuova_separazione(T, C, X1, X2), quicksort(X1, Y1), quicksort(X2, Y2), concatenazione(Y1, [T | Y2], L_ordinata).
/* nuova_separazione(E, L, L1, L2): separa L in L1 ed L2 tali che ogni elemento di L inferiore od uguale all'elemento E compare in L1 e tutti gli altri in L2; l'ordine originario degli elementi risulta preservato entro L1 ed L2 */
nuova_separazione(_, [], [], []).
nuova_separazione(T, [T1 | C], [T1 | D1], D2) :- T1 =< T, nuova_separazione(T, C, D1, D2).
nuova_separazione(T, [T1 | C], D1, [T1 | D2]) :- T1>T, nuova_separazione(T, C, D1, D2).
concatenazione([], L, L).
concatenazione([T|L1], L2, [T|L3]) :- concatenazione(L1, L2, L3).
```

Ordinamento ibrido.

Il programma di quicksort è di particolare efficacia su liste di lunghezza apprezzabile, in quanto converge più rapidamente ad una soluzione rispetto ad altri metodi. Tuttavia la quantità di operazioni da effettuare ad ogni ricorsione di quicksort è superiore a quella richiesta dagli altri metodi, in quanto deve utilizzare la procedura **nuova_separazione**, molto costosa sotto il profilo computazionale. La maggiore velocità di esecuzione viene dunque controbilanciata da un più esteso utilizzo della memoria.

Questo fatto suggerisce che in fase di ordinamento di piccole liste sia opportuno sostituire il programma di quicksort con una chiamata ad un altro metodo di ordinamento (per esempio quello per inserimento): sorge allora spontaneo definire un programma "ibrido" che utilizza il quicksort per operare su liste molto lunghe (rese disponibili via via dalla procedura **nuova_separazione**), ma usa un diverso metodo quando, in fase di innestamento delle ricorsioni, le liste fornite in uscita da essa sono abbastanza brevi:

```
/*
PROCEDURA: ordinamento_ibrido(L1, L2).
D: L2 è la lista L1 ordinata.
P1: ordinamento_ibrido(<, <): verifica la relazione.
P2: ordinamento_ibrido(<, >): ricerca la lista ordinata.
T: verdi, per eliminare la ripetizione della stessa soluzione.
U: lunghezza/2, ordinamento_per_inserimento/2, nuova_separazione/4, concatenazione/3.
*/
ordinamento_ibrido([], []) :- !.
ordinamento_ibrido(L, L_ord) :- lunghezza(L, N), N<10, !, ordinamento_per_inserimento(L, L_ord).
ordinamento_ibrido([Testa | Coda], L_ord) :- nuova_separazione(Testa, Coda, X, Y), ordinamento_ibrido(X, X1), ordinamento_ibrido(Y, Y1), concatenazione(X1, [Testa | Y1], L_ord).
/* nuova_separazione(E, L, L1, L2): separa L in L1 ed L2 tali che ogni elemento di L inferiore od uguale all'elemento E compare in L1 e tutti gli altri in L2; l'ordine originario degli elementi risulta preservato entro L1 ed L2 */
nuova_separazione(_, [], [], []).
nuova_separazione(T, [T1 | C], [T1 | D1], D2) :- T1 =< T, nuova_separazione(T, C, D1, D2).
nuova_separazione(T, [T1 | C], D1, [T1 | D2]) :- T1>T, nuova_separazione(T, C, D1, D2).
concatenazione([], L, L).
concatenazione([T|L1], L2, [T|L3]) :- concatenazione(L1, L2, L3).
/*
PROCEDURA: ordinamento_per_inserimento(L1,L2).
D: L2 è la lista L1 ordinata
P1: ordinamento_per_inserimento(<, <): verifica la relazione.
P2: ordinamento_per_inserimento(<, >): fornisce la lista ordinata.
U: inserimento/3.
```

```

C: l'ordine delle due sottomete della seconda clausola è essenziale, perché i primi due parametri di inserimento devono essere istanziati.
*/
/* La lista vuota è ordinata */
ordinamento_per_inserimento([], []).
/* Una lista non vuota è ordinata ordinandone la coda e inserendone la testa nella posizione appropriata della coda ordinata */
ordinamento_per_inserimento([Testa | Coda], Lista_ordinata) :- ordinamento_per_inserimento(Coda, Lista), inserimento(Testa, Lista, Lista_ordinata).
/*
La procedura inserimento è la seguente:
PROCEDURA: inserimento(E, L1, L2).
D: La lista ordinata L2 è uguale alla lista ordinata L1 con in più l'elemento E inserito nell'ordine.
P1: inserimento(<, <, <): verifica la relazione.
P2: inserimento(<, <, >): produce la nuova lista ordinata.
T: rosso, per simulare la struttura "if-then-else".
*/
/* Se l'elemento da inserire è maggiore della testa, va inserito nella coda */
inserimento(E, [T | C], [T | C1]) :- T < E, !, inserimento(E, C, C1).
/* Altrimenti, è inserito come primo elemento */
inserimento(E, Lista, [E | Lista]).
/* lunghezza */
lunghezza([], 0).
lunghezza([_|C], N):-lunghezza(C, N1), N is N1 + 1.

```

Alberi.

Per la rappresentazione di alberi n-ari si possono utilmente impiegare strutture dotate di **n + 1** componenti: il nodo dell'albero e le sue n ramificazioni di primo livello. Le ramificazioni ai livelli successivi vengono a loro volta rappresentate, ricorsivamente, con l'utilizzo di strutture, che fungeranno così da componenti della struttura principale. Consideriamo il caso particolare di alberi binari, che si possono definire come segue.

Un albero binario è una struttura vuota, oppure una struttura che consiste di una radice, un sottoalbero sinistro e un sottoalbero destro, dove la radice è un termine qualsiasi mentre i due sottoalberi devono essere a loro volta alberi binari. Occorrono due simboli speciali per rappresentare tale struttura: un atomo per denotare l'albero vuoto, sia esso vuoto, ed un funtore, chiamiamolo albero, per collegare le componenti, ponendole in un ordine stabilito. Rappresentiamo quindi un albero binario con il termine:

albero(Sottoalb_sinistro, Nodo, Sottoalb_destro)

Ad esempio, l'albero della figura seguente:

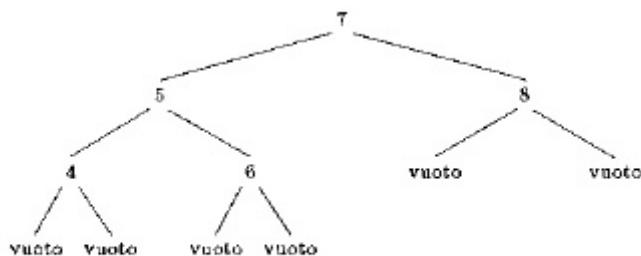


Figura 9.1.

è rappresentato dal termine:

albero(albero(albero(vuoto, 4, vuoto), 5, albero(vuoto, 6, vuoto)), 7, albero(vuoto, 8, vuoto))

La definizione doppiamente ricorsiva della struttura albero è evidenziata dalla seguente relazione, che opera come procedura di controllo che una data struttura sia un albero binario:

```
albero_binario(vuoto).
albero_binario(albero(S, _, D)) :- albero_binario(S), albero_binario(D).
```

Ad esempio, dando alla procedura come argomento d'ingresso il termine precedente si ottiene risposta positiva, mentre si ha un fallimento con il quesito:

?- albero_binario(albero(5, 7, 8)).

Come per tutte le strutture, è possibile definire relazioni che estraiano le componenti di un albero. Per esempio, date le asserzioni:

```
sottoalbero_sinistro(albero(As, _, _), As).
nodo(albero(_, N, _), N).
sottoalbero_destro(albero(_, _, Ad), Ad).
```

i quesiti:

?- sottoalbero_sinistro(albero(albero(albero(vuoto, 4, vuoto), 5, albero(vuoto, 6, vuoto)), 7, albero(vuoto, 8, vuoto)) , As).

?- nodo(albero(albero(albero(vuoto, 4, vuoto), 5, albero(vuoto, 6, vuoto)), 7, albero(vuoto, 8, vuoto)) , N).

?- sottoalbero_destro(albero(albero(albero(vuoto, 4, vuoto), 5, albero(vuoto, 6, vuoto)), 7, albero(vuoto, 8, vuoto)) , Ad).

otterranno rispettivamente come risposta:

As = albero(albero(vuoto, 4, vuoto), 5, albero(vuoto, 6, vuoto))

N = 7

Ad = albero(vuoto, 8, vuoto)

La relazione di appartenenza di un elemento ad un albero binario si esprime nel modo seguente:

```

/* E è un nodo dell'albero se ne è la radice */
in_albero(E, albero(_, E, _)).
/* oppure se è un nodo del sottoalbero sinistro */
in_albero(E, albero(S, _, _)) :- in_albero(E, S).
/* oppure se è un nodo del sottoalbero destro */
in_albero(E, albero(_, _, D)) :- in_albero(E, D).

```



La procedura deve essere chiamata con il secondo argomento sempre istanziato, mentre il primo può esserlo o meno. Lasciando variabile il primo argomento, la procedura effettua, mediante ritorno indietro, l'attraversamento dell'albero. Ad esempio, il quesito:

?- in_albero(E, albero(albero(albero(vuoto, 4, vuoto), 5, albero(vuoto, 6, vuoto)), 7, albero(vuoto, 8, vuoto))).

ottiene le risposte:

E=7;

E=5;

E=4;

E=6;

E=8

Per sostituire un elemento **E** di un albero **A** con un elemento **E1** ottenendo l'albero **A1**:

```

sost(_, vuoto, _, vuoto).
sost(E, albero(S, E, D), E1, albero(S1, E1, D1)) :- sost(E, S, E1, S1), sost(E, D, E1, D1).
sost(E, albero(S, N, D), E1, albero(S1, N, D1)) :- E==N, sost(E, S, E1, S1), sost(E, D, E1, D1).

```



La procedura di ricerca di un elemento in un albero può essere resa più efficiente se vi è un ordinamento tra i nodi, per esempio da sinistra a destra, tale cioè che tutti i nodi del sottoalbero di sinistra sono minori della radice e quelli di destra maggiori, per ogni sottoalbero (l'albero di [Figura 9.1](#) è ordinato). In questo caso la ricerca può essere binaria, cioè per ogni nodo deve essere esaminato solo uno dei due suoi sottoalberi; a questo scopo è sufficiente aggiungere, nella procedura **in_albero**, i controlli sull'ordine:

```

in_albero_ordinato(E, albero(_, E, _)).
in_albero_ordinato(E, albero(S, R, _)) :- maggiore(R, E), in_albero_ordinato(E, S).
in_albero_ordinato(E, albero(_, R, D)) :- maggiore(E, R), in_albero_ordinato(E, D).
maggior(X, Y) :- X>Y.

```



La sottometa **X > Y**, la quale presuppone che i nodi sono numeri, può essere sostituita da **X @> Y** se i nodi sono atomi, o da una relazione d'ordine opportunamente definita se sono strutture.

La procedura seguente inserisce un elemento in un albero binario ordinato se non è già presente, altrimenti riesce ugualmente, ma non duplica il nodo; l'elemento viene sempre inserito come foglia dell'albero, nella posizione adeguata a mantenere l'ordinamento:

```
/* E inserito in albero vuoto */
inser_in_albero(vuoto, E, albero(vuoto, E, vuoto)).
/* se E è uguale alla radice, rimane */
inser_in_albero(albero(S, E, D) ,E,albero(S, E, D)).
/* se E è minore della radice, è inserito nel sottoalbero sinistro */
inser_in_albero(albero(S, N, D), E, albero(S1, N, D)) :- maggiore(N, E), inser_in_albero(S, E, S1).
/* se E è maggiore della radice, è inserito nel sottoalbero destro */
inser_in_albero(albero(S, N, D), E, albero(S, N, D1)):- maggiore(E, N), inser_in_albero(D, E, D1).
maggior(X, Y) :-X>Y.
```

Si noti che la seconda clausola non è proceduralmente necessaria, ma rende più chiaro il programma. La procedura ha la stessa struttura di [in_albero_ordinato](#), con un argomento in più per rappresentare il nuovo albero. Sempre in riferimento all'albero dell'esempio precedente, il quesito:

?- inser_in_albero(albero(albero(albero(vuoto, 4, vuoto), 5, albero(vuoto, 6, vuoto)), 7, albero(vuoto, 8, vuoto)) , 3, A1).

ottiene come risposta:

albero(albero(albero(albero(vuoto, 3, vuoto), 4, vuoto), 5, albero(vuoto, 6, vuoto)), 7, albero(vuoto, 8, vuoto))

mentre il quesito:

?- inser_in_albero(albero(albero(albero(vuoto, 4, vuoto), 5, albero(vuoto, 6, vuoto)), 7, albero(vuoto, 8, vuoto)) , 5, A1).

restituisce:

albero(albero(albero(vuoto, 4, vuoto), 5, albero(vuoto, 6, vuoto)), 7, albero(vuoto, 8, vuoto))

La procedura precedente può essere usata, scambiando i ruoli dei parametri, per cancellare nodi che sono foglie dell'albero, ma non per cancellare nodi interni. In quest'ultimo caso, infatti, la rimozione di un nodo potrebbe lasciare sconnessi i sottoalberi; è quindi necessaria una procedura più articolata. Se l'uno o l'altro dei sottoalberi del nodo da cancellare è vuoto, il sottoalbero non vuoto viene fatto avanzare al posto del nodo. Se invece nessuno dei due sottoalberi è vuoto, per preservare l'ordinamento occorre sostituire il nodo cancellato con l'elemento maggiore del sottoalbero sinistro, o con il minore del sottoalbero destro. La seconda clausola risponde a motivi di simmetria e di efficienza, ma anche in sua assenza il comportamento del programma è quello desiderato:

/* Se l'elemento da cancellare è uguale alla radice ed il sottoalbero sinistro è vuoto, il sottoalbero destro è il nuovo albero */
cancell_da_albero(albero(vuoto, N, D), N, D).

```

/* Se l'elemento da cancellare è uguale alla radice ed il sottoalbero destro è vuoto, il sottoalbero sinistro è il nuovo albero */
cancell_da_albero(albero(S, N, vuoto), N, S).
/* Se l'elemento da cancellare è uguale alla radice ed entrambi i sottoalberi non sono vuoti, l'elemento massimo del sottoalbero sinistro sostituisce quello cancellato */
cancell_da_albero(albero(S, N, D), N, albero(S1, E_max, D)) :- spostamento(S, E_max, S1).
/* Ricerca l'elemento da cancellare se diverso dalla radice */
cancell_da_albero(albero(S, R, D), N, albero(S1, R, D)) :- maggiore(R, N), cancell_da_albero(S, N, S1). cancell_da_albero(albero(S, R, D), N, albero(S, R, D1)) :- maggiore(N, R), cancell_da_albero(D, N, D1).
/* Cerca e sposta l'elemento massimo (il più a destra del sottoalbero sinistro) */
spostamento(albero(S, E_max, vuoto), E_max, S).
spostamento(albero(S, R, D), E_max, albero(S, R, D1)):- spostamento(D, E_max, D1).

```

La seguente procedura effettua l'attraversamento di un albero binario, dando in uscita la lista degli elementi:



```

attraversamento(albero(S, E, D), L) :- attraversamento(S, S1), attraversamento(D, D1), concatenazione(S1, [E | D1], L).
attraversamento(vuoto, []).
concatenazione([], L, L).
concatenazione([T|L1], L2, [T|L3]):-concatenazione(L1, L2, L3).

```

Se l'albero è ordinato, la lista d'uscita risulta ordinata. Infatti gli elementi dell'albero vengono considerati nel seguente ordine (detto in-ordine): prima i nodi del sottoalbero di sinistra, poi il nodo radice, e quindi i nodi del sottoalbero di destra (e lo stesso per ogni sottoalbero). Vi sono altri due possibili modi di attraversamento, detti di pre-ordine (prima la radice, poi i nodi del sottoalbero sinistro, ed infine quelli del destro), e di post-ordine (la radice dopo i nodi dei sottoalberi sinistro e destro).

Si ottiene la lista dei nodi in pre-ordine sostituendo la precedente sottometà **concatenazione** con:

concatenazione([E | S1], D1, L)

ottenendo:



```

attraversamento(albero(S, E, D), L) :- attraversamento(S, S1), attraversamento(D, D1), concatenazione([E | S1], D1, L).
attraversamento(vuoto, []).
concatenazione([], L, L).
concatenazione([T|L1], L2, [T|L3]):-concatenazione(L1, L2, L3).

```

nell'esempio si ha: **[7, 5, 4, 6, 8]** cioè lo stesso ordine ottenuto prima attivando il ritorno indietro con la procedura **in_albero**. Infatti il pre-ordine corrisponde alla ricerca in profondità nell'albero, tipica dell'interprete Prolog. Per ottenere invece la lista dei nodi in post-ordine occorre sostituire una doppia sottometà di concatenazione:

concatenazione(D1, [E], D2), concatenazione(S1, D2, L)

ottenendo:

```

attraversamento(albero(S, E, D), L) :- attraversamento(S, S1), attraversamento(D, D1), concatenazione(D1, [E], D2),
concatenazione(S1, D2, L).
attraversamento(vuoto, [ ]).
concatenazione([], L, L).
concatenazione([T|L1], L2, [T|L3]):-concatenazione(L1, L2, L3).

```

nell'esempio si ha la lista: **[4,6, 5, 8, 7]**. Naturalmente è anche possibile realizzare la costruzione della lista senza l'uso esplicito di **concatenazione**, usando un argomento aggiuntivo, come nella seguente versione della procedura di attraversamento in-ordine:

```

attraversamento_1(A, L) :- attr(A, L, [ ]).
attr(vuoto, L, L).
attr(albero(S, E, D), L, L1) :- attr(S, L, [E | D1]), attr(D, D1, L1).

```

Il seguente programma costruisce un albero binario ordinato a partire da una lista non ordinata. La costruzione dell'albero avviene identificando la sua radice con il primo elemento della lista; il ramo sinistro comprenderà poi gli elementi inferiori alla radice, il ramo destro quelli superiori. In questo caso, l'albero vuoto è denotato dall'atomo lista vuota (**[]**).

```

/*
PROCEDURA: formazione_albero(L, A).
D: A è l'albero binario ordinato contenente tutti e soli gli elementi della lista non ordinata L.
P1: formazione_albero(<, <): verifica la relazione.
P2: formazione_albero (<, >): costruisce l'albero ordinato.
U: nuova_separazione/4
*/
formazione_albero([], []).
formazione_albero([El], albero([ ], El, [ ])).
formazione_albero([T | C], albero(C1, T, C2)) :- nuova_separazione(T, C, X1, X2), formazione_albero(X1, C1), formazione_albero(X2, C2).
nuova_separazione(_, [], [], []).
nuova_separazione(T, [T1 | C], [T1 | D1], D2) :- T1 =< T, nuova_separazione(T, C, D1, D2).
nuova_separazione(T, [T1 | C], D1, [T1 | D2]) :- T1 > T, nuova_separazione(T, C, D1, D2).

```

La procedura che segue realizza un altro tipo di ordinamento di una lista: la lista d'ingresso viene trasformata in un albero binario ordinato, e da questo viene poi costruita la lista d'uscita ordinata, attraversandolo in-ordine. Per l'attraversamento in-ordine si utilizza la procedura vista prima, sostituendo nella condizione limite l'atomo **[]** all'atomo **vuoto**.

```

/*
PROCEDURA: ordinamento_con_albero(L1, L2).
D: L2 è la lista L1 ordinata.
P1: ordinamento_con_albero(<, <): verifica la relazione.
P2: ordinamento_con_albero(<, >): costruisce la lista ordinata.
U: formazione_albero/2, attraversamento_1/2 .
*/
ordinamento_con_albero(Lista, Lista_ordinata) :- formazione_albero(Lista, Albero), write(Albero), nl,attraversamento_1(Albero, Lista_ordinata).
attraversamento_1(A, L) :- attr(A, L, [ ]).
attr([ ], L, L).
attr(albero(S, E, D), L, L1) :- attr(S, L, [E | D1]), attr(D, D1, L1).
/*
PROCEDURA: formazione_albero(L, A).
D: A è l'albero binario ordinato contenente tutti e soli gli elementi della lista non ordinata L.
P1: formazione_albero(<, <): verifica la relazione.
P2: formazione_albero (<, >): costruisce l'albero ordinato.
U: nuova_separazione/4
*/
formazione_albero([], []).
formazione_albero([El], albero([ ], El, [ ])).

```

```

formazione_albero([T | C], albero(C1, T, C2)) :- nuova_separazione(T, C, X1, X2), formazione_albero(X1, C1), formazione_albero(X2, C2).
nuova_separazione(_, [], [], []).
nuova_separazione(T, [T1 | C], [T1 | D1], D2) :- T1 =< T, nuova_separazione(T, C, D1, D2).
nuova_separazione(T, [T1 | C], D1, [T1 | D2]) :- T1 > T, nuova_separazione(T, C, D1, D2).

```

La formulazione del quesito:

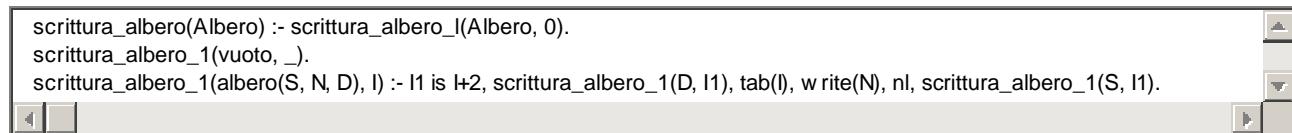
?- ordinamento_con_albero([5,3,6,1,0], S).

porta ad ottenere l'uscita:

albero(albero(albero(albero([], 0, []), 1, []), 3, []), 5, albero([], 6, [])))

S = [0,1,3,5,6]

La procedura seguente consente di scrivere un albero in forma più leggibile del termine che lo rappresenta; la radice è scritta per prima, il sottoalbero destro è scritto su righe più in alto di essa e quello sinistro su righe più in basso, con uno spostamento di due posizioni più a destra per ogni livello di ramificazione:



```

scrittura_albero(Albero) :- scrittura_albero_1(Albero, 0).
scrittura_albero_1(vuoto, _).
scrittura_albero_1(albero(S, N, D), I) :- I1 is I+2, scrittura_albero_1(D, I1), tab(I), write(N), nl, scrittura_albero_1(S, I1).

```

(il predicato predefinito **tab(N)** produce **N** spazi sulla riga corrente). Ad esempio, l'albero della [Figura 9.1](#) viene scritto nella forma:

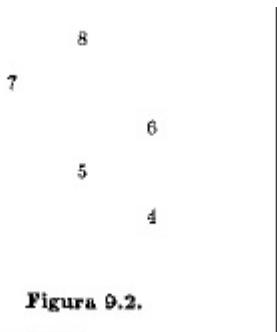


Figura 9.2.

Grafi.

Un grafo è costituito da nodi connessi da archi, che possono essere orientati (unidirezionali) o non orientati (bidirezionali). Il modo più diretto per rappresentare un grafo è mediante clausole unitarie, che esprimono la relazione di connessione di due nodi mediante un arco, ad esempio:

arco(a, b).

arco(a, c).

arco(b, d).

arco(c, d).

arco(d, e).

corrispondente al grafo:

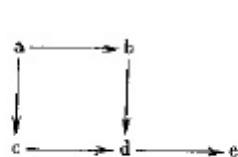


Figura 9.3.

Tale relazione si estende facilmente a quella di connessione di due nodi mediante un qualunque numero di archi attraverso nodi intermedi:

/* un nodo è connesso a un altro nodo se è connesso ad esso da un arco */

connessione(Nodo_1, Nodo_2) :- arco(Nodo_1, Nodo_2).

/* o se è connesso da un arco ad un altro nodo connesso a quel nodo */

connessione(Nodo_1, Nodo_2) :- arco(Nodo_1, Nodo), connessione(Nodo, Nodo_2).

Questa definizione dichiarativa della relazione può essere usata come procedura che consente di stabilire se un nodo è connesso ad un altro nodo, o di trovare tutti i nodi connessi ad un nodo dato, o ancora di trovare tutte le coppie di nodi connessi (per questi ultimi due usi si ottiene tante volte la stessa soluzione quanti sono i modi di ottenerla, considerando nodi intermedi diversi).

Usando un argomento aggiuntivo come accumulatore, si può ottenere la successione di nodi da attraversare per passare da un nodo di partenza ad un altro di arrivo, cioè il cammino tra i due nodi dati (inclusi):

cammino(Nodo, Nodo, [Nodo]).

cammino(Nodo_1, Nodo_2, [Nodo_1 | C]) :- arco(Nodo_1, Nodo), cammino(Nodo, Nodo_2, C).

La procedura **connessione** funziona correttamente con grafi orientati senza cicli, diversamente l'esecuzione può entrare in un ciclo infinito. Consideriamo ad esempio il grafo:

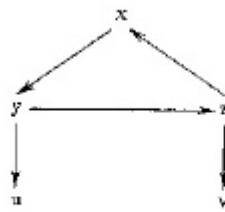


Figura 9.4.

espresso dalle clausole:

arco(x, y).

arco(y, z).

arco(z, x).

arco(y, u).

arco(z, v).

Aggiungendo tali fatti si ottiene:

```
/* un nodo è connesso a un altro nodo se è connesso ad esso da un arco */
connessione(Nodo_1, Nodo_2) :- arco(Nodo_1, Nodo_2).
/* o se è connesso da un arco ad un altro nodo connesso a quel nodo */
connessione(Nodo_1, Nodo_2) :- arco(Nodo_1, Nodo), connessione(Nodo, Nodo_2).
cammino(Nodo, Nodo, [Nodo]).
cammino(Nodo_1, Nodo_2, [Nodo_1 | C]) :- arco(Nodo_1, Nodo), cammino(Nodo, Nodo_2, C).
arco(x, y).
arco(y, z).
arco(z, x).
arco(y, u).
arco(z, v).
```

Con il quesito:

?- connessione(x, u).

la procedura entra in un ciclo infinito, e con il quesito:

?- connessione(x, N).

mediante ritorno indietro vengono ripetute indefiniteamente le soluzioni **x, y, z**, senza mai ottenere le soluzioni **u, v**. Si può risolvere il problema aggiungendo alla procedura **connessione** un argomento accumulatore dei nodi già traversati, che viene esaminato per ogni nuovo nodo in modo da non considerare due volte:

```
connessione_1(Nodo_1, Nodo_2) :- conn(Nodo_1, Nodo_2, [Nodo_1]).
```

```
conn(Nodo, Nodo, Lista_nodi).
```

```
conn(Nodo_1, Nodo_2, Lista_nodi) :- arco(Nodo_1, Nodo), non_appartenenza(Nodo, Lista_nodi), conn(Nodo, Nodo_2, [Nodo | Lista_nodi]).
```

Aggrendendo i fatti dell'esempio precedente, si ottiene:

```
connessione_1(Nodo_1, Nodo_2) :- conn(Nodo_1, Nodo_2, [Nodo_1]).  
conn(Nodo, Nodo, _).  
conn(Nodo_1, Nodo_2, Lista_nodi) :- arco(Nodo_1, Nodo), non_appartenenza(Nodo, Lista_nodi), conn(Nodo, Nodo_2, [Nodo | Lista_nodi]).  
non_appartenenza(_, []).  
non_appartenenza(E, [T | C]) :- E\== T, non_appartenenza(E, C).  
arco(x, y).  
arco(y, z).  
arco(z, x).  
arco(y, u).  
arco(z, v).
```

Se il grafo è non orientato, la relazione arco dev'essere riflessiva; a tale scopo si può sostituire la sottometa **arco(X, Y)** delle procedure precedenti con la sottometa adiacente, definita in modo disgiuntivo:

```
adiacente(N1, N2) :- arco(N1, N2).
```

```
adiacente(N1, N2) :- arco(N2, N1).
```

Ottenendo il programma:

```
conn(Nodo_1, Nodo_2, Lista_nodi) :- adiacente(Nodo_1, Nodo), non_appartenenza(Nodo, Lista_nodi), conn(Nodo, Nodo_2, [Nodo | Lista_nodi]).  
non_appartenenza(_, []).  
non_appartenenza(E, [T | C]) :- E\== T, non_appartenenza(E, C).  
adiacente(N1, N2) :- arco(N1, N2).  
adiacente(N1, N2) :- arco(N2, N1).  
arco(x, y).  
arco(y, z).  
arco(z, x).  
arco(y, u).  
arco(z, v).
```

Questa sostituzione, se effettuata nella procedura **connessione**, la condurrebbe a cicli infiniti, mentre può essere effettuata nella procedura **connessione_1** senza dar luogo a tale problema. La ricerca dei cammini di un grafo può essere sottoposta a particolari vincoli. Ad esempio, quello della figura seguente:

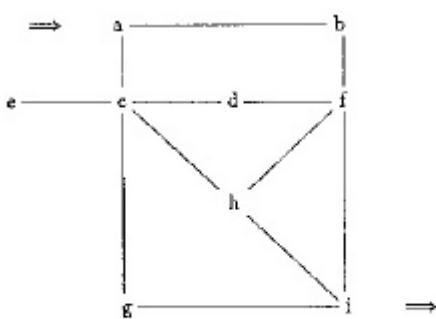


Figura 9.5.

è un grafo non orientato caratterizzato da un nodo di partenza **a**, da un nodo di arrivo **i**, da due nodi proibiti **b** ed **h** e da un nodo obbligato **f**. L'obiettivo consiste nel percorrere il grafo (che rappresenta un labirinto) passando per il nodo obbligato, evitando i nodi proibiti e terminando il percorso nel nodo finale. Rappresentiamo tale grafo con i seguenti fatti:

arco(a, b).

arco(a, c).

arco(b, f).

arco(c, d).

arco(c, e).

arco(d, f).

arco(f, h).

arco(f, i).

arco(c, g).

arco(g, i).

arco(c, h).

arco(h, i).

Definiamo una relazione, **cammino(Partenza, Arrivo, Nodi_proibiti, Lista_nodi)** con un argomento aggiuntivo, rispetto all'omonima relazione precedente, che rappresenta la lista dei nodi proibiti. Questi possono essere facilmente evitati controllando, ogni volta che si raggiunge un nuovo **Nodo**, che esso non sia compreso fra quelli proibiti (oltre al controllo che non si tratti di uno di quelli già visitati, per non incorrere in circoli viziosi). Il programma per la percorrenza del labirinto è:

```

nodi_da_evitare([b,h]).  

nodo_obb1igato(f).  

attraversamento(Partenza, Arrivo) :- nodi_da_evitare(Nodi_proibiti), cammino(Partenza, Arrivo, Nodi_proibiti, [Partenza]).
```

```

/* Clausola utilizzata solo quando è già stato raggiunto il nodo d'arrivo; la sottometa appartenenza impone che il nodo obbligato
appartenga alla lista dei nodi visitati; la sottometa inversione_1 fornisce la lista contenente i nodi del cammino nell'ordine da quello
iniziale a quello finale, in quanto in Lista_nodi essi vengono ottenuti in ordine inverso */
cammino(Arrivo, Arrivo, _, Lista_nodi) :- nodo_obbligato(N_obbligato), appartenenza(N_obbligato, Lista_nodi), inversione_1(Lista_nodi, Cammino), write(Cammino).
/* Clausola utilizzata se il nodo di arrivo non è stato ancora raggiunto, o se è stato raggiunto ma il nodo obbligato non appartiene alla
lista dei nodi visitati */
cammino(Da, A, Nodi_proibiti, Lista_nodi) :- adiacente(Da, Nodo), non_appartenenza(Nodo, Nodi_proibiti), non_appartenenza(Nodo, Lista_nodi), cammino(Nodo, A, Nodi_proibiti, [Nodo | Lista_nodi]).
non_appartenenza(_, []).
non_appartenenza(E, [T | C]) :- E\== T, non_appartenenza(E, C).
arco(a, b).
arco(a, c).
arco(b, f).
arco(c, d).
arco(c, e).
arco(d, f).
arco(f, h).
arco(f, i).
arco(c, g).
arco(g, i).
arco(c, h).
arco(h, i).

```

Formulando il quesito:

?- attraversamento(a, i).

si ottiene l'unica soluzione **[a, c, d, f, i]**. Se vi fossero altre soluzioni, si potrebbero ottenere forzando il ritorno indietro; il risoddisfacimento della meta attraversamento non sarebbe invece possibile se all'interno della prima clausola per la procedura cammino, dopo la chiamata ad appartenenza, comparisse un taglio.

Si osservi la facilità con la quale può essere seguita la logica del programma, dovuta al fatto che esso specifica semplicemente che cosa si richiede di fare alla procedura cammino, e non come farlo; quest'ultimo problema viene completamente delegato al sistema Prolog, la cui strategia predefinita consente l'esplorazione in profondità del grafo. Il programma può essere facilmente generalizzato rendendo parametrica la lista dei nodi proibiti e di quelli obbligati.

Un cammino di un grafo è detto cammino di Eulero se passa per tutti i nodi percorrendo ogni arco una ed una sola volta; si definisce grafo di Eulero un grafo che contenga (almeno) un cammino di Eulero. Per esempio, nel grafo della figura seguente:

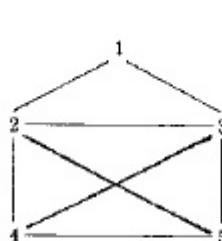


Figura 9.6.

il cammino **421325345** è un cammino di Eulero. Rimuovendo uno dei due archi **(3, 4)** o **(2, 5)** si otterebbe ancora un cammino di Eulero, mentre ciò non vale se si rimuove l'arco **(2, 3)**. Un programma per la ricerca di cammini di Eulero di un grafo non orientato può basarsi su due proprietà che seguono direttamente dalle definizioni precedenti: un grafo con un solo arco è un grafo di Eulero; un grafo è di Eulero se, sottraendovi un arco, il grafo che si ottiene è anch'esso di

Eulero e contiene un cammino di Eulero che comincia da uno dei nodi dell'arco sottratto. S'intende, così, che l'arco sottratto è uno dei due archi terminali di un cammino di Eulero.

Conveniamo di rappresentare un grafo non orientato privo di nodi isolati (in caso contrario non sarebbe un grafo di Eulero) con un insieme (una lista, in cui l'ordine è irrilevante) di termini **arco(N1, N2)**, dove **N1** ed **N2** sono i nodi dell'arco. Un cammino viene rappresentato mediante liste di nodi; la relazione **cammino(L, E)** è vera se **L** è l'insieme di archi presenti in un grafo di Eulero ed **E** è un cammino di Eulero di tale grafo. Le due proprietà prima esposte vengono facilmente formulate con le seguenti clausole:

```

cammino([arco(N1,N2)], [N1,N2]).  

cammino([arco(N2, N1)], [N1, N2]).  

cammino(L_archi, [N1,N2 | Nodi_rimanenti]) :- sottrazione_arco(L_archi, arco(N1, N2), Nuova_L_archi), cammino(Nuova_L_archi, [N2 | Nodi_rimanenti]).  

cammino(L_archi, [N1, N2 | Nodi_rimanenti]) :- sottrazione_arco(L_archi, arco(N2, N1), Nuova_L_archi), cammino(Nuova_L_archi, [N2 | Nodi_rimanenti]).  

/* La relazione sottrazione_arco(X, Y, Z) vale se la lista Z si ottiene sottraendo il termine Y dalla lista X; denota cioè l'operazione X - Y = Z */  

sottrazione_arco([arco(N1, N2) | Archi], arco(N1, N2), Archi).  

sottrazione_arco([Arco | Archi], Arco_da_sottrarre, [Arco | Resto]) :- sottrazione_arco(Archi, Arco_da_sottrarre, Resto).  

arco(1,3).  

arco(2,1).  

arco(2,5).  

arco(3,2).  

arco(3,4).  

arco(4,2).  

arco(4,5).  

arco(5,3).

```

Il programma necessita di un'ottimizzazione, dal momento che - nel caso più sfavorevole - la **Lista_archi** va attraversata due volte prima di individuare l'arco da estrarre. Si può superare questo problema rendendo sensibile all'ordine dei nodi la procedura **sottrazione_arco** anziché **cammino**. Ciò si ottiene aggiungendo la clausola:

sottrazione_arco([arco(N2, N1) | Altri_archi], arco(N1, N2), Altri_archi).

e cancellando una qualsiasi delle due clausole ricorsive di cammino.

```

cammino_1([arco(N1,N2)], [N1,N2]).  

cammino_1([arco(N2, N1)], [N1, N2]).  

cammino_1(L_archi, [N1,N2 | Nodi_rimanenti]) :- sottrazione_arco(L_archi, arco(N1, N2), Nuova_L_archi), cammino_1(Nuova_L_archi, [N2 | Nodi_rimanenti]).  

/* La relazione sottrazione_arco(X, Y, Z) vale se la lista Z si ottiene sottraendo il termine Y dalla lista X; denota cioè l'operazione X - Y = Z */  

sottrazione_arco([arco(N1, N2) | Archi], arco(N1, N2), Archi).  

sottrazione_arco([Arco | Archi], Arco_da_sottrarre, [Arco | Resto]) :- sottrazione_arco(Archi, Arco_da_sottrarre, Resto).  

sottrazione_arco([arco(N2, N1) | Altri_archi], arco(N1, N2), Altri_archi).  

arco(1,3).  

arco(2,1).  

arco(2,5).  

arco(3,2).  

arco(3,4).  

arco(4,2).  

arco(4,5).  

arco(5,3).

```

La procedura cammino può venire usata in maniera non deterministica, per la ricerca di tutti i cammini di Eulero di un grafo assegnato, come in:

```

cammini_Eulero(_).  

cammino_1([arco(N1,N2)], [N1,N2]).  

cammino_1([arco(N2, N1)], [N1, N2]).  

cammino_1(L_archi, [N1,N2 | Nodi_rimanenti]) :- sottrazione_arco(L_archi, arco(N1, N2), Nuova_L_archi), cammino_1(Nuova_L_archi, [N2 | Nodi_rimanenti]).

```

```

/* La relazione sottrazione_arco(X, Y, Z) vale se la lista Z si ottiene sottraendo il termine Y dalla lista X; denota cioè l'operazione X - Y = Z */
sottrazione_arco([arco(N1, N2) | Archi], arco(N1, N2), Archi).
sottrazione_arco([Arco | Archi], Arco_da_sottrarre, [Arco | Resto]) :- sottrazione_arco(Archi, Arco_da_sottrarre, Resto).
sottrazione_arco([arco(N2, N1) | Altri_archi], arco(N1, N2), Altri_archi).
arco(1,3).
arco(2,1).
arco(2,5).
arco(3,2).
arco(3,4).
arco(4,2).
arco(4,5).
arco(5,3).

```

oppure in maniera deterministica, con l'invocazione del predicato di taglio, per controllare se il grafo è di Eulero, trovando in esso un cammino di Eulero, come in:

```

grafo_Eulero(_) :- write('Grafo non di Eulero.').
cammino_1([arco(N1,N2)], [N1,N2]).
cammino_1([arco(N2, N1)], [N1, N2]).
cammino_1(L_archi, [N1,N2 | Nodi_rimanenti]) :- sottrazione_arco(L_archi, arco(N1, N2), Nuova_L_archi), cammino_1(Nuova_L_archi, [N2 | Nodi_rimanenti]).
/* La relazione sottrazione_arco(X, Y, Z) vale se la lista Z si ottiene sottraendo il termine Y dalla lista X; denota cioè l'operazione X - Y = Z */
sottrazione_arco([arco(N1, N2) | Archi], arco(N1, N2), Archi).
sottrazione_arco([Arco | Archi], Arco_da_sottrarre, [Arco | Resto]) :- sottrazione_arco(Archi, Arco_da_sottrarre, Resto).
sottrazione_arco([arco(N2, N1) | Altri_archi], arco(N1, N2), Altri_archi).
arco(1,3).
arco(2,1).
arco(2,5).
arco(3,2).
arco(3,4).
arco(4,2).
arco(4,5).
arco(5,3).

```

Ulteriori possibilità di utilizzo della procedura **cammino_1** consistono nella costruzione del grafo di Eulero contenente un cammino di Eulero assegnato, e nella ricerca di tutti i cammini di Eulero di un grafo a partire da uno solo di essi, rispettivamente mediante:

```

ricerca_cammini(Cammino_Eulero) :- cammino_1(Lista_archi, Cammino_Eulero), !, cammini_Eulero(Lista_archi).
cammino_1([arco(N1,N2)], [N1,N2]).
cammino_1([arco(N2, N1)], [N1, N2]).
cammino_1(L_archi, [N1,N2 | Nodi_rimanenti]) :- sottrazione_arco(L_archi, arco(N1, N2), Nuova_L_archi), cammino_1(Nuova_L_archi, [N2 | Nodi_rimanenti]).
/* La relazione sottrazione_arco(X, Y, Z) vale se la lista Z si ottiene sottraendo il termine Y dalla lista X; denota cioè l'operazione X - Y = Z */
sottrazione_arco([arco(N1, N2) | Archi], arco(N1, N2), Archi).
sottrazione_arco([Arco | Archi], Arco_da_sottrarre, [Arco | Resto]) :- sottrazione_arco(Archi, Arco_da_sottrarre, Resto).
sottrazione_arco([arco(N2, N1) | Altri_archi], arco(N1, N2), Altri_archi).
arco(1,3).
arco(2,1).
arco(2,5).
arco(3,2).
arco(3,4).
arco(4,2).
arco(4,5).
arco(5,3).

```

Note bibliografiche.

Studi comparativi sulle strutture di dati per la programmazione logica compaiono nei lavori di Kowalski (1979a) e Kowalski (1979b), nei quali vengono proposti esempi che utilizzano - per la rappresentazione dei dati - sia termini strutturati che asserzioni. Esempi di uso di termini per l'elaborazione di liste, alberi ed altre strutture sono illustrati anche in Clark e Tarnlund (1977). In Hogger (1981) viene illustrato il compito di trasformare logicamente un programma basato su termini in uno basato su asserzioni.

L'elencazione sistematica delle circostanze nelle quali è opportuno ricorrere all'uso di liste è discussa in O'Keefe (1983).

La trattazione relativa a unimodale è stata ripresa da Hogger (1984). Il programma per il calcolo del prodotto di due matrici è stato tratto da Conery e Kibler (1985). Il programma di ordinamento per divisione e composizione è stato proposto da Clark e McCabe (1984). Il programma di ordinamento a bolla d'aria è accreditato da Coelho, Cotta e Pereira (1982) a van Emden. Il programma di quicksort è un'implementazione dell'algoritmo proposto in Hoare (1961).

La trattazione del problema dei cammini di Eulero di un grafo compare in Kluzniak e Szpakowicz (1985). Il rapporto interno di Coelho, Cotta e Pereira (1982) costituisce un eserciziario di numerosi problemi svolti in Prolog.

Sommario.

Lo studente ha ora un quadro abbastanza ampio delle possibilità di rappresentazione e di elaborazione fornite dall'uso delle clausole di Horn come meccanismo di computazione, già di per sé sufficientemente flessibile e potente.

Tuttavia, allo scopo di rispondere alle diverse esigenze pratiche della programmazione, Prolog fornisce un insieme ulteriore di costrutti, generalmente sotto forma di predici predefiniti. Per uniformità essi hanno quindi la stessa sintassi del linguaggio logico finora considerato, ma non un'analogia interpretazione logica o dichiarativa.

10. Operatori e predicati aritmetici

Dove si considerano l'introduzione di operatori per estendere la sintassi standard di Prolog, e la valutazione di espressioni aritmetiche come meccanismo di computazione aggiuntivo rispetto all'unificazione.

Gli operatori in Prolog.

Gli operatori sono, in Prolog, una semplice convenienza notazionale, un'alternativa sintattica per facilitare l'utilizzo, la manipolazione e la leggibilità degli usuali termini strutturati. È possibile rappresentare nella sintassi di operatore strutture con 1 o 2 argomenti dichiarando quale operatore l'atomo usato come funtore principale della struttura, e scrivendo quindi le componenti della struttura quali argomenti dell'operatore. La sintassi Prolog prevede tre categorie di operatori: infissi, prefissi e postfissi. Un operatore infisso può venire utilizzato per strutture binarie, e compare sempre fra i suoi due argomenti; operatori prefissi o postfissi possono essere usati per strutture unarie, e figurano sempre rispettivamente prima o dopo il loro argomento. Per esempio, strutture come:

> (X,Y)

riceve('Arianna', 'una visita')

;(H, K)

fatto(lavoro)

appartiene_a(Y, Z)

possono venire più convenientemente scritte come:

X>Y

'Arianna' riceve 'una visita'

H;K

lavoro fatto

Y appartiene_a Z

Chiameremo espressione una struttura il cui funtore è rappresentato come operatore. E da notare tuttavia che, ad esempio, l'espressione **4 * 3** rappresenta unicamente la struttura ***(4, 3)** e non il numero **12**. L'espressione verrebbe valutata (nell'esempio, la moltiplicazione verrebbe eseguita) soltanto se tale struttura fosse passata come argomento in ingresso agli operatori di sistema rivolti a questo scopo.

Poiché una struttura può avere argomenti che sono a loro volta strutture, quando i suoi funtori sono rappresentati come operatori si ha un'espressione contenente sottoespressioni. Diventa allora necessario evitare ambiguità su quali sono le sottoespressioni componenti, definendo precedenza ed associatività degli operatori.

Precedenza di un operatore.

A ciascun operatore va associata una precedenza (o priorità), rappresentata da un numero intero compreso in un intervallo, che cambia da implementazione ad implementazione: in alcuni casi si estende da **1** a **255**, in altri da **1** a **1200**, come nel **Prolog/DEC-10** e nel **CProlog**, od altri ancora; in tutti i casi, a numeri più piccoli corrisponde una precedenza più alta.

La precedenza serve ad eliminare l'ambiguità di espressioni la cui struttura non è resa esplicita mediante l'uso delle parentesi; la regola generale consiste nel considerare quale funtore principale l'operatore dotato della precedenza più alta, ovvero con numero di precedenza minore. Così, se **-** ha una precedenza più bassa di *****, allora le espressioni:

a-b*c

a-(b*c)

sono tra loro equivalenti e rappresentano entrambe la struttura:

-(a, *(b, c))

La forma infissa del termine ***(-(a,b),c)** va scritta con le parentesi:

(a - b) * c

Alle parentesi viene tacitamente assegnato numero di precedenza zero (o negativo), in modo da prevalere su ogni altro operatore; esse possono quindi essere usate per annullare gli effetti della precedenza e dell'associatività definite, nonché per esplicitare con maggiore evidenza la struttura.

Se, in una data espressione o sottoespressione, sono presenti due operatori che hanno la stessa precedenza, l'ambiguità che così si determina può essere risolta definendo l'associatività (o tipo) degli operatori.

Associatività di un operatore.

L'associatività specifica se l'operatore è infisso, prefisso o postfisso; specifica inoltre come va interpretata un'espressione contenente più operatori dotati della stessa precedenza. Per definirla si hanno le seguenti possibilità:

natura dell'operatore	tipo
prefisso (unario)	fx
infisso (binario)	fy x _f x
postfisso (unario)	xf y _f

In questa notazione f rappresenta l'operatore, x e y i tipi degli argomenti, nelle corrispondenti posizioni. Una x (rispettivamente, una y) rappresenta un argomento che, se contiene operatori, può contenere solo operatori di precedenza strettamente inferiore (rispettivamente, uguale od inferiore) a quella di f .

Consideriamo dapprima il caso degli operatori infissi. Con un operatore del tipo xfx i funtori principali di entrambe le sottoespressioni che costituiscono gli argomenti dell'operatore devono essere di precedenza più bassa rispetto all'operatore stesso, a meno che la sottoespressione non risulti racchiusa tra parentesi (il che le assegna precedenza massima). Un operatore del tipo xfx è dunque non associativo. Con un operatore di tipo xfy , invece, dev'essere di precedenza inferiore la sottoespressione alla sua sinistra, mentre l'altra potrà essere di precedenza inferiore o uguale a quella dell'operatore principale, che sarà dunque associativo a destra. Il viceversa vale naturalmente per un operatore del tipo yfx , che risulterà così associativo a sinistra.

Ad esempio, se gli operatori "+" e "-" sono entrambi di tipo yfx ed hanno lo stesso numero di precedenza, allora l'espressione:

a-b+c

è corretta e va interpretata come:

(a-b)+c ossia: $+(-(a,b),c)$

e non come:

a-(b + c) che equivarrebbe a: $-(a, +(b, c))$

Risulta anche possibile costruire espressioni sintatticamente corrette, che tuttavia non si conformano alle suddette regole di precedenza e di associatività, e che sono pertanto potenzialmente ambigue; ne è un esempio l'espressione precedente nel caso che entrambi gli operatori siano di tipo xfx . In tali situazioni si considera dapprima l'operatore a sinistra, quindi l'espressione precedente verrebbe interpretata come:

(a-b) +c

Significherebbe invece:

a-(b+c) ossia $-(a, +(b, c))$

se i tipi fossero entrambi xfy .

Ancora, se l'operatore ">>" è associativo a sinistra, ossia di tipo yfx , e l'operatore "<<" associativo a destra, di tipo xfy , in base alle regole suddette, implicite nel sistema Prolog:

1 >> 2 >> 3 viene considerato come: $(1 >> 2) >> 3$

1 << 2 << 3 viene considerato come: $1 << (2 << 3)$

1 << 2 << 3 viene considerato come: $(1 << 2) << 3$

Si noti che lo specificatore **x** è quello in grado di risolvere l'eventuale ambiguità di un'espressione, in quanto i termini presenti in **x** devono sempre avere una precedenza più bassa rispetto all'operatore **f**.

Il significato dei tipi ammessi per gli operatori prefissi e postfissi può essere visto per analogia con quello degli operatori infissi. Per esempio, se **not** è un operatore prefisso di tipo **fy**, allora **not G** e **not not G** sono due modi corretti per indicare rispettivamente le strutture **not(G)** e **not(not(G))**. Se invece il tipo è **fx**, allora **not G** è ancora corretto, mentre non è accettabile **not not G**. Si ha dunque che **fy** ed **yf** sono associativi, al contrario di **fx** ed **xf**.

Dichiarazione ed uso di operatori.

In Prolog è possibile dichiarare come operatore un funtore di numero di precedenza **P**, associatività **T** e nome **N** effettuando, mediante una direttiva nel programma, una chiamata al predicato predefinito **op**:

```
: - op(P, T, N).
```

Per esempio, per definire l'operatore "**->**" che realizza il costrutto "if-then-else" si dichiara:

```
op(1050, xfy, ->).
```

dopo di che si potrà utilizzare l'operatore con tali precedenza e tipo all'interno del programma.

L'argomento **N** può anche essere una lista di nomi di operatori (eventualmente uno solo) che vengono dichiarati dello stesso tipo e precedenza, come in:

```
: - op(300, yfx, [-,%,&]).
```

```
: - op(1050, xfy, [->]).
```

Il predicato **op** consente di dichiarare più operatori dotati dello stesso nome, ma di differente natura: infissi, prefissi o postfissi. Un qualsiasi operatore, anche predefinito nel sistema, può venire ridefinito per mezzo di una nuova dichiarazione, che annulla così tutte le precedenti definizioni, esplicite od implicite.

L'insieme degli operatori di sistema ed i rispettivi numeri di precedenza variano nelle diverse versioni di Prolog, mentre non mutano i loro tipi; è dunque possibile formulare il seguente prospetto, nel quale sono indicati gli operatori principali e le loro precedenze relative, ma in una scala arbitraria:

```
: - op(250, fx, [?-]).
```

```
: - op(250, xfy, [:-]).
```

```
: - op(230, xfy, [;]).
```

```
: - op(210, xfy, [,]).
```

```
: - op(200, fx, [not]).
```

```

:- op(180,xfy,[.]).  

:- op(160, xfx,[is,=..,==,\==,@<,@>, @=<, @>=,<,>,=:=,=|=,=<,>=,=,\=]).  

:- op(140, yfx,[+,-]).  

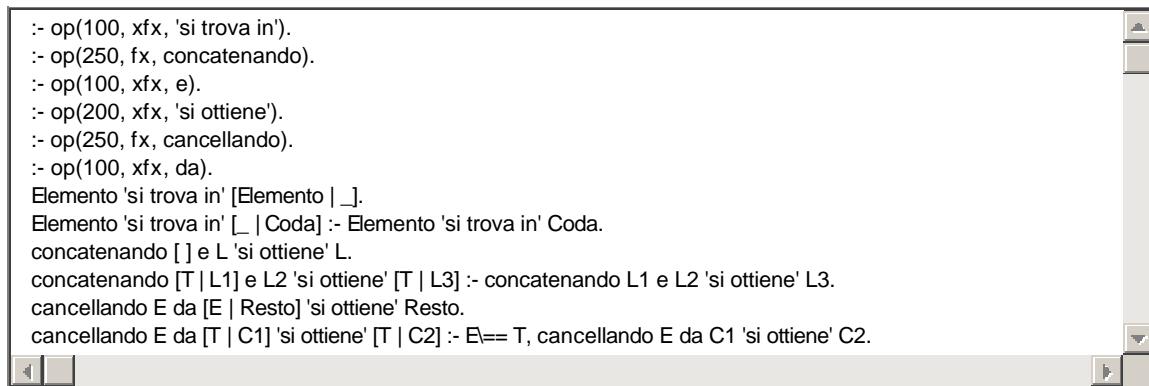
:- op(140, fx,[+,-]).  

:- op(120, yfx,[*/,//]).  

:- op(100, xfx,[mod]).
```

Alcuni di questi operatori sono già stati considerati precedentemente (in particolare, gli operatori ":" e "," fanno parte della sintassi standard delle clausole).

L'uso degli operatori consente di formulare in maniera espressiva la definizione di relazioni. Si vedano come esempio le seguenti procedure:



```

:- op(100, xfx, 'si trova in').  

:- op(250, fx, concatenando).  

:- op(100, xfx, e).  

:- op(200, xfx, 'si ottiene').  

:- op(250, fx, cancellando).  

:- op(100, xfx, da).  

Elemento 'si trova in' [Elemento | _].  

Elemento 'si trova in' [_ | Coda] :- Elemento 'si trova in' Coda.  

concatenando [ ] e L 'si ottiene' L.  

concatenando [T | L1] e L2 'si ottiene' [T | L3] :- concatenando L1 e L2 'si ottiene' L3.  

cancellando E da [E | Resto] 'si ottiene' Resto.  

cancellando E da [T | C1] 'si ottiene' [T | C2] :- E== T, cancellando E da C1 'si ottiene' C2.
```

Esempi di quesiti secondo questa sintassi sono:

?- b 'si trova in' [a,b,c].
?- concatenando [a,b] e X 'si ottiene' [a,b,c,d].
?- cancellando 5 da [1,3,5,7] 'si ottiene' X.

La dichiarazione di nuovi operatori fornisce la possibilità di estendere la sintassi standard del linguaggio, e più generalmente di sviluppare programmi che accettano clausole scritte in una specifica notazione desiderata; ciò rende molto facile sviluppare un'interfaccia di utente congeniale ad un particolare utilizzo del linguaggio.

Funtori e predicati aritmetici.

In tutti i sistemi Prolog l'effettuazione di operazioni aritmetiche viene assicurata da un insieme di procedure predefinite, espresse da predicati aritmetici, che accettano come argomenti espressioni aritmetiche e le valutano, fornendo in uscita i risultati nella forma di variabili istanziate.

Un'espressione aritmetica è o un singolo numero, o un termine costruito a partire da numeri, variabili e funtori valutabili. L'espressione è intera, o reale, se i numeri in questione sono interi, o reali, l'insieme dei numeri rappresentabili varia a seconda delle implementazioni. I principali funtori valutabili sono alcuni degli operatori le cui dichiarazioni sono state indicate nel paragrafo precedente, con il seguente significato, rispettivamente:

```
: - op(140, yfx, [+,-]). /* addizione e sottrazione */  
:- op(140, fx, [+,-]) /* segno positivo e segno negativo */  
:- op(120, yfx, [*,///]). /* moltiplicazione, divisione reale, divisione intera */  
:- op(100, xfx, [mod]). /* resto della divisione intera */
```

Nelle varie versioni di Prolog si trovano, in aggiunta ai precedenti, diversi altri funtori valutabili, corrispondenti a funzioni di esponenziazione, radice quadrata, logaritmo, funzioni trigonometriche, funzioni di arrotondamento e di troncamento di numeri reali, e così via. In alcuni sistemi, come il Prolog/DEC-10, può essere valutata anche una lista di un solo elemento, **[E]**; se **E** è un'espressione aritmetica intera, lo è anche **[E]**, che viene valutata al valore di **E**. Poiché una stringa racchiusa tra doppi apici è semplicemente una lista di interi, risulta così possibile utilizzare un carattere fra doppi apici al posto del suo codice ASCII. Ciò è molto utile per la manipolazione di caratteri: per esempio "**d**" si comporta, all'interno delle espressioni aritmetiche, come la costante intera che è il codice ASCII della lettera **d**, ossia come l'intero **100**, e l'espressione aritmetica "**x**" + "**P**" - "**p**" vale **88**, codice ASCII della lettera **X**.

Al momento della valutazione, ogni variabile presente in un'espressione aritmetica dev'essere istanziata ad un numero o ad un'espressione aritmetica. Le espressioni aritmetiche vengono valutate soltanto quando figurano quali argomenti dei prediciati aritmetici che verranno illustrati nel seguito. Il risultato della valutazione di un'espressione aritmetica è il valore numerico che si ottiene valutando ognuno degli operatori aritmetici che costituiscono l'espressione, cioè effettuando l'operazione appropriata sui risultati ottenuti valutando i suoi argomenti; l'ordine di valutazione è definito dall'associatività e dalla precedenza degli operatori interessati.

Ricerca del risultato di un'espressione aritmetica.

Il processo di unificazione, descritto in precedenza, non valuta espressioni, bensì semplicemente le sottopone a successive sostituzioni simboliche: così, se **X** è istanziata a **+(Y, 1)**, **Y** a **+(Z, 2)** e **Z** a **3**, l'istanziamento finale per **X** è: **+(+3, 2), 1**, e non **6**. Il risultato **6** viene ottenuto solo mediante l'utilizzo del predicato di sistema **is**, che forza la valutazione delle espressioni.

Il predicato **is** è definito come operatore infisso. Il suo argomento sinistro deve essere una variabile (libera od istanziata) od un numero, ed il suo argomento destro un'espressione aritmetica valutabile od un numero; affinché l'espressione sia valutabile, tutte le variabili che compaiono in essa devono essere istanziate.

L'invocazione della meta:

Z is X

comporta la valutazione dell'espressione aritmetica **X**, e riesce o fallisce a seconda che il risultato possa o meno venire unificato con **Z**; si determina un errore di esecuzione se **X** non è un'espressione

aritmetica valutabile, oppure se **Z** non è una variabile od un numero. La meta non ha effetti collaterali e non può venire risoddisfatta. Secondo le convenzioni del Prolog/DEC-10, la procedura **is** può anche valutare una lista di un elemento istanziato ad un'espressione intera.

Alcuni esempi sono:

X is 4 * 4 riesce ed istanzia **X** a **16**;

1 is 4-3 riesce;

1 is 3*4-2 fallisce;

a is 4 - 3 dà errore;

1+1 is 4-2 dà errore;

3 is 4 - X fallisce se **X** è istanziata ad un numero diverso da **1**, dà errore se non è istanziata;

5 is 4 + a dà errore;

40 is [40] riesce.

Si noti che la necessità che l'argomento destro di **is** sia istanziato rende la procedura non reversibile; più in generale i predicati aritmetici, in quanto forzano la valutazione delle espressioni in deroga al normale funzionamento dell'unificazione, comportano uno scostamento dalla semantica dichiarativa usuale. Come esempio consideriamo la seguente ulteriore versione di procedura per il calcolo del fattoriale:

```
fattoriale_rapido(Numer0, Fattoriale) :- fattoriale_rapido_1(1, Numer0, Fattoriale).
fattoriale_rapido_1(N, N, N).
fattoriale_rapido_1(N1, N2, Risultato) :- N1 < N2, Medio is (N1 + N2)//2, Medio_1 is Medio + 1, fattoriale_rapido_1(N1, Medio, R1),
fattoriale_rapido_1(Medio_1, N2, R2), Risultato is R1 * R2.
```

Si tratta di un esempio di procedura deterministica per la quale l'unica risposta possibile viene trovata al termine del ramo più a sinistra dell'albero di ricerca. La procedura ausiliaria, dopo aver calcolato i due numeri interi che meglio approssimano il numero medio fra i primi due argomenti, dà luogo ad una doppia chiamata ricorsiva, i risultati della quale vengono moltiplicati fra loro per ottenere l'uscita richiesta. La procedura costituisce una buona esemplificazione del metodo di "divisione e conquista", nel senso che scinde il calcolo in due sottocalcoli indipendenti; se questi ultimi fossero eseguiti in parallelo, si otterrebbe il risultato più rapidamente, rispetto alle procedure considerate in precedenza.

Confronto dei risultati di un'espressione aritmetica.

I predicati di sistema **=:=**, **=\=**, **<**, **>**, **=<**, **>=** sono definiti come operatori infissi; entrambi i loro argomenti devono essere numeri od espressioni aritmetiche, altrimenti si determina un errore di esecuzione. Detto **p** uno qualsiasi di questi predicati, la meta:

X p Y

forza la valutazione delle espressioni aritmetiche **X** ed **Y**, e riesce se i risultati stanno fra loro nella relazione definita da **p**, altrimenti fallisce; non ha effetti collaterali e non può venire risoddisfatta. Le relazioni definite da questi predicati sono:

X =:= Y (i valori di **X** e di **Y** sono uguali);

X =\= Y (i valori di **X** e di **Y** non sono uguali);

X < Y (il valore di **X** è minore del valore di **Y**);

X > Y (il valore di **X** è maggiore del valore di **Y**);

X =< Y (il valore di **X** è minore o uguale al valore di **Y**);

X >= Y (il valore di **X** è maggiore o uguale al valore di **Y**).

Per esempio:

I1 + I2 + I3 =:= 3 * I2 riesce;

I2*0 =\= 12*1 riesce;

2-3<0 riesce;

4 >= 3 riesce;

15+2<15+1 fallisce.

Gli operatori **<**, **>**, **=<**, **>=** possono essere utilizzati fra espressioni intere e reali, in una combinazione qualsiasi. Gli operatori **=:=** e **=\=** possono essere utilizzati soltanto fra espressioni intere, in quanto non ha senso sottoporre a verifica di uguaglianza o disuguaglianza dei numeri reali, poiché la rappresentazione in virgola mobile di un numero non è precisa.

Note bibliografiche.

La procedura [fattoriale_rapido](#) è stata ripresa da Conery e Kibler (1983).

Sommario.

Lo studente ha ora acquisito una maggiore consapevolezza dell'opportunità di estendere il linguaggio delle clausole di Horn e le dimostrazioni per risoluzione. Se infatti in linea di principio l'aritmetica può essere trattata in modo esclusivamente simbolico, questo sarebbe molto poco conveniente ai fini pratici.

In Prolog, quindi, la valutazione di espressioni aritmetiche si affianca all'unificazione come meccanismo di computazione.

11. Predicati di controllo e di ingresso/uscita

Dove si presentano alcuni predicati di sistema che consentono ulteriori possibilità di specificazione del controllo, ed i prediciati predefiniti per l'ingresso e l'uscita dei dati.

Predicati predefiniti di controllo.

Tutte le implementazioni di Prolog dispongono di prediciati predefiniti di comodo, che sono cioè utili nella pratica ma risultano o ridondanti, nel senso che comunque il loro effetto può essere ottenuto usando le usuali caratteristiche del linguaggio, o necessari solo per ragioni di efficienza o comunque da un punto di vista procedurale. Tra questi vi sono i prediciati di controllo descritti in questo paragrafo, ad eccezione del seguente predicato cali, che ha un ruolo diverso.

Il predicato "call".

Se **M** è un termine accettabile entro il corpo di una clausola, ossia una meta eseguibile, oppure una variabile istanziata a tale termine, la meta **call(M)** viene eseguita esattamente come se il termine **M** comparisse testualmente al posto di **call(M)**, e quindi riesce o fallisce secondo che **M** riesce o fallisce. Se **M** è una congiunzione di mete, questa va racchiusa in una coppia di parentesi aggiuntive, per evitare ambiguità sul numero di argomenti. Se invece **M** non soddisfa le condizioni suddette, la chiamata fallisce (producendo eventualmente un messaggio d'errore).

Ad esempio, date le clausole:

pred(a).

pred(b).

pr(c).

pr(d).

si ha:

?- call(pred(a)). riesce;

?- call(pred(c)). fallisce;

?- call(pred(a), pr(c)). fallisce;

?- call ((pred(a), pr(c))). riesce;

?- call(pred(X)). istanzia **X** ad **a** e, con ritorno indietro, a **b**;

?- call(X). con **X** istanziata ad esempio a **pr(Y)**, istanzia **Y** a **c** e, con ritorno indietro, a **d**.

Il predicato cali trasforma quindi un termine in una meta; esso consente di eseguire una meta non predeterminata ma variabile dinamicamente a seconda di come è istanziato il suo argomento (come

se si usasse un predicato con funtore variabile). Questo predicato è quindi meta- logico, nel senso che opera sul linguaggio logico stesso.

Equivalentemente, come pura variante sintattica all'utilizzo di cali, una variabile può comparire essa stessa come meta (in un quesito o nel corpo di una clausola), alle stesse condizioni dell'argomento di cali. Una variabile che compare in un programma in tale ruolo è detta una metavariabile, nel senso che sta al posto non di un termine come di solito, ma di un predicato. Ad esempio, sono equivalenti i seguenti quesiti:

?- Z is 1 + 1.

?- call(Z is 1 + 1).

?- X = (Z is 1 + 1), call(X).

?- X = (Z is 1 + 1), X.

dove " $=$ " è un predicato predefinito che istanzia la variabile a sinistra al termine a destra.

La procedura seguente istanzia la variabile a secondo argomento all'atomo vero in caso di soddisfacimento della meta che compare al primo, ed a falso in caso contrario:

```
booleano(Meta, vero) :- call(Meta), !.  
booleano(_, falso).  

```

oppure:

```
booleano(Meta, vero) :- Meta, !.  
booleano(_, falso).  

```

La procedura:

una_volta(M) :- call(M), !.

oppure:

una_volta(M) :- M, !.

I.

consente l'esecuzione della meta **M** in maniera deterministica; costituisce quindi una possibile realizzazione del predicato **once**.

Il predicato di disgiunzione ";".

Il predicato di sistema ";" è definito quale operatore infisso che richiede, come argomenti, due termini che rappresentano mete eseguibili. La chiamata:

G1 ; G2

riesce se la meta **G1** riesce oppure, nel caso che **G1** fallisce, se la meta **G2** riesce; in caso contrario fallisce. Il predicato ";" esprime quindi la disgiunzione tra le due mete fornite come argomenti.

Le alternative rappresentate, all'interno di una regola, dall'operatore ";" subiscono, nel caso di presenza del predicato di taglio, il consueto trattamento: dopo il soddisfacimento della meta "!", tutte le scelte fatte a partire dall'invocazione della meta genitrice vengono automaticamente fissate. Per esempio, la clausola:

a :- b, c, (d, ! ; e, f).

è del tutto equivalente alle tre clausole:

a :- b, c, a_1.

a_1 :- d, !.

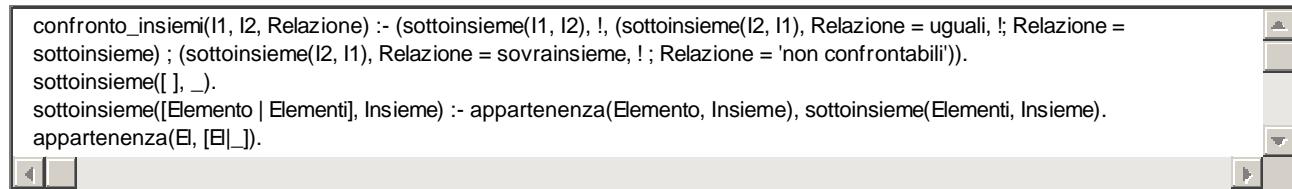
a_1 :- e, f.

L'operatore di disgiunzione ";", disponibile in molte versioni di Prolog, rappresenta unicamente un elemento di convenienza sintattica; infatti il suo uso può sempre venire ovviato o con le seguenti definizioni generali (usando call o metavariabili):

G1 ; G2 :- call(G1). (oppure **G1 ; G2 :- G1.**)

G1 ; G2 :- call(G2). (oppure **G1 ; G2 :- G2.**)

ovvero mediante una differente realizzazione della procedura nella quale compare. Per esempio, la procedura confronto_insiemi, può venire così riformulata (in modo deterministico) usando l'operatore ";" per la specificazione delle alternative:



```
confronto_insiemi(I1, I2, Relazione) :- (sottoinsieme(I1, I2), !, (sottoinsieme(I2, I1), Relazione = uguali, !; Relazione = = sottoinsieme) ; (sottoinsieme(I2, I1), Relazione = sovrainsieme, !; Relazione = 'non confrontabili')).  
sottoinsieme([ ], _).  
sottoinsieme([Elemento | Elementi], Insieme) :- apparteness(Elemento, Insieme), sottoinsieme(Elementi, Insieme).  
apparteness(El, [El|_]).
```

Come si vede, tale formulazione risulta più compatta ma alquanto meno leggibile dell'altra.

I predicati "true" e "fail".

Nel corso della normale esecuzione di un programma Prolog sono possibili il successo od il fallimento di ogni meta invocata. I predicati **true** e **fail**, da utilizzarsi quali procedure senza argomenti, servono rispettivamente a forzare il successo ed il fallimento della loro meta genitrice.

La meta **true** riesce ogni volta che viene invocata, non presenta effetti collaterali e non può venire risoddisfatta in caso di ritorno indietro. Ad esempio, la seguente procedura:

```
ricerca_numero :- ricerca(X), X>10, nl, write('Numero trovato.'); true.
```

usa il predicato **ricerca**, che si suppone definito altrove, per trovare un numero: se tale numero è maggiore di **10** viene emessa una segnalazione, in caso contrario viene invocata **true**; in tal modo questa digiunzione di mete riesce sempre.

La disponibilità del predicato di sistema **true** è da intendersi anch'essa come una semplice convenienza offerta all'utente; infatti è sempre possibile evitarne l'uso con una opportuna ridefinizione.

La procedura definita sopra si può per esempio modificare come segue:

```
ricerca_numero :- ricerca(X), X > 10, nl, write('Numero trovato.').
```

```
ricerca_numero.
```

Si osservi che la naturale definizione del predicato **true** consiste semplicemente nella clausola:

```
true.
```

La procedura **fail** fallisce ad ogni sua invocazione, impedendo il successo della congiunzione di mete in cui compare e forzando di conseguenza l'attivazione del meccanismo di ritorno indietro. Se non fosse predefinita si potrebbe realizzare semplicemente non rappresentandola con alcuna clausola nel programma, ovvero sostituendola con qualsiasi meta per la quale non esiste alcuna clausola, nella base di dati, in grado di soddisfarla.

Viene usata essenzialmente nella congiunzione "**!, fail**" ed in combinazione con il predicato **repeat**.

Il predicato "repeat".

Quando viene invocato come meta, il predicato senza argomenti **repeat** riesce sempre, come **true**, ma - al contrario di questo - può venire risoddisfatto un numero arbitrario di volte, generando così una successione potenzialmente infinita di scelte mediante ritorno indietro.

Se non fosse un predicato di sistema potrebbe essere definito nel modo seguente:

```
repeat.
```

```
repeat :- repeat.
```

Con questa definizione, all'atto della prima invocazione di **repeat** viene utilizzata la prima clausola; successivamente, se la meta **repeat** viene interessata dal ritorno indietro, la seconda clausola - utilizzata come alternativa - porta ad una nuova invocazione di **repeat**, che riesce con l'impiego della prima clausola, lasciando nuovamente a disposizione - per un eventuale ulteriore ritorno

indietro - la seconda. L'ordine relativo delle due clausole è cruciale: se il fatto fosse preceduto dalla regola, infatti, si avrebbe un tipico caso di ricorsione infinita.

Il predicato **repeat** è quindi sempre non deterministico; la sua funzionalità può considerarsi in un certo senso antitetica a quella del taglio, dal momento che rende disponibili ulteriori rami nell'albero di dimostrazione della meta principale, anziché eliminarli.

Si noti che non è possibile effettuare un'iterazione da **N1** a **N2** con una procedura del tipo:

iterazione_sbagliata(N1, N2) :- repeat, write(N1), M is N1 + 1, (M == N2, ! ; fail).

perché il ritorno indietro distrugge gli istanziamenti, cosicché **write** scriverà non i valori da **N1** a **N2**, ma sempre e indefinitamente il valore cui è istanziato **N1** al momento della chiamata della procedura. Il predicato **repeat** può essere usato solo con al suo interno sottomete che producono nuovi istanziamenti ogni volta che vengono chiamate.

Predicati predefiniti di ingresso/uscita.

I predicati di ingresso/uscita rispondono all'esigenza, comune alla programmazione tradizionale, di introdurre dati d'ingresso durante l'esecuzione del programma e di ottenere delle risposte nelle forme e nei modi più opportuni nelle differenti applicazioni. D'altra parte, mentre l'ingresso/uscita non pone particolari questioni di natura teorica a nessun linguaggio procedurale, esso costituisce un problema per Prolog. Infatti per tali operazioni Prolog utilizza predicati extra-logici, cioè predicati che non hanno significato logico ma sono impiegati solo per produrre come effetti collaterali la lettura o la scrittura di caratteri o di termini.

Predicati per la gestione di files.

Le operazioni di ingresso e di uscita avvengono nei confronti di files, che possono essere o files su memoria di massa, aventi nomi definiti dall'utente, oppure il file standard associato al terminale, che ha nome predefinito **user**. Ogni file può essere aperto in ingresso (in lettura) oppure in uscita (in scrittura), ad eccezione del file **user** che può essere sia in lettura che in scrittura.

In un dato momento diversi files possono trovarsi o venire aperti in ingresso od in uscita, ma ad ogni istante uno solo di essi è il flusso corrente di ingresso, ed uno solo il flusso corrente di uscita. Il flusso corrente隐式 è **user**, a meno che, e fino a quando, non viene modificato mediante i predicati predefiniti che seguono, che hanno la caratteristica comune di non potere essere risoddisfatti in caso di ritorno indietro.

I predicati **see(F)** e **tell(F)**, invocati con **F** istanziato a un nome di file (eventualmente, **user**), comportano che il file menzionato diviene il flusso corrente, rispettivamente di ingresso o di uscita (non hanno effetto se lo è già). Per entrambi, se **F** non è istanziato, si determinerà un errore; se **F** è istanziato ad un nome di file che non esiste, con il predicato **see** si avrà un errore, mentre con il predicato **tell** esso verrà creato. Un file diverso da **user** viene aperto alla prima occorrenza di **see(F)** o **tell(F)** nel programma, e resta disponibile per operazioni di ingresso e di uscita fino alla sua chiusura, che si attua invocando i predicati **seen** e **told** rispettivamente quando esso è il flusso corrente, con il conseguente ripristino del flusso standard **user**. Un'eventuale nuova apertura di un file di uscita dopo la sua chiusura con **told** determinerà la cancellazione del contenuto preesistente.

Il nome del flusso di ingresso o di uscita corrente può essere conosciuto invocando le procedure **seeing(F)** o **telling(F)**; in entrambi i casi l'effetto è di istanziare la variabile **F** al nome del file in questione. Possono venire anche utilizzate con il loro argomento già istanziato; in tal caso riusciranno o falliranno a seconda che esso coincida o no con il nome del flusso corrente.

In alcuni sistemi è disponibile la procedura **exists(F)** (o **exists_file(F)**), che riesce se il file il cui nome istanzia la variabile **F** esiste, e fallisce in caso contrario; genera un errore se l'argomento non è istanziato ad un atomo, o comunque se non specifica un nome di file valido.

La combinazione dei suddetti predicati consente di operare su due (o più) files. Per esempio, con la procedura:

```
apertura_elaborazione_chiusura(F1, F2) :- seeing(X), see(F1), lettura, elaborazione,
telling(Y), tell(F2), scrittura, seen, told, see(X), tell(Y).
```

i files **F1** e **F2** vengono aperti, rispettivamente in lettura ed in **scrittura**, per le operazioni definite dalle procedure **lettura**, **elaborazione** e **scrittura**; al termine vengono ripristinati, quali flussi correnti, i flussi **X** ed **Y** in precedenza attivi per l'ingresso e per l'uscita.

Particolare attenzione va posta quando si intende scrivere su più di un file, come per esempio in:

```
tell(file), write(...), tell(alter_file), write(.. .), tell(user), write(...), tell(file), write(...), told.
```

In questo caso il programma scrive alternatamente su tre files diversi, non avendo richiamato **told** se non alla fine delle operazioni di scrittura. Finché un file non viene esplicitamente chiuso, le operazioni di uscita su di esso si susseguono continuando ogni volta dal punto in cui si sono fermate alla precedente invocazione del predicato **tell** per quel file.

Diversi altri predicati per la gestione di files possono essere presenti nelle varie versioni di Prolog. In tutti i casi, usando i predicati predefiniti, si possono costruire altre procedure di ingresso/uscita più sofisticate. Ad esempio si possono definire procedure interattive, come la seguente, che richiedono all'utente il nome del file interessato e lo verificano, per prevenire il tentativo di apertura di files che non esistono:

```
verifica_nome_file(File, File) :- atom(File), exists(File), !.
```

```
verifica_nome_file(File, Nuovo_file) :- atom(File), !, nl, write('Il file indicato non esiste.'), nl,
write('Segnalare un nuovo file: '), nl, read(F), verifica_nome_file(F, Nuovo_file).
```

```
verifica_nome_file(File, Nuovo_file) :- nl, write('Il nome del file indicato '), nl, write('è''
scorretto.'), nl, write('Segnalare, quale nome, un atomo!'), nl, read(F), verifica_nome_file(F,
Nuovo_file).
```

Analogamente si può definire una procedura che evita la sovrascrittura di files già esistenti.

Ingresso ed uscita di termini.

L'assegnamento dei flussi correnti di ingresso e di uscita consente di iniziare le operazioni di lettura e scrittura di caratteri e termini su essi, mediante i predicati predefiniti che seguono; le relative mete non possono venire risoddisfatte in caso di ritorno indietro.

La meta **read(T)** effettua la lettura di un termine dal flusso corrente di ingresso (mostrando un prompt). Viene abitualmente utilizzata con **T** come variabile libera, e riesce se **T** unifica con il successivo termine in ingresso, che deve essere delimitato da un punto seguito da uno spazio o da <return>; il punto non diviene parte del termine e viene rimosso. La sintassi del termine dev'essere in accordo con le dichiarazioni di operatori (sia quelle implicite che quelle decise dall'utente) in quel momento valide. Se ciò che viene letto non è un termine valido, viene generato un errore sintattico, e la meta fallisce. Se una chiamata di **read(T)** determina il raggiungimento della fine del flusso corrente di ingresso, **T** viene unificata con l'atomo end-of-file, corrispondente al carattere di end-of-file; eventuali ulteriori chiamate al predicato **read**, relative allo stesso flusso di ingresso, provocheranno un fallimento ed una segnalazione di errore.

La meta **nl** determina una nuova riga sul flusso corrente di uscita. Si può definire la procedura **salto_righe** per ottenere spaziature di **N** righe:

```
salto_righe(0).
salto_righe(N) :- nl, N1 is N - 1, salto_righe(N1).
```

La meta **write(T)** riesce sempre, e scrive il termine **T** sul flusso corrente di uscita secondo le dichiarazioni di operatori in quel momento in vigore. Se **T** contiene una o più variabili istanziate, queste vengono scritte nella forma dei termini a cui sono istanziate se **T** contiene delle variabili libere, ciascuna di esse viene scritta nella forma "**_N**", dove **N** è un numero intero (generato dal sistema) che la identifica univocamente rispetto alle altre. Variabili che si trovano fra loro in condivisione verranno indicate con lo stesso numero di sistema. Nessun atomo viene racchiuso tra apici.

Ad esempio, per scrivere gli elementi di una lista uno per riga:

```
scrittura_elementi([]).
scrittura_elementi([Testa | Coda]) :- write(Testa), nl, scrittura_elementi(Coda).
```

La meta **writeq(T)** scrive sul flusso corrente di uscita il termine **T**, secondo le dichiarazioni di operatori in vigore, in maniera simile a **write(T)**, salvo che i nomi degli atomi, dei funtori e degli identificatori in genere (anche quelli che coincidono con nomi di operatori) vengono automaticamente racchiusi dal sistema entro apici singoli, per rendere il risultato accettabile come ingresso al predicato **read**. Per esempio, la meta:

tell(f), writeq('Franco'), told.

ha come effetto collaterale la scrittura, sul file **f**, dell'atomo '**Franco**'; con **write('Franco')** si otterebbe invece la scrittura **Franco**, che verrebbe considerata, in una eventuale successiva lettura, una variabile non istanziata anziché un atomo.

Una procedura di copiatura di un file termine per termine, in cui si presuppone che i termini siano seguiti da un punto e disposti su righe separate, è la seguente:

```
copiatura_termini(File_ingresso, File_uscita) :- seeing(Flusso_corrente_ingresso),  
telling(Flusso_corrente_uscita), see(File_ingresso), tell(File_uscita), repeat, read(Termine),  
(Termine == end-of-file, !, seen, told, see(Flusso_corrente_ingresso),  
tell(Flusso_corrente_uscita) ; writeq(Termine), write(., nl, fail).
```

Se il **Termine** letto non è l'atomo end-of-file (cioè che avviene quando si effettua un tentativo di lettura oltre l'ultimo termine presente nel file), viene scritto sul **File_di_uscita**, e la chiamata di **fail** forza un ritorno indietro sino a **repeat** (**read** e **write**, infatti, non possono venire interessati dal ritorno indietro). Quando la variabile istanziata **Termine** risulta letteralmente uguale ad end-of-file, il taglio determina la cancellazione di tutta la storia dei ritorni indietro forzati in precedenza; successivamente vengono chiusi i due files di ingresso e di uscita e vengono riattivati i flussi che erano aperti prima dell'invocazione della procedura.

Ingresso ed uscita di caratteri.

I predicati di sistema per l'ingresso e l'uscita di caratteri hanno come argomenti numeri interi che rappresentano i codici ASCII dei caratteri; anch'essi riescono una volta sola e non possono venire risoddisfatti in fase di ritorno indietro.

La meta **get(C)** ha come argomento il codice ASCII del successivo carattere stampabile diverso da spazio (blank) in ingresso; **C** può essere un intero oppure una variabile non istanziata. Nel primo caso, riesce se il successivo carattere in ingresso unifica tale argomento, fallisce altrimenti; nel secondo caso, l'argomento viene istanziato al valore del codice ASCII del successivo carattere in ingresso.

L'effetto collaterale di questa meta è dunque che i caratteri fino al successivo carattere stampabile compreso, presenti sul flusso corrente di ingresso, vengono da questo rimossi, sia che la meta riesca, sia che fallisca. Se **C** non è una variabile non istanziata o un intero, la meta fallisce e nulla viene rimosso dal flusso di ingresso. Il carattere di end_of_file (il cui codice ASCII è 26) conta, agli effetti del predicato **get**, come un carattere stampabile. Qualsiasi tentativo di lettura di un carattere oltre la fine del file genera un errore di esecuzione, a meno che il flusso corrente di ingresso non sia **user**.

La meta **get0(C)** ha come argomento il codice ASCII del successivo carattere (stampabile o non stampabile) in ingresso; al contrario di **get**, dunque, opera su qualsiasi carattere. **C** dev'essere una variabile non istanziata o un intero. L'effetto collaterale di questa meta, qualunque sia il suo esito, è la rimozione dal flusso corrente di ingresso del carattere successivo. **get0** leggerà anche il carattere di end-of-file, ma un tentativo di lettura dopo di esso genererà un errore di esecuzione.

La procedura seguente legge un carattere: se è **s** o **n** istanzia il suo argomento a tale carattere, altrimenti invita l'utente a replicare con **s** o con **n** ed effettua una nuova chiamata a se stessa:

```

lettura_si_no(C) :- lettura_carattere(C), (C == s ; C == n), !.
lettura_si_no(C) :- nl, write('Risposta errata; digitare "s" o "n":'), lettura_si_no(C).
/* La metà lettura_carattere(C) riesce se C è un carattere diverso da blank, dopo il quale viene letto il carattere <return> (codice ASCII 10); eventuali altri caratteri presenti fra il primo e <return> vengono eliminati (il predicato predefinito name dà nel secondo argomento il codice ASCII del carattere al primo argomento) */

```

La metà **put(C)** emette sul flusso corrente di uscita il carattere il cui codice ASCII è **C**. L'argomento **C** dev'essere istanziato, e può essere un'espressione aritmetica intera; se l'espressione non è corretta, viene generato un errore di esecuzione. Esempi sono:

put(97) scrive **a**;

put(98 - 32) scrive **n**;

put("X") scrive **X**;

put([88]) scrive **X**;

put(x(y, z)) genera un errore;

put("p" + "C" - "c") scrive **P**.

Usato in combinazione con **get**, **put** consente ad esempio di visualizzare i codici ASCII relativi a ciascun carattere stampabile:

```

/*
PROCEDURA: stampa_codici_ASCII(M, N).
P: stampa tutti i caratteri stampabili il cui codice ASCII è compreso fra M ed N.
*/
stampa_codici_ASCII(Ultimo, Ultimo) :- stampa_carattere(Ultimo).
stampa_codici_ASCII(M, N) :- stampa_carattere(M), M1 is M+1,
stampa_codici_ASCII(M1, N).

```

Si voglia procedere alla copiatura di un file su un altro file, considerando i due files come stringhe di caratteri, e leggendo e scrivendo carattere per carattere. Si può realizzare un ciclo iterativo con la tecnica del ritorno indietro forzato da una sottometa: ad ogni passo elementare del ciclo ha luogo la ricezione di un carattere dal flusso di ingresso e la sua trasmissione sul flusso di uscita. Al raggiungimento del carattere di end-of-file la sottometa che aveva in precedenza attivato il ritorno indietro viene soddisfatta, e la procedura termina con la chiusura dei due flussi di ingresso/uscita e la riattivazione di quelli precedentemente aperti:

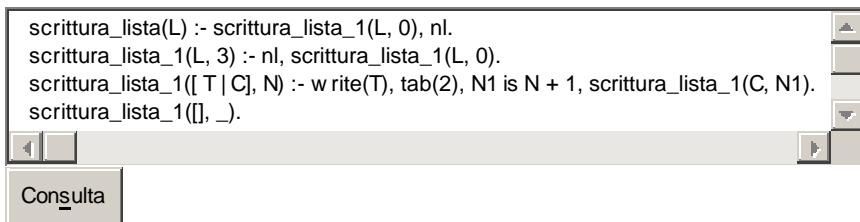
```

copiatura_caratteri(File_ingresso, File_uscita) :- seeing(Flusso_corrente_ingresso),
telling(Flusso_corrente_uscita), see(File_ingresso), tell(File_uscita), repeat, get0(Carattere),
put(Carattere), Carattere == 26, !, seen, told, see(Flusso_corrente_ingresso),
tell(Flusso_corrente_uscita).

```

È possibile utilizzare questa procedura con il primo argomento, il secondo od entrambi istanziati a **user**. Ad esempio è possibile aprire un file e scrivere su esso da tastiera: basterà istanziare la variabile **File_ingresso** a **user**, mentre si dovrà istanziare **File_uscita** al file da creare.

Nella metà **tab(N)**, l'argomento va istanziato ad un valore intero; vengono emessi **N** spazi sul flusso corrente di uscita. **N** può essere un'espressione aritmetica intera; se non è un intero od un'espressione intera, viene generato un errore di esecuzione. Ad esempio, il seguente programma scrive gli elementi di una lista separati da due spazi sulla riga, con al più tre elementi per riga:



```

scrittura_lista(L) :- scrittura_lista_1(L, 0), nl.
scrittura_lista_1(L, 3) :- nl, scrittura_lista_1(L, 0).
scrittura_lista_1([T | C], N) :- write(T), tab(2), N1 is N + 1, scrittura_lista_1(C, N1).
scrittura_lista_1([], _).

```

The screenshot shows a Prolog interface window with the above code. At the bottom left is a button labeled "Consulta".

Naturalmente la procedura può essere parametrizzata ad **N** elementi per riga separati da **M** spazi.

A partire dai predicati di sistema si possono realizzare con facilità altre procedure per la scrittura di termini secondo le modalità desiderate. Un esempio è il seguente:

/*

PROCEDURA: scrittura_complessa(S).

D: S è una successione di termini connessi dall'operatore &.

P: scrive S sul flusso corrente di uscita.

T: verdi, in una selezione a più casi mutuamente esclusivi.

C: Ogni termine presente nella successione viene scritto mediante la procedura predefinita **writeq**, eccetto che - se il termine è @T - verrà invocato quale meta il termine T.

*/

:op(180, xfx, &), op(100, fx, @), op(100, xf, %), op(160, xfy, tb), op(100, xf, /), op(100, fx, @\$).

**scrittura_complessa(Termine_1 & Termine_2):- scrittura_complessa(Termine_1),
scrittura_complessa(Termine_2).**

scrittura_complessa(X tb N):- tab(N), scrittura_complessa(X).

scrittura_complessa(@ Termine):- call(Termine).

scrittura_complessa(Termine %):- scrittura_complessa(Termine), nl, tab(10).

scrittura_complessa(@\$ Termine):- writeq(Termine).

```
scrittura_complessa(/):- !,nl, tab(10).
```

```
scrittura_complessa(Termine):- write(Termine).
```

```
writeln(Termine):-scrittura_complessa(Termine %).
```

a(b).

la meta:

```
?- scrittura_complessa(/ & 'Questa' tb 12 & / & 'e" una stampa di prova' & / & 'per illustrare  
l'"utilizzo' % & 'della procedura' % & @$ 'SCRITTURA_COMPLESSA' & : & / & 'la meta  
a(X) e" soddisfatta' & / & 'da ' & @ a(X) & 'X = ' & X & '.' & / ).
```

consente di ottenere la seguente uscita:

Questa è una stampa di prova

per illustrare l'utilizzo

della procedura

'SCRITTURA_COMPLESSA':

la meta a(X) è soddisfatta

da X = b.

Si può ampliare a piacimento la procedura, per adattarla a specifiche esigenze, semplicemente aggiungendo nuove clausole con nome di predicato **scrittura_complessa** e definendo, in corrispondenza, gli operatori che in esse compaiono.

Per facilitare l'utilizzo di programmi la cui uscita è costituita unicamente da effetti collaterali, senza istanziamento di variabili, può essere conveniente consentire di continuare l'esecuzione con nuovi ingressi senza ritornare al livello principale. Lo scopo può essere ottenuto con procedure come la seguente, dove si ipotizza che **elaborazione**, chiamata con il suo unico argomento istanziato, emetta i propri risultati invocando opportuni predicati di scrittura:

```
programma(Dati_ingresso) :- elaborazione(Dati_ingresso), scrittura_complessa(/ & 'L"utente  
vuole continuare' & / & 'con altri ingressi? (s / n): ' & /), lettura_si_no(Risposta),  
gestione_risposta(Risposta).
```

```
gestione_risposta(s) :- scrittura_complessa(/ & 'Nuovi dati:' & / & /),  
read(Nuovi_dati_Ingresso), programma(Nuovi_dati_ingresso)
```

```
gestione_risposta(n) :- scrittura_complessa(/ & 'Fine elaborazione.' & /).
```

Tutti i predicati descritti in questo paragrafo sono disponibili anche in due possibili varianti: scritti con il prefisso **tty**, si riferiscono solo al file **user**, indipendentemente da quali siano i flussi correnti (di ingresso e di uscita) in quel momento; usati con un argomento aggiuntivo, operano sul file d'uscita da esso indicato.

Note bibliografiche.

Esempi di uso estensivo di predicati di ingresso/uscita si trovano nelle due realizzazioni di un editor proposte in Kluzniak e Szpakowicz (1985) ed in Sterling e Shapiro (1986).

Sommario.

Lo studente può ora sviluppare programmi Prolog la cui interazione con l'utente avviene non solo mediante le modalità dell'interprete, ma anche secondo quelle previste nel programma stesso in base alle specifiche esigenze.

I predicati di sistema considerati in questo capitolo estendono il linguaggio delle clausole di Horn per soddisfare le usuali necessità di ingresso ed uscita dei dati.

12. Predicati meta-logici

Dove si presentano i predicati predefiniti per la manipolazione di termini, e si esemplificano diverse procedure, costruibili a partire da essi, per trattare programmi, o loro parti, come dati. Questo insieme di predicati predefiniti in Prolog è dedicato ad operare su termini non in quanto denotanti oggetti del mondo rappresentato, ma in quanto costrutti sintattici che possono essere analizzati o trasformati sintatticamente. Tali predicati sono definiti in base alle caratteristiche del linguaggio nel quale i termini sono espressi, e riguardano quindi relazioni sul linguaggio logico e non sul mondo da esso descritto; in questo senso sono detti meta-logici.

A differenza dei predicati extra-logici, i predicati meta- logici non comportano generalmente effetti collaterali e presentano un contenuto dichiarativo, che si pone tuttavia ad un livello differente rispetto al significato dichiarativo associato alle relazioni tra oggetti.

Verifica del tipo di un termine.

In tutte le versioni di Prolog, anche se con possibili varianti, è disponibile un insieme di predicati predefiniti per verificare il tipo di un termine. Le mete invocate usando uno dei predicati seguenti riescono o falliscono in dipendenza dal tipo di termine che figura quale loro unico argomento (od al quale esso è istanziato); non presentano alcun effetto collaterale, né possono determinare errori di esecuzione o venire risoddisfatte in caso di ritorno indietro:

var(V) V è una variabile non istanziata;

nonvar(T) T è una variabile istanziata oppure un termine diverso da una variabile;

atomic(C) C è una costante (atomo o numero);

atom(A) A è un atomo;

number(N) N è un numero (intero o reale);

integer(I) I è un numero intero;

real(R) R è un numero reale.

Si noti che **atomic** può essere definito in termini di **atom** e **number**, con:

atomic(X) :- atom(X).

atomic(X) :- number(X).

oppure, equivalentemente, con:

atomic(X) :- atom(X) ; number(X).

In maniera analoga **number** può essere definito in termini di **integer** e **real**.

È facile ampliare la gamma dei predicati per la classificazione dei termini, utilizzando quelli predefiniti per realizzarne altri. Una chiamata alla seguente procedura, per esempio, termina con successo se la struttura in questione è una lista non vuota i cui elementi sono tutti atomi:

```
lista_atomi(Lista) :- Lista \==[], lista_atomi_1(Lista).
```

```
lista_atomi_1([]).
```

```
lista_atomi_1([Testa|Coda]) :- atom(Testa), lista_atomi_1(Coda).
```

La disponibilità di predicati predefiniti per la verifica del tipo dei termini consente di sviluppare procedure il cui comportamento non sia affidato solo ai meccanismi predefiniti del linguaggio, ma sia controllato nei modi voluti. La seguente procedura, per esempio, informa sul motivo del suo eventuale fallimento.

```
/*
```

PROCEDURA: controllo_tipo(Tp, T).

D: il termine T è di tipo Tp.

P: controllo_tipo(<, <): verifica il tipo del termine a secondo argomento, fallendo con una segnalazione di errore se non coincide con quello indicato a primo argomento.

T: tutti rossi, simulano strutture di selezione.

U: scrittura_complessa/1

```
*/
```

```
controllo_tipo(intero, T):-integer(T), !.
```

```
controllo_tipo(intero, T):-segnalazione(intero, T),!,fail.
```

```
controllo_tipo(reale, T):-real(T),!.
```

```
controllo_tipo(reale, T):-segnalazione(reale, T),!,fail.
```

```
controllo_tipo(variabile, T):-var(T),!.
```

```
controllo_tipo(variabile, T):-segnalazione(variabile, T),!,fail.
```

```
controllo_tipo(atomo, T):-atom(T),!.
```

```
controllo_tipo(atomo, T):-segnalazione(atomo, T),!,fail.
```

```
controllo_tipo('lista vuota', T):-lista_vuota(T),!.
```

```
controllo_tipo('lista vuota', T):-segnalazione('lista vuota', T),!,fail.
```

```
controllo_tipo('lista non vuota', T):-lista_non_vuota(T).
```

```

controllo_tipo('lista non vuota', T):-segnalazione('lista non vuota', T),!,fail.

controllo_tipo(struttura, T):-struttura(T),!.

controllo_tipo(struttura, T):-segnalazione(struttura, T),fail.

segnalazione(Tipo_richiesto, T):-scrittura_complessa(/ & 'Il termine ' & T & ' non e'' un(a) ' & Tipo_richiesto & '.' & /
& 'Viene attivato il ritorno indietro.' & /).

lista_vuota([]).

lista_non_vuota([_|_]).

struttura(T):-arg(1,T,_).

```

I predicati di verifica del tipo dei termini possono essere usati per rafforzare un certo modo d'uso di una procedura (controllando con **var** e **nonvar** quali argomenti sono liberi e quali istanziati), allo scopo di evitare che una procedura contenente predicati predefiniti generi un errore in esecuzione quando gli argomenti di questa non si confanno agli usi prescritti (soprattutto per i predicati aritmetici) ed anche per escludere questi ultimi, per esempio renderli reversibili quando tali non sono. Un uso combinato per questi scopi è illustrato dalla seguente procedura:

```

/*
PROCEDURA: somma_o_sottrazione(X, Y, Z).
D: la somma dei numeri X ed Y è Z.
P1: somma_o_sottrazione(<, <, >): istanzia Z alla somma di X e Y.
P2: somma_o_sottrazione(<, >, <): istanzia Y alla differenza di X rispetto a Z.
P3: somma_o_sottrazione(>, <, <): istanzia X alla differenza di Y rispetto a Z.
C: I controlli su quali argomenti sono istanziati a numeri sono necessari per selezionare la clausola appropriata.
*/
somma_o_sottrazione(X,Y,Z):-number(X),number(Y),Z is X+Y.
somma_o_sottrazione(X,Y,Z):-number(X),number(Z),Y is Z-X.
somma_o_sottrazione(X,Y,Z):-number(Y),number(Z),X is Z-Y.

```

Ad esempio, il quesito:

?-somma_o_sottrazione(A,1,5), somma_o_sottrazione(1,B,10),somma_o_sottrazione(B,A,C).

fornisce per **C** il valore **13**.

Confronto di termini.

I predicati per il confronto di termini sono predefiniti come operatori binari infissi, e vanno utilizzati per controllare se due termini sono fra loro uguali o disuguali. Questi predicati considerano le variabili non istanziate come termini che possono essere confrontati, ma non possono in alcun modo istanziare tali variabili e non vanno mai utilizzati per effettuare confronti di tipo aritmetico od unificazioni. Essi non presentano effetti collaterali, non possono venire risoddisfatti mediante ritorno indietro, né possono produrre errori di esecuzione.

Tali predicati di confronto presuppongono il seguente ordinamento lessicografico totale standard dei termini Prolog:

- variabili, in ordine di definizione (e non in relazione ai nomi delle variabili);
- numeri, da - "infinito" a + "infinito";
- atomi, in ordine alfabetico (secondo l'ordinamento dei codici ASCII dei caratteri);
- strutture, ordinate secondo i seguenti criteri, in ordine decrescente di priorità: numero di argomenti, nome del funtore principale, ordine degli argomenti (da sinistra a destra).

Ad esempio, la seguente lista di termini rispetta tale ordinamento:

[V, -256, -2, 0, 24, atomo, termine, A = B, struttura(0, 1), struttura(0, 1), struttura(1, 2)].

La meta:

X == Y (rispettivamente, X \== Y)

riesce se i termini che, al momento della chiamata, istanziano le variabili **X** ed **Y** sono (rispettivamente, non sono) tra loro letteralmente identici, cioè, alternativamente, sono:

- lo stesso intero;
- lo stesso reale;
- lo stesso atomo;
- variabili che si trovano in condivisione prima dell'esecuzione della meta;
- strutture aventi lo stesso funtore e lo stesso numero di argomenti, e tali che gli argomenti che si corrispondono siano tra loro, ricorsivamente, identici; in particolare, variabili in posizioni equivalenti nei due termini devono essere tra loro identiche.

L'esecuzione della meta non dà luogo ad unificazione; di conseguenza gli istanziamenti, parziali o totali, di **X** ed **Y** non vengono mutati. Esempi di applicazione sono:

term == term riesce;

1 == 1 riesce;

var(X), var(Y), fun(X) == fun(Y) fallisce;

fun(X) == fun(X) riesce;

var(X), X == v fallisce.

La meta: **T1 @< T2** (rispettivamente, **T1 @> T2**, **T1 @=< T2**, **T1 @>= T2**)

riesce se il termine **T1** precede (rispettivamente, è preceduto, non segue, non precede) il termine **T2** nell'ordinamento lessicografico predefinito.

Ad esempio, la seguente procedura termina con successo se la stringa di caratteri a primo argomento è seguita, nell'ordine lessicografico di Prolog, da quella a secondo argomento; le due stringhe sono rappresentate mediante liste:

confronto_stringhe([],[_|_]).

confronto_stringhe([Car_1|_],[Car_2|_]):-Car_1 @< Car_2.

confronto_stringhe([Car_1|Cars_1],[Car_1|Cars_2]):-confronto_stringhe(Cars_1,Cars_2).

Unificazione di termini.

La meta:

X=Y

cerca di unificare le due variabili **X** ed **Y**. Se, al momento della chiamata, **X** ed **Y** sono completamente disistanziate, vengono poste in condivisione fra loro. Se sono parzialmente istanziate (a strutture), il soddisfacimento della meta può comportare un maggior grado di istanziamento dell'una o dell'altra, o la condivisione di alcune delle loro variabili. Se una di esse è istanziata e l'altra non lo è, anche quest'ultima viene istanziata, ed allo stesso termine della prima. Se sono entrambe pienamente istanziate, la meta riesce se sono istanziate allo stesso termine.

I comportamenti descritti sono conseguenza immediata del meccanismo di unificazione; se non fosse predefinito il predicato "**=**" dopo averlo dichiarato come operatore, potrebbe venire definito nel. programma mediante la clausola:

T = T.

Alcuni esempi sono i seguenti:

a = a riesce;

a = b fallisce;

var(X), X = a riesce ed istanzia **X** ad **a**;

var(X), var(Y), X = a, Y = b, X = Y fallisce;

var(X), var(Y), X = a, Y = X, Y = a riesce;

var(X), var(Y), X = Y, X = [1, 2] riesce ed istanzia **X** e **Y** a **[1,2]**;

var(X), var(Y), X = f(a,b,c,Y), Y = w(1) riesce ed istanzia **X** a **f(a, b, c, w(1))** e **Y** a **w(1)**;

var(X), var(Y), X = Y, fun(X) == fun(Y) riesce;

var(X), var(Y), X = Y, Y = v, X == v riesce.

La meta:

X\=Y

riesce se le variabili **X** ed **Y** non sono in condivisione fra loro e - di conseguenza - non sono sottoponibili ad unificazione.

Si noti la differenza tra gli operatori "**=**" e "**==**". Il primo può venire utilizzato per una delle seguenti funzionalità:

- istanziare una variabile ad un atomo o ad una struttura (parzialmente o completamente istanziata);
- porre in condivisione due variabili, una delle quali istanziata e l'altra no, oppure entrambe non istanziate;
- sottoporre a verifica una variabile, per accettare se sia istanziata ad un certo termine o meno.

Il secondo operatore può invece venire utilizzato soltanto per stabilire l'uguaglianza letterale fra due termini, che potranno eventualmente essere rappresentati da variabili istanziate. Di conseguenza, l'unica possibilità di utilizzare indifferentemente l'uno o l'altro dei due predicati si ha quando si desidera effettuare una verifica sull'istanziamento di una variabile; per esempio, le mete:

Numero = 10

Numero == 10

sono destinate a venire entrambe soddisfatte od a fallire entrambe.

```
/*
PROCEDURA: min_max(L, Min, Max).
D: Min e Max sono il minimo e il massimo degli eventuali elementi numerici della lista di termini qualsiasi L.
P1: min_max(<, <, <): verifica della relazione.
P2: min_max(<, >, <): fornisce il minimo e il massimo degli eventuali elementi numerici di L; se non ve ne sono, viene segnalato che sono indefiniti.
T: i tagli hanno un doppio effetto: il primo simula la struttura "if-then-else" sulla condizione var, ed impedisce la ricerca di un'altra soluzione mediante le altre clausole (che darebbero in uscita le variabili non istanziate); il secondo e l'ultimo portano all'applicazione dell'ultima clausola sia quando T (testa della lista corrente) è un numero compreso tra Min_p e Max_p (minimo provvisorio e massimo provvisorio correnti) sia quando non è un numero, quindi sostituiscono la presenza nell'ultima clausola di una triplice condizione.
Sarebbe analogamente possibile sostituire le condizioni nonvar sulle ultime tre clausole con altri tagli, anch'essi rossi, sulle clausole precedenti.
*/
min_max(L,Min,Max):-min_max_1(L,_,_,Min,Max).
/* Prima chiamata, con lista vuota: le variabili d'uscita sono libere, e viene loro assegnato valore indefinito */
min_max_1([],Min,Max,Min,Max):-var(Min),var(Max),!,Min='indefinito',Max='indefinito'.
/* Scansione della lista terminata: minimo e massimo provvisori diventano definitivi */
min_max_1([],Min,Max,Min,Max).
/* Prima chiamata, con primo elemento di lista numerico: di vento il minimo e massimo provvisori */
min_max_1([T|C],Min_p,Max_p,Min,Max):-var(Min_p),var(Max_p),number(T),!,min_max_1(C,T,T,Min,Max).
/* Prima chiamata, con primo elemento di lista non numerico: viene tralasciato */
min_max_1([_|C],Min_p,Max_p,Min,Max):-var(Min_p),var(Max_p),min_max_1(C,Min_p,Max_p,Min,Max).
/* L'elemento numerico in testa alla lista è minore del minimo corrente e quindi lo sostituisce */
min_max_1([T|C],Min_p,Max_p,Min,Max):-nonvar(Min_p),nonvar(Max_p), number(T),T<Min_p,!,min_max_1(C,T,Max_p,Min,Max).
/* L'elemento numerico in testa alla lista è maggiore del massimo corrente e quindi lo sostituisce */
min_max_1([T|C],Min_p,Max_p,Min,Max):-nonvar(Min_p),nonvar(Max_p),
number(T),T>Max_p,!,min_max_1(C,Min_p,T,Min,Max).
/* L'elemento in testa è numerico e compreso tra minimo e massimo correnti oppure non è numerico: viene tralasciato */
min_max_1([_|C],Min_p,Max_p,Min,Max):-nonvar(Min_p),nonvar(Max_p), min_max_1(C,Min_p,Max_p,Min,Max).
```

In alcune applicazioni può avere interesse operare su una lista per conoscere quali sono tutti e soli i suoi componenti elementari, ignorando eventuali strutture di lista presenti all'interno della lista stessa. Questa funzionalità è assicurata dalla procedura **appiattimento_lista(L1, L2)**, in cui **L1** è una lista d'ingresso che può contenere sottoliste a qualsiasi livello di innestamento, ma contenenti costanti come elementi di ultimo livello, ed **L2** è una lista d'uscita che contiene, nello stesso ordine, tutte le costanti presenti nella lista originaria. Per esempio, il seguente quesito ottiene risposta positiva:

?- appiattimento_lista([atomo, [elemento, 1], a, [c, [d, [e,f]]]], [atomo, elemento, 1,a, c, d,e,f]).

e la meta:

?- appiattimento_lista([1,2,3, [4,5,[[6],7]],8], X).

termina con successo istanziando **X** alla lista [1,2,3,4,5,6,7,8].

```
/*
PROCEDURA: appiattimento_lista(L1, L2).
D: L2 è la lista contenente tutte e sole le costanti presenti in L1.
P1: appiattimento_lista(<, <): verifica la relazione.
P2: appiattimento_lista(<, >): fornisce la lista di costanti L2 ottenuta eliminando da L1 eventuali strutture di sottolista.
U: concatenazione/3.
*/
appiattimento_lista([Testa|Coda],Lista):-appiattimento_lista(Testa,T), appiattimento_lista(Coda,C),concatenazione(T,C,Lista).
appiattimento_lista([E],E):-atomic(E),E\==[].
appiattimento_lista([],[]).
concatenazione([],L,L).
concatenazione([T|L1],L2,[T|L3]):-concatenazione(L1,L2,L3).
```

Il programma utilizza una doppia ricorsione sugli elementi della lista a primo argomento. Alla chiarezza della sua interpretazione dichiarativa non corrisponde però analoga efficienza.
Naturalmente è possibile sostituire la chiamata di [concatenazione](#) con l'uso di un accumulatore, ottenendo:

```
appiattimento_lista_1(L1,L2):-appiattimento_lista_2(L1,[],L2).
appiattimento_lista_2([T|C],Acc,L):-appiattimento_lista_2(C,Acc,L1),appiattimento_lista_2(T,L1,L).
appiattimento_lista_2(T,C,[T|C]):-atomic(T),T\==[].
appiattimento_lista_2([],L,L).
```

L'utilizzo di una pila (rappresentata con una lista) consente di realizzare una versione più efficiente (in questo caso si presuppone che nella lista in ingresso non vi siano sottoliste vuote):

```
/* Si inizializza la pila alla lista vuota */
appiattimento_lista_3(L1,L2):-appiattimento_lista_4(L1,[],L2).
/* Se la testa della lista è una lista non vuota, si inserisce la coda in cima alla pila e si appiattisce la testa */
appiattimento_lista_4([T|C],Stack,L):-T = [_|_],appiattimento_lista_4(T,[C|Stack],L).
/* Se la testa della lista è una costante, diversa dalla lista vuota, si aggiunge alla lista di uscita, e si appiattisce la coda */
appiattimento_lista_4([T|C],Stack,[T|L])->atomic(T),T\==[],appiattimento_lista_4(C,Stack,L).
/* Se la lista è vuota e la pila no, si estrae la cima della pila e la si appiattisce */
appiattimento_lista_4([],[],Stack,L):-appiattimento_lista_4(T,Stack,L).
/* Se la lista e la pila sono vuote, si termina */
appiattimento_lista_4([],[],[]).
```

Il predicato "name".

Per l'accesso ai caratteri di un atomo (e, in alcuni sistemi, di un intero) è disponibile la procedura predefinita:

name(X, L)

dove **X** e **L** sono, o devono divenire, rispettivamente un atomo (o un intero), e la lista dei codici ASCII dei caratteri che formano **X** (se l'atomo che istanzia **X** è racchiuso tra apici singoli, questi non faranno parte dei codici in **L**). Può essere usata con uno dei due argomenti istanziato e l'altro come variabile libera (è reversibile). La meta non presenta effetti collaterali e non può essere risoddisfatta in caso di ritorno indietro. Si determina un errore di esecuzione se gli argomenti non soddisfano le condizioni richieste.

Ad esempio:

name (atomo, X) riesce ed istanzia **X** alla lista [97, 116, 111, 109, 111];

name(X, [80,114,111,108,111,103]) riesce ed istanzia **X** all'atomo Prolog;

name(lista, "lista") riesce;

name('is an atom', "is an atom") riesce;

name(1, X) riesce ed istanzia **X** a **[49]** (nei sistemi in cui è ammesso un intero);

name(abc, a(b)) fallisce;

name([], []) riesce;

name(Y, X) fallisce, con un messaggio d'errore;

name(:- , [58,45]) riesce;

name('Ascii', [65, X, 99, 105, Y]) riesce ed istanzia **X** a **115**, **Y** a **105**;

name(X, "?") riesce ed istanzia **X** a **?**.

Per esempio, la relazione **comincia_con(Atomo, Carattere)** permette di controllare se l'**Atomo** a primo argomento comincia con il **Carattere** a secondo argomento (entrambi devono essere istanziati):

```
comincia_con(Atomo,Carattere):-name(Carattere,[Codice]),name(Atomo,[Codice|_]).
```

La seguente procedura trasforma un atomo con soli caratteri letterali minuscoli in un atomo con soli caratteri maiuscoli:

```
da_minuscolo_a_maiuscolo(Minuscolo,Maiuscolo):-atom(Minuscolo), name(Minuscolo,L),a_maiuscolo(L,L1),name(Maiuscolo,L1).
a_maiuscolo([],[]).
a_maiuscolo([T|C],[T1|C1]):-T>=97,T=<122,T1 is T-32,a_maiuscolo(C,C1).
a_maiuscolo([T|C],[T|C1]):-T>=65,T=<90,a_maiuscolo(C,C1).
```

Analogamente si può definire la trasformazione inversa.

Per trasformare un atomo in ingresso nella lista dei suoi caratteri:

```

da_atomo_a_lista(Atomo,Lista_caratteri):-atom(Atomo),name(Atomo,Lista_codici), da_atomo_a_lista_1(Lista_codici,Lista_caratteri).
da_atomo_a_lista_1([],[]).
da_atomo_a_lista_1([T|C],[T1|C1]):-name(T1,[T]),da_atomo_a_lista_1(C,C1).

```



Per la concatenazione di due atomi si può usare la procedura:

```

concatenazione_costanti(C1,C2,C):-atomic(C1),atomic(C2), name(C1,L1),name(C2,L2),concatenazione(L1,L2,L),name(C,L).
concatenazione([],L,L).
concatenazione([T|L1],L2,[T|L3]):-concatenazione(L1,L2,L3).

```

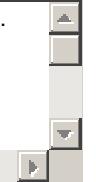


La procedura seguente accerta se una parola data in ingresso è o meno un palindromo, cioè si legge allo stesso modo in entrambi i versi (ad esempio, madam è un palindromo):

```

palindromo(Parola):-name(Parola,Lista_codici), inversione(Lista_codici,Lista_codici), nl, write('La parola e" un palindromo.').
palindromo(_):-nl, write('La parola non e" un palindromo.').
inversione(L1,L2):-inv(L1,[],L2).
inv([],L,L).
inv([T|C],L1,L2):-inv(C,[T|L1],L2).

```

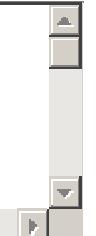


Se l'inversione della lista dei codici ASCII dei caratteri che compongono la parola porta a riottenere la lista stessa, la parola è un palindromo; in caso contrario il ritorno indietro, non potendo portare al risoddisfacimento di inversione o di name (che sono entrambe deterministiche), attiva la seconda clausola. Una versione alternativa è:

```

palindromo_1(Parola):-name(Parola,Lista_codici), palindromo_2(Lista_codici,[]).
palindromo_2(X,X).
palindromo_2([_|C],C).
palindromo_2([T|C],C1):-palindromo_2(C,[T|C1]).
inversione(L1,L2):-inv(L1,[],L2).
inv([],L,L).
inv([T|C],L1,L2):-inv(C,[T|L1],L2).

```



Essa si basa sull'osservazione che una parola è un palindromo se la sua prima metà è l'inverso della seconda, e quindi non è necessario invertire l'intera lista; dichiarativamente **palindromo_2(L1, L2)** significa che si ha un palindromo concatenando **L1** all'inversa di **L2**.

La procedura che segue fornisce in uscita nel secondo argomento una classificazione del carattere specificato (fra apici) nel primo argomento:

```
tipo_carattere(Carattere,Tipo):-name(Carattere,[Codice]),tipo(Codice,Tipo).  
tipo(Codice,cifra):-Codice>=48,Codice=<57,!.  
tipo(Codice,'lettera minuscola'):-Codice>=97,Codice=<122,!.  
tipo(Codice,'lettera maiuscola'):-Codice>=65,Codice=<90,!.  
tipo(C,parentesi):-C=40;C=41;C=91;C=93;C=123;C=125,!.  
tipo(C,punteggiatura):-C=33;C=44;C=46;C=58;C=59;C=63,!.  
tipo(Codice,'carattere simbolico'):-Codice>=33,Codice=<126.
```

Verifica, accesso e formazione di strutture.

Alcuni predicati di sistema permettono la creazione di strutture o l'accesso al loro funtore principale ed alle loro componenti, nonché l'individuazione della loro molteplicità. Permettono anche l'esame della struttura e del contenuto delle clausole che compongono un programma.

Il predicato "functor".

La meta:

functor(T, F, N)

termina con successo se il termine **T** ha funtore principale **F** e molteplicità **N**. **T** può essere una struttura, un atomo od una variabile; **F** può essere un atomo o, a patto che **N** valga **0**, un numero (un atomo viene considerato come una struttura di molteplicità **0**). La meta non ha effetti collaterali e non può venire risoddisfatta in caso di ritorno indietro; se gli argomenti in ingresso non soddisfano le condizioni sopra descritte, viene prodotto un errore di esecuzione.

La procedura **functor** è utilizzabile in due modi principali. Se **T** è istanziata ad un termine non variabile, permette di esaminare il suo funtore principale ed il numero di argomenti, istanziando ad essi, rispettivamente, **F** ed **N**. Se, viceversa, **T** è una variabile libera e **F** ed **N** sono rispettivamente istanziati ad un atomo e ad un intero non negativo, o ad un intero ed a **0**, viene creata una struttura, e viene unificata con **T**: il risultato della chiamata è di istanziare **T** al termine più generale che ha il funtore principale indicato.

I suddetti usi sono schematizzabili come segue:

functor(<, >, >) dà il funtore **F** ed il numero di argomenti **N** di un termine **T** assegnato;

functor(>, <, <) crea una struttura con funtore **F** assegnato e numero assegnato **N** di argomenti (non istanziati).

Ricordando dal § 6.4 che un predicato deterministico per un certo modo è tale anche per ogni modo ottenuto dal precedente facendo diventare d'ingresso uno o più parametri d'uscita, risultano utilizzabili anche i modi di uso, derivati dai precedenti:

functor(<, <, <) verifica che **T** abbia funtore principale **F** e numero di argomenti **N**;

functor(<, <, >) verifica che **T** abbia funtore **F**;

functor(<, >, <) verifica che **T** abbia **N** argomenti.

Per esempio:

functor(a(b, c), a, 2) riesce;

functor(a(1, 2, 3), a, 4) fallisce;

functor([1,2,3], , 2) riesce;

functor(X, fun, 2) riesce ed istanzia **X** a **fun(_,_)**;

functor(struttura(X, Y, c), F, G) riesce ed istanzia **F** a **struttura**, e **G** a **3**; **X** e **Y** rimangono non istanziate;

functor(El, _, 1) riesce se il termine **El**, qualunque sia il suo funtore principale, ha un solo argomento, e fallisce altrimenti;

functor(Termine, Nome, _) riesce se **Termine** ha per funtore **Nome**, qualunque sia il suo numero di argomenti.

Il predicato **functor** è utile anche per analizzare la struttura delle clausole Prolog; in questa interpretazione il primo argomento rappresenta la clausola, il secondo rappresenta il predicato o l'operatore che costituisce il funtore della clausola ed il terzo la molteplicità della stessa. Per esempio, la meta:

functor(a(b, X) :- c, :-, 2).

riesce, ed **X** rimane non istanziata.

Il predicato "arg".

La meta:

arg(I, T, A)

termina con successo se l'**I**-esimo argomento della struttura **T** è **A**. **I** e **T** devono essere, o diventare, rispettivamente un intero positivo ed una struttura (non un atomo). Gli argomenti vengono numerati a partire da **1**, ed **I** deve essere non superiore al numero delle componenti di **T**. Anche questa meta non ha effetti collaterali e non può venire risoddisfatta in fase di ritorno indietro; fallisce (o produce un errore di esecuzione) se le condizioni iniziali non sono soddisfatte.

Il modo d'uso principale è:

arg(<, <, >) la variabile **A** viene istanziata all'**I**-esimo argomento di **T**;

che ammette come caso particolare:

arg(<, <, <) verifica la relazione.

Alcuni esempi sono i seguenti:

arg(2, struttura(a, b, c), S) riesce ed istanzia S a **b**;

arg(3, a(g, h), X) fallisce (oppure provoca un errore di esecuzione);

arg(2, f(a, b), a) fallisce;

arg(3, f(a, b, c), c) riesce;

arg(2, f(a, X, c), b) riesce ed istanzia **X** a **b**;

arg(4, f(g, h, i, j(k)), V) riesce ed istanzia **V** a **j(k)**;

arg(2, a(X, Y), Z) riesce e pone in condivisione le variabili **Y** e **Z**, senza tutta via istanziarle;

arg(1, [a, b, c], X) riesce ed istanzia **X** ad **a**;

arg(2, [a,b,c], L) riesce ed istanzia **L** alla lista **[b,c]**;

read(T), arg(1, T, X) riesce ed istanzia **X** al primo argomento del termine **T**.

Anche **arg**, come già si è detto per **functor**, può essere utilizzato per la rappresentazione di una clausola: in tal caso il primo argomento rappresenterà il particolare argomento della clausola a cui si è interessati (la testa o la coda), o - se la clausola è unitaria - la componente così indicata, il secondo sarà la clausola stessa ed il terzo rappresenterà il valore dell'argomento così definito. Per esempio:

?- arg(2, appartenenza(X, [a, b,c,d]), Y).

X =_12004 (è un numero di sistema)

Y = [a, b, c, d]

Usando **arg** si può verificare se un termine è una struttura:

```
struttura(T) :- arg(1, T, _).
```

Usando **arg** e **functor** si può verificare se un termine è completamente istanziato:

```
chiuso(T):-atomic(T).
chiuso(T):-nonvar(T),functor(T,_,N),chiuso_1(N,T).
chiuso_1(N,T):-N > 0,arg(N,T,A),chiuso(A),N1 is N-1, chiuso_1(N1,T).
chiuso_1(0,_).
```

Un altro esempio è il seguente:

```

/*
PROCEDURA: stessi_argomenti(T1, T2, N).
D: le strutture T1 e T2 hanno ordinatamente argomenti uguali a partire dall'argomento di posizione N-esima.
P: stessi_argomenti(<, <, <): verifica la relazione; in particolare, fallisce se i due termini hanno diversa molteplicità o se il terzo argomento non è un numero od è un numero superiore alla molteplicità dei due termini.
C: La condizione limite fa terminare con successo la procedura quando il contatore supera la molteplicità dei due termini.
*/
stessi_argomenti(T1,T2,Posizione) :- functor(T1,_,N),functor(T2,_,N), stessi_argomenti_1(T1,T2,Posizione,N).
stessi_argomenti_1(T1,T2,Posizione,N):-Posizione =< N, arg(Posizione,T1,A1),arg(Posizione,T2,A2),A1==A2, Pos is Posizione+1,stessi_argomenti_1(T1,T2,Pos,N).
stessi_argomenti_1(_____,N,N).

```

Il predicato "`=..`".

Il predicato di sistema "`=..`" (si legge "univ") è un operatore infisso da invocarsi nella forma:

T=.. L

i cui due argomenti sono o divengono, rispettivamente, una struttura od un atomo, ed una lista. La meta riesce se **L** è la lista la cui testa è l'atomo che costituisce il funtore principale di **T** e la cui coda è la lista degli argomenti di quel funtore nel termine **T**. La meta non ha effetti collaterali e non può venire risoddisfatta in caso di ritorno indietro; produce un errore se gli argomenti non soddisfano le condizioni sudette. I modi d'uso sono:

`<=.. >` crea la lista;

`>=.. <` crea la struttura avente come funtore la testa della lista (che dev'essere un atomo od un intero) e come argomenti (non istanziati) i successivi elementi della lista;

e, di conseguenza:

`<=.. <` verifica della relazione.

Fallisce se usata con:

`>=.. >`

Esempi sono:

n-1=.. [-n,1] riesce;

a =.. [a] riesce;

a(b,c,X) =.. [W,Y,c,Z] riesce ed istanzia **W** ad **a**, **Y** a **b** e pone in condivisione le variabili **X** e **Z**, senza però istanziarle;

fun(1, 2, 3) =.. [fun,1,2,3] riesce;

fun(1, 2, X) =.. [fun,A,B,Y] riesce ed istanzia **A** ad **1**, **B** a **2** e pone in condivisione le variabili **X** ed **Y**, che rimangono non istanziate;

T =.. [pred,1,m,m,-1] riesce ed istanzia **T** alla struttura **pred(1, m, m - 1)**;

[ab,c,def] =.. [.,ab,[c,def]] riesce.

Si osservi che ognuno dei predicati **functor**, **arg** e **=..** può essere definito mediante uno degli altri (la loro compresenza è dunque ridondante, per semplice convenienza):

```
functor(T, F, N) :- T=.. [F|L], lunghezza(L, N).
```

```
arg(N, S, A) :- S=.. [_|L], ennesimo(L, N, A).
```

```
: op(200, xfx, =..).
```

```
S=.. [F|L] :- lunghezza(L, N), functor(S, F, N).
```

La procedura seguente permette di aggiungere altri argomenti (specificati come lista) ad una struttura, senza cambiare il suo funtore né gli argomenti preesistenti:

```
aggiunta_argomenti(Struttura,Altri_argomenti,Nuova_struttura):- Struttura=..Funtore_e_argomenti,
concatenazione(Funtore_e_argomenti,Altri_argomenti,Funtore_e_nuovi_arg), Nuova_struttura=..Funtore_e_nuovi_arg.
concatenazione([],L,L).
concatenazione([T|L1],L2,[T|L3]):-concatenazione(L1,L2,L3).
```

Altri esempi sono i seguenti:

```
/*
PROCEDURA: argomenti(T, L, N).
D: il termine T (strutturato o no) ha lista di argomenti L e numero di argomenti N.
P1: argomenti(<, <, <): verifica della relazione.
P2: argomenti(<, <, >): ricerca del numero di argomenti del termine in ingresso.
P3: argomenti(<, >, <): ricerca della lista degli argomenti del termine in ingresso.
P4: argomenti(<, >, >): ricerca della lista e del numero degli argomenti del termine in ingresso. il molteplice funzionamento è dovuto alla flessibilità d'uso della procedura arg.
*/
argomenti(Term,Lista_arg,Num_arg):- var(Lista_arg),Term=..[_|Lista_arg],argomenti_1(Term,Lista_arg,0,Num_arg).
argomenti(Term,Lista_arg,Num_arg):- nonvar(Lista_arg),argomenti_1(Term,Lista_arg,0,Num_arg).
argomenti_1(_,[],Num_arg,Num_arg).
argomenti_1(Term,[Argom|Resto_arg],K,N):- K1 is K+1,arg(K1,Term,Argom),argomenti_1(Term,Resto_arg,K1,N).

/*
PROCEDURA: innestamento_argomenti(T, L).
D: L è la lista di tutti gli argomenti costanti presenti entro il termine T, a qualsiasi livello di innestamento.
P1: innestamento_argomenti(<, >): verifica della relazione.
P2: innestamento_argomenti(<, >): ricerca della lista degli argomenti costanti presenti entro il termine T, a qualsiasi livello di innestamento.
T: simula la struttura "if-then-else".
U: concatenazione/3
*/
innestamento_argomenti([],[]).
innestamento_argomenti(T,[T]):-atomic(T)!.
innestamento_argomenti(T,Lista_uscita):- T=..[_|Lista_termini],innestamento(Lista_termini,Lista_uscita).
innestamento(Lista_termini,Lista_uscita):- innestamento_1(Lista_termini,[],Lista_uscita).
innestamento_1([],L,L).
innestamento_1([Termino|Termini],L,Lista_uscita):- innestamento_argomenti(Termino,Lista_per_termino),
concatenazione(L,Lista_per_termino,Lista_provvisoria), innestamento_1(Termini,Lista_provvisoria,Lista_uscita).
concatenazione([],L,L).
concatenazione([T|L1],L2,[T|L3]):-concatenazione(L1,L2,L3).
```

Il predicato "**=..**" è particolarmente utile nella creazione di mete complesse, e consente di invocare mete il cui funtore non è conosciuto al momento della formulazione del quesito. Un caso tipico è quello in cui, essendo le variabili **Fun**, **X**, **Y** e **Z** istanziate, viene formata una nuova struttura con la meta:

S=..[Fun,X,Y,Z]

e questo termine viene poi invocato come meta: **call(S)**. Si osservi che nella maggior parte delle implementazioni Prolog è da considerarsi scorretto l'uso di una variabile - sia pure istanziata - quale simbolo di predicato; non è dunque ammessa una scrittura del tipo: **call(Fun(X, Y, Z))**.

La procedura [ordinata](#), può essere generalizzata, per verificare l'ordinamento di liste di termini qualsiasi utilizzando un secondo argomento, che specifica il nome di un operatore che esprime una relazione d'ordine predefinita o definita nel programma:

```
ordinata_1([],_).
ordinata_1([_|_],_).
ordinata_1([X,X|Y],_):-ordinata_1([X|Y],_).
ordinata_1([X,Y|Z],Operatore):- Termine=..[Operatore,X,Y],call(Termine),ordinata_1([Y|Z],Operatore).
```

Ad esempio:

ordinata_1([1,2,3], <) riesce;

ordinata_1([1,2,3], >) fallisce;

ordinata_1([a,b,c], @<) riesce.

La seguente procedura riesce se tutti gli elementi di una lista assegnata a secondo argomento godono di una certa proprietà, specificata da un predicato unario fornito quale primo argomento:

```
proprieta_elemento(P,[Elemento|Coda]):- Meta=..[P,Elemento],call(Meta),proprieta_elemento(P,Coda).
proprieta_elemento(_,[]).
/* clausole aggiunte al fine di provare il programma */
p(a).
p(b).
```

Una variante è data dalla seguente relazione:

corrispondenza(Predicato, L1, L2)

verificata se gli elementi della lista **L2** sono posti in corrispondenza con quelli della lista **L1** da un insieme di regole specificate tramite il **Predicato** a primo argomento:

corrispondenza(_,[],[]).

corrispondenza(Predicato,[T1|C1],[T2|C2]):- Struttura=..[Predicato,T1,T2],call(Struttura),
corrispondenza(Predicato,C1,C2).

Ad esempio, dato il predicato :

corr(a, x).

corr(b y).

corr(c, z).

si ha:

```
corr(a,x).
corr(b,y).
corr(c,z).
```

```
corrispondenza([],[]).  
corrispondenza(Predicato,[T1|C1],[T2|C2]):- Struttura=..[Predicato,T1,T2],call(Struttura), corrispondenza(Predicato,C1,C2).
```

il quesito:

?- corrispondenza(corr, [a,b,c], L).

fornisce la risposta **L = [x, y, t]**.

Note bibliografiche.

Sterling e Shapiro (1986) confrontano tre tipi di uguaglianza fra termini in programmazione logica unificabilità (predicato $=$) varianza alfabetica ed uguaglianza letterale (predicato \equiv) intendendo per la seconda che due termini sono uguali a meno della ridefinizione delle variabili in uno di essi. Mostrano quindi come in generale, l'unificabilità è l'uguaglianza più debole quella letterale la più forte e la varianza alfabetica è intermedia mentre per termini chiusi esse coincidono.

Ancora Sterling e Shapiro (1986) presentano una procedura, ground, che termina con successo se il suo unico argomento è completamente istanziato. Sia in Sterling e Shapiro (1986) che in Clocksin e Mellish (1981) sono discusse le possibili realizzazioni di un predicato come [appiattimento lista](#), da loro chiamato **flatten**.

In Kowalski (1979a) e in Hogger (1984) è trattato il problema dei palindromi. In Kluzniak e Szpakowicz (1985) compare una procedura analoga a [stessi argomenti](#).

Sommario.

Le possibilità offerte dai predicati meta-logici di definire procedure che manipolano i termini del linguaggio consentono la costruzione di programmi che operano su altri programmi, cioè di strumenti che possono costituire un ambiente di programmazione più ricco del solo interprete Prolog.

13. La negazione

Dove si affronta il problema della rappresentazione di informazioni negative, si introduce la nozione di negazione per fallimento e si considerano le modalità con cui questa è usualmente realizzata in Prolog.

Nell'[Interpretazione dichiarativa](#) si è visto come un programma logico **P** può essere utilizzato, mediante la regola d'inferenza della risoluzione, per dedurre informazione "positiva", ossia per dimostrare, dato un quesito **q**, che **q** è una conseguenza logica di **P**. Tale regola d'inferenza non può però essere usata per dedurre informazione "negativa", cioè per dimostrare, dato un quesito negato **non q**, che **non q** è conseguenza logica di E. Infatti, se un predicato chiuso **q** è una conseguenza logica, non si può dimostrare **non q**; e se **q** non è conseguenza logica, non lo è neanche **non q**.

Per esempio, consideriamo il programma consistente dei seguenti quattro fatti:

```
giocatore(sergio).
giocatore(franco).
allenatore(enzo).
allenatore(guido).  
[Navigation icons: back, forward, search, etc.]
```

Il fatto **non giocatore(enzo)** non è conseguenza logica del programma, come non lo è il fatto **giocatore(enzo)**.

Il **non** usato sopra - inteso come classico operatore logico di negazione, il cui significato è di rendere falsa la proposizione a cui è anteposto - non è esprimibile mediante una clausola di Horn, perché per esempio:

p(X, Y) se (non q(X) e r(Y))

con lì significato: "per ogni X, Y, **p(X, Y)** è vero se **q(X)** è falso e **r(Y)** è vero", non è una clausola di Horn, perché è equivalente a **(p(X, Y) o q(X)) se r(Y)**, che non è una clausola di Horn.

Tuttavia, in diversi problemi di programmazione logica può essere necessario, od utile, esprimere la negazione, sia nel formulare un quesito con cui chiedere se qualche dato non compare nella base di dati, che nel definire condizioni negative nel corpo delle clausole.

L'ipotesi del mondo chiuso.

Un modo per affrontare il problema consiste nel fare appello ad una regola d'inferenza speciale, detta ipotesi del mondo chiuso (closed world assumption), secondo la quale se un predicato chiuso **p** non è conseguenza logica di un programma, allora si inferisce **not p**. Si è qui usato **not** per distinguere questo tipo di negazione da quello classico precedentemente indicato con **non**, in quanto queste due forme non hanno lo stesso significato, come si vedrà meglio nel seguito.

Nell'esempio precedente, con questa regola può essere inferito il fatto **not giocatore(enzo)** in quanto **giocatore(enzo)** non è conseguenza logica del programma. In altri termini, se non risulta

possibile effettuare la dimostrazione di un predicato chiuso, la negazione di quel predicato è assunta come vera, ossia viene ammessa come risultato del quesito negato.

Si ricorderà che in generale la risposta **no** del sistema Prolog ad una meta non attesta la falsità di questa, ma piuttosto la sua non dimostrabilità dati i fatti e le regole presenti nella base di dati. Così, per esempio, ponendo il quesito:

?- giocatore(enzo).

si ottiene la risposta **no**.

Il sistema Prolog dispone di una regola di inferenza, aggiuntiva rispetto alla confutazione per risoluzione, basata sull'ipotesi del mondo chiuso; è pertanto ammesso entro un quesito l'uso della negazione **not**, intesa come non dimostrabilità. Pertanto la meta:

?- not giocatore(enzo).

ottiene risposta positiva.

Questa forma di negazione può essere vista come equivalente ad aggiungere implicitamente ai prediciati della base di dati tutte le negazioni ditali prediciati. Nella maggior parte dei casi non sarebbe possibile l'aggiunta esplicita ditali negazioni nella base di dati, in quanto il numero difatti negativi relativi ad un certo dominio può essere di gran lunga superiore a quello dei fatti positivi. Inoltre le regole negative, che ovviamente avrebbero la stessa struttura delle corrispondenti regole positive, risulterebbero altamente ridondanti rispetto a queste ultime.

Equivalentemente, la negazione come non dimostrabilità può essere intesa interpretando, sempre implicitamente, il simbolo di implicazione delle clausole ("`:-`") come una doppia implicazione ("se e solo se"). È da notare che, adottando questa convenzione, si sta ipotizzando che la conoscenza sul dominio rappresentata dalle clausole sia completa, ovvero che tutto ciò che non è esplicitamente rappresentato vada considerato come falso. È opportuno tenere presente che l'ipotesi del mondo chiuso è relativa alla rappresentazione del problema che si realizza nel programma, cosicché il suo uso va giustificato caso per caso. Per esempio, in una base di dati che rappresenta i voli che collegano le città, espressi da fatti del tipo:

volo('Alitalia', 'Milano', 'Roma').

volo('Alitalia', 'Roma', 'Cagliari').

...

è ragionevole ritenere che tutti i voli siano rappresentati, e quindi quelli non presenti (per esempio **volo('Alitalia', 'Lodi', 'Latina')**) sono da considerare come non disponibili. Invece in una base di dati nella quale si introducono degli individui mediante fatti del tipo:

umano('Giorgio').

umano('Maria').

...

è presumibile che '**Giorgio**', '**Maria**', ... siano coloro che interessa considerare nel contesto in questione, ma ciò non comporta che per esempio '**Filippo**' non sia un **umano** se non vi è un fatto che lo affermi esplicitamente. In tale situazione si considera incompleta la conoscenza espressa nella base di dati, adottando l'ipotesi del mondo aperto (open world assumption), che - contrariamente alla precedente - assume come vera solo l'informazione presente; in questo caso occorre quindi rappresentare esplicitamente tutti i fatti, sia positivi che negativi, e l'implicazione rimane intesa in senso strettamente unidirezionale.

L'ipotesi del mondo aperto è monotona, ossia i risultati dimostrati rimangono validi anche se successivamente si aggiungono altri fatti, mentre l'ipotesi del mondo chiuso è non monotona. Per esempio, aggiungendo alla precedente base di dati su giocatori ed allenatori anche il nuovo fatto **giocatore(enzo)** non vale più la sua negazione, che prima era stata inferita con l'ipotesi del mondo chiuso.

La negazione come fallimento finito.

L'interpretazione della negazione come impossibilità di dimostrazione (si interpreta il fallimento del tentativo di dimostrare **p** come una "dimostrazione" di **not p**) è più restrittiva dell'interpretazione logica classica, perché "non dimostrabile" è diverso da "non vero". Per rendere implementabile la negazione come fallimento, inoltre, è necessaria una ulteriore restrizione. Infatti, per usare l'ipotesi del mondo chiuso come regola d'inferenza, occorre dimostrare che una meta non è conseguenza logica del programma, ma il fallimento della dimostrazione non va inteso solo come mancanza di successo, che può avversi anche a causa di un ciclo infinito, bensì va considerato più restrittivamente come la dimostrazione che tutti i possibili casi terminano con un fallimento. Si adotta allora una regola d'inferenza, più restrittiva rispetto alla sola ipotesi del mondo chiuso, detta regola di negazione come fallimento finito (negation as finite failure): "se tutti i possibili tentativi di dimostrare una meta **M** giungono a termine con un fallimento, allora si considera dimostrata **not M**".

L'operatore di negazione **not**, disponibile nei sistemi Prolog standard, è basato sul principio di negazione come fallimento finito, che però realizza - come vedremo - in modo limitato. Infatti le definizioni precedenti sono indipendenti dalla strategia di prova, mentre nell'implementazione occorre fare i conti con gli aspetti procedurali della particolare strategia standard di Prolog.

La procedura predefinita **not(M)**, che realizza tale principio, termina con successo se il soddisfacimento della meta **M** fallisce, e riesce in caso contrario. Ha dunque comportamento opposto rispetto alla procedura **call(M)**, e può considerarsi definita in termini di essa:

not(M):- call(M), !, fail.

not(_).

In tale definizione, se **M** riesce, la congiunzione "**!, fail**" determina l'abbandono del tentativo di risoddisfare la meta **not** (il taglio impedisce la selezione di una clausola alternativa, mentre **fail** provoca il fallimento della clausola e - conseguentemente - della meta genitrice); viceversa, se **M** fallisce, il ritorno indietro forza l'utilizzo della seconda clausola, grazie alla quale **not(M)** riesce. Pertanto la meta **not M** termina con successo se e solo se **M** non può venire soddisfatta (non è dimostrabile). Si noti che l'esecuzione della meta **not(M)** non può mai modificare il grado di istanziamento par-ziale o totale, o di non istanziamento, della meta **M**.

Si può osservare che tale realizzazione è basata sullo schema del costrutto condizionale:

P -> Q ; R :- P, !, Q.

P -> Q ; R :- R.

dove il ruolo di **Q** è giocato da **fail**, la meta che fallisce sempre, mentre **R** è da intendersi in questo caso come la procedura che riesce sempre (**true**).

L'operatore **not** può essere usato sia nei quesiti che nel corpo delle clausole. Il sistema, incontrando la meta **not p(X)**, passa la chiamata **p(X)** come parametro alla procedura predefinita **not**, la cui definizione interna è equivalente a quella vista sopra.

Consideriamo come esempio il programma logico che consiste dei seguenti tre fatti:

```
azzurro(cielo).
azzurro(mare).
verde(erba).  
[Navigation icons: back, forward, search, etc.]
```

In base alla suddetta implementazione della procedura **not(M)**, è facile verificare che la meta:

?- not azzurro(cielo).

ottiene la risposta **no**, mentre la meta:

?- not azzurro(erba).

ottiene risposta positiva.

In questi casi le risposte sono ottenute correttamente, in quanto i predicati negati sono, come richiesto dall'ipotesi del mondo chiuso, predici chiusi. Se questo non avviene, si possono ottenere risposte inattese; per esempio, sempre in riferimento alla base di dati precedente, la meta:

?- not azzurro(X).

non determina, come ci si potrebbe aspettare, l'istanziamento della variabile **X** ad **erba**, bensì si conclude semplicemente con un fallimento (risposta **no**). Infatti tale meta genera, con la prima clausola della procedura **not(p)**, la risolvente:

:- azzurro(X), !, fail.

e quest'ultima con la prima clausola del programma dà a sua volta luogo alla risolvente:

:- !, fail.

A questo punto l'esecuzione del taglio elimina come scelta alternativa sia la seconda clausola della procedura **not(p)** che le altre clausole del programma, e quindi la meta rimanente:

:- fail.

fallisce.

Con la negazione come fallimento finito si possono dunque verificare predicati negativi, ma non è possibile usarli per generare risposte. Questo però non comporta che una variabile non possa comparire come argomento di un predicato negato: ciò che conta è che essa sia istanziata nel momento in cui viene chiamata la procedura corrispondente al predicato negato.

Consideriamo il seguente semplice programma:

```
p(a).  
q(b).
```

con la meta:

```
?- not p(X), q(X).
```

Il sistema Prolog fornirà la risposta **no**, perché, scegliendo per primo - in base alla strategia standard - il predicato più a sinistra **not p(X)**, X risulta non istanziata.

Invece con la meta, equivalente alla precedente sotto il profilo dichiarativo:

```
?- q(X), not p(X).
```

Prolog fornirà la risposta:

X=b

perché la prima chiamata, **q(X)**, comporta l'unificazione tra **X** e **b**, e quindi la seconda chiamata, **not p(b)**, trova il predicato negato con l'argomento già istanziato. La restrizione che un predicato negato debba essere istanziato nel momento in cui è chiamato può quindi essere superata mediante un riordinamento delle mete. Tuttavia tale riordinamento, che nell'esempio precedente è molto semplice, può risultare alquanto difficile in programmi più ampi e complessi, e quindi dar luogo a possibili errori; comunque impone la necessità di tenere conto del comportamento del programma durante l'esecuzione.

Ciò vale anche se si hanno negazioni di predicati nel corpo delle clausole. Con il programma:

```
p(a).  
q(b).  
r(X) :- not p(X), q(X).
```

la meta:

```
?- r(X).
```

ottiene la risposta **no**; occorre riordinare la terza clausola per ottenere con la stessa meta la risposta **X = b**.

In generale, la regola d'inferenza della negazione come fallimento finito è attuabile con la risoluzione, a condizione che la regola di selezione sia sicura (safe), cioè che selezioni predicati negati solo se i loro argomenti sono istanziati. Poiché i legami vengono stabiliti solo mediante chiamate con successo di predicati non negati, mentre chiamate di predicati negati creano mai legami, ma solo riescono o falliscono, la negazione come fallimento è solo la verifica di una condizione; come tale è restrittiva rispetto alla negazione logica.

Ricordando le definizioni di correttezza e di completezza di una regola d'inferenza date alla fine dell'[Interpretazione dichiarativa](#), si dimostra che - se la regola di selezione è sicura - la risoluzione incrementata con la negazione come fallimento finito è una regola d'inferenza corretta; essa è però incompleta per programmi con negazioni nelle clausole (a causa del fatto che alcune soluzioni non possono essere trovate, perché la computazione con predicati negati npn attua legami).

L'implementazione della negazione in molti sistemi Prolog, basata sull'uso dei predicati "!" e **fail**, è una né corretta né completa realizzazione della negazione come fallimento finito. Infatti la regola di selezione di Prolog non è sicura, perché non effettua il controllo che i predicati negati siano istanziati quando vengono chiamati. Inoltre la strategia di ricerca di Prolog standard non garantisce di percorrere i rami di fallimento finito, anche se questi esistono, perché può perdersi in un ramo infinito, come è stato discusso nell'[Interpretazione procedurale](#) e nella [Strutturazione del controllo](#). Questo tipo di problemi viene superato in realizzazioni del linguaggio nelle quali si utilizzano meccanismi di esecuzione diversi da quello più comune; essi controllano l'istanziamento, al momento della chiamata, delle variabili presenti entro la meta negata, e ne rimandano l'esecuzione fino a che essa non contiene alcuna variabile libera. Meccanismi di questo tipo compaiono ad esempio nello IC-Prolog, nel Prolog II e nel MU-Prolog.

Un predicato negato può riuscire anche quando i suoi argomenti non sono completamente istanziati al momento della chiamata, purché il fallimento del predicato avvenga senza creare legami. Consideriamo il programma seguente:

```
categoria(X, direttore_generale) :-not dipende(X, _).
dipende('Rossi', 'Brambilla').
dipende('Bianchi', 'Esposito').
```

La meta:

```
?- categoria('Verdi', Z).
```

comporta la chiamata della meta **dipende('Verdi', _)**, che fallisce senza legami; di conseguenza **not dipende('Verdi', _)** riesce, fornendo la risposta **Z = direttore_generale**.

Il comportamento del programma varia però se si scambiano i modi di ingresso e uscita dei parametri; per esempio, la meta:

```
?- categoria (X, direttore_generale).
```

dà luogo alla chiamata **not dipende (X, _)** che ha successo se e solo se **dipende(X, _)** fallisce, il che non avviene; si ottiene quindi la risposta **no**. Una soluzione per evitare che una chiamata di procedura come **not dipende(X, _)**, contenente una variabile **X** che compare altrove nella stessa clausola, venga eseguita prima che la variabile **X** sia istanziata, consiste nell'introdurre un'altra condizione che restringa il "tipo" della variabile **X**. Nell'esempio si può procedere sostituendo la prima clausola con la seguente:

```
categoria(X, direttore_generale) :- impiegato(X), not dipende(X, _).
```

ed aggiungendo le seguenti clausole, che permettono di definire l'appartenenza al tipo:

```
impiegato('Rossi').
```

```
impiegato('Brambilla').
```

```
impiegato('Bianchi').
```

```
impiegato('Esposito').
```

```
impiegato( 'Verdi').
```

ottenendo il programma:

```
categoria(X, direttore_generale) :- impiegato(X), not dipende(X, _).  
dipende( 'Rossi', 'Brambilla').  
dipende('Bianchi', 'Esposito').  
impiegato('Rossi').  
impiegato('Brambilla').  
impiegato('Bianchi').  
impiegato('Esposito').  
impiegato('Verdi').
```

La meta:

```
?- categoria(Z, direttore_generale).
```

ottiene allora la risposta:

Z = Brambilla;

Z = Esposito;

Z = Verdi.

La congiunzione "!", fail" per condizioni negative.

Condizioni di verifica in negativo possono essere realizzate utilizzando la congiunzione "**!, fail**". Ad esempio, per accertare che il termine che istanzia una variabile **X** non è né un atomo né un numero, si può definire la procedura:

```
non_atomico(X) :- atomic(X), !, fail.
non_atomico(_).
```

Esaminiamone i possibili comportamenti. Se la variabile **X** è istanziata ad un atomo o ad un numero, la sottometà **atomic(X)** viene soddisfatta e, dopo il taglio, la chiamata a **fail** forza un fallimento che - a causa del taglio - si propaga alla metà **non_atomico(X)**, escludendo la possibilità di utilizzare la seconda clausola. Se invece **X** non è istanziata o è istanziata ad altro, il fallimento di **atomic(X)** porta a considerare la seconda clausola, e la metà **non_atomico(X)** risulta soddisfatta; in tal caso il taglio non ha alcun effetto, in quanto non viene neppure incontrato.

Questa modalità d'uso del taglio, in combinazione con **fail**, ha influenza sul soddisfacimento (o fallimento) della metà che chiama la procedura in cui compare. Usato per la conferma della scelta di una regola o per la terminazione della generazione di soluzioni alternative, infatti, un taglio influisce sulla metà che ne determina l'attivazione solo per le soluzioni successive alla prima. Nella congiunzione "**!, fail**", invece, riferendoci all'esempio precedente per fissare le idee, in assenza del taglio il fallimento forzato dal predicato tali porterebbe ad un tentativo (destinato a positiva conclusione) di risoddisfacimento della metà **non_atomico** mediante la seconda clausola: il risultato sarebbe così di vedere soddisfatta tale metà indipendentemente dalla natura di **X**.

Il programma seguente presenta i due differenti usi del taglio. Una chiamata alla procedura **si_o_no** riesce sempre. Se il **Termine** a primo argomento appartiene alla **Lista** che istanzia il secondo, si ha in uscita un **sì**. La presenza del taglio permette di evitare che eventuali successivi fallimenti portino all'utilizzo della seconda clausola (per la quale si avrebbe la segnalazione opposta!):

```
si_o_no(Termine, Lista) :- entro(Termine, Lista), !, nl, write('sì.'), nl.
si_o_no(_, _) :- nl, write('no.'), nl.
entro(_, Variabile) :- var(Variabile), !, fail.
entro(Elemento, [Elemento|_]).
entro(Elemento, [_|Lista]) :- entro(Elemento, Lista).
```

La procedura **entro** presenta un comportamento leggermente diverso da quello di appartenenza: mentre - se la variabile a secondo argomento è libera - quest'ultima procedura la istanzia secondo necessità, **entro** fallisce definitivamente; non è dunque possibile utilizzarla per istanziare ulteriormente tale lista. Come si è visto nel **paragrafo precedente**, il predicato **not** è definito proprio mediante la congiunzione "**!, fail**". Perciò, per esempio, la definizione precedente di **non_atomico(X)** è equivalente a: **not atomic(X)**. In generale, tutte le clausole intese a dare una definizione in negativo di un predicato mediante "**!, fail**", e ad essere usate con gli argomenti istanziati, sono sostituibili dall'applicazione di **not** a quel predicato. Come ulteriore esempio, definito:

```

pari(N) :- N1 is N mod 2, N1 == 0.
/* not pari(N) ha lo stesso comportamento di dispari(N) */
dispari(N) :- pari(N), !, fail.
dispari(_).

```

Le coppie di predicati predefiniti in cui uno è la negazione dell'altro sono ridondanti:

nonvar(T) è equivalente a: **not var(T)**

X \== Y è equivalente a: **not X == Y.**

Qualsiasi invocazione di " \==" può sempre essere sostituita dall'uso della seguente procedura, che termina con successo se **X** ed **Y** non possono essere unificati:

```

diff(X, X) :- !, fail.
diff(_, _).

```

Tutte le volte che si ha un'equivalenza, è preferibile non usare il taglio direttamente, ma usare il **not**, in quanto costrutto di più alto livello. La disponibilità dei costrutti "**if-then-else**", **not** ed **once** permette di sostituire la maggior parte degli usi del predicato di taglio, limitandone i possibili effetti negativi discussi nel [Controllo del ritorno indietro](#).

Note bibliografiche.

L'ipotesi del mondo chiuso è stata discussa, in relazione alle basi di dati, da Reiter (1978), il quale ha evidenziato la possibilità di incongruenze per basi di dati non organizzate a clausole di Horn. In Gallaire e Minker (1978) si trovano diversi lavori su Prolog e basi di dati. Alcune soluzioni più specifiche al problema della negazione in questo ambito sono trattate in Dahl (1980).

Clark (1978) ha dimostrato per primo che la negazione per fallimento finito è una regola di inferenza corretta, se si impone la condizione che le mete negate siano chiuse; in questo caso si possono derivare fatti negati che potrebbero venire inferiti dalla base di dati completata, cioè una base di dati in cui le definizioni delle relazioni sono espresse mediante la doppia implicazione ("se e solo se") al posto dell'usuale implicazione semplice ("se") delle clausole di Horn; ciò corrisponde a rendere esplicita, anziché lasciare implicita, l'ipotesi del mondo chiuso. La condizione che le mete negate non vengano tentate fino a che non sono chiuse non è molto restrittiva, poiché riguarda soltanto l'ordinamento delle sottomete nella clausola. Un'implementazione corretta della negazione consiste nel "rimandare" la chiamata a **not** sino a che il suo argomento risulta pienamente istanziato. Molti sistemi Prolog però non realizzano tale implementazione, e non è quindi in essi assicurata la correttezza delle soluzioni.

Jaffar, Lassez e Lloyd (1983) hanno poi esteso i risultati di Clark al problema della completezza della negazione come fallimento. Altri, come Gabbay e Sergot (1986) hanno quindi cercato di

definire nuove nozioni di negazione, che includessero la negazione per fallimento come caso particolare. Una trattazione estesa del problema si trova in Naish (1986).

La semantica e l'implementazione del not in IC-Prolog sono descritte da Clark e McCabe (1980). Il Prolog II è esposto in Giannesini, Kanoui, Pasero e van Caneghem (1985). Il MU-Prolog è descritto in Naish (1987).

Sommario.

Lo studente è ora a conoscenza delle conseguenze che la specificazione in un programma di una base di dati incompleta può avere sulle risposte a quesiti che comportano l'invocazione di sottometodi negati. È inoltre consapevole del fatto che la negazione può essere implementata nel sistema Prolog a sua disposizione in modo non corretto. Questa costituisce quindi un'altra possibile fonte di errore da parte del sistema, che si aggiunge alla mancanza della verifica di occorrenza nel procedimento di unificazione, discussa nell'[Interpretazione procedurale](#).

14. Gestione della base di dati

Dove si introducono i predicati predefiniti che operano sul contenuto della base di dati, aggiungendo, cancellando o verificando la presenza di clausole.

E dove si esaminano gli effetti di tale variazione del contenuto logico del programma, differenziando i casi in cui essa ne oscura inutilmente il significato da quelli in cui risulta utile o necessaria. I predicati di sistema introdotti permettono di modificare il programma durante l'esecuzione, mediante l'aggiunta e/o la cancellazione di clausole dalla base di dati, oppure di accettare la presenza o meno di una o più clausole nella base di dati. Utilizzando tali predicati predefiniti si possono poi sviluppare diverse procedure che operano sulla base di dati con varie finalità e modalità.

Aggiunta di clausole.

Le clausole possono essere aggiunte alla base di dati utilizzando i predicati unari **assert(C)**, **asserta(C)** ed **assertz(C)**, il cui argomento dev'essere istanziato ad un termine diverso da una variabile. Le mete espresse da questi predicati riescono sempre e non possono venire risoddisfatte in caso di ritorno indietro. Il loro effetto collaterale è l'aggiunta alla base di dati dell'istanza di **C**, interpretata come clausola (non unitaria se il suo funtore principale ha molteplicità 2, unitaria altrimenti), con nuove variabili locali in sostituzione di tutte le variabili non istanziate.

La posizione della nuova clausola all'interno della procedura alla quale appartiene è non specificata (definita dall'implementazione) per **assert**, è la prima per **asserta**, è l'ultima per **assertz**; se non esiste alcuna procedura dotata del nome di predicato cui è istanziato **C**, essa viene creata. Si determina un errore di esecuzione se l'argomento, pur essendo corretto come termine, non è corretto come clausola, o se la testa della clausola indicata ha come funtore principale il nome di un predicato di sistema; se la coda è costituita da una congiunzione di mete, questa va circondata da parentesi supplementari, in modo da essere trattata come termine singolo.

Esempi:

assert(pred(fun)) riesce;

assert((testa(X, Y) :- coda(X, Y), fun(Y))) riesce;

assert((name(X, Y) :- list(X, Z), carattere(Z, Y))) causa un errore, perché **name** è un predicato di sistema;

assert(1) causa un errore, perché un intero non è una clausola valida.

Dopo l'esecuzione della seguente meta:

?- assert(proc(b)), assertz(proc(c)), asserta(proc(a)), assertz(proc(d)).

con il quesito:

?- proc(X), write(X), tab(2), fail.

si ottiene l'uscita: **a b c d**

Come esempio di applicazione di **assertz**, la seguente procedura legge da un file un programma clausola per clausola, sino alla clausola conclusiva **end.**, e lo memorizza nella base di dati:

```
lettura(File):-seeing(I), see(File), repeat, read(Clausola), assertz(Clausola), Clausola == end,  
!,seen, see(I).
```

Nelle varie implementazioni di Prolog si trovano diverse varianti di tali predicati.

Cancellazione di clausole.

L'esecuzione della meta:

retract(C)

determina la ricerca nella base di dati di una clausola che unifichi con la variabile istanziata **C** (che deve soddisfare gli stessi requisiti richiesti per **assert**); se la meta riesce, il suo effetto collaterale consiste nella cancellazione della clausola che ha determinato l'unificazione.

La meta **retract(C)** può venire risoddisfatta in seguito a ritorno indietro; è una delle poche procedure predefinite di natura non deterministica. La variabile **C** può essere istanziata anche solo parzialmente; questo dà la possibilità di utilizzare la procedura **retract** in maniera non deterministica per cancellare progressivamente dalla base di dati, mediante ritorno indietro, tutte le clausole che danno luogo ad una unificazione con l'argomento **C**. I seguenti sono esempi di applicazione del predicato predefinito **retract**, di cui esistono varianti nelle diverse versioni di Prolog.

La congiunzione di mete:

```
retract((risposta(atomo_1, atomo_2, _) :- trova(_, atomo_1), cerca(atorno_2, a))), fail.
```

permette di cancellare tutte le clausole della procedura **risposta**, i cui primi due argomenti della testa unificano con **atorno_1** ed **atomo_2** (mentre il terzo può essere qualsiasi) e la cui coda è formata dalle due sottomete indicate.

Per cancellare tutte le clausole specificate in una lista:

cancellazione_clausole([]).

```
cancellazione_clausole([Cl|Cl_rimanenti]):-retract(Cl), cancellazione_clausole(Cl_rimanenti).
```

La procedura che segue cancella invece tutte le clausole aventi la testa specificata dall'argomento d'ingresso:

```
cancellazione_di_clausole(Testa):-retract(Testa), fail.
```

```
cancellazione_di_clausole(Testa):-retract((Testa:-_)), fail.
```

```
cancellazione_di_clausole(_).
```

E un esempio di procedura iterativa con ritorno indietro forzato dal predicato di sistema **fail**, che utilizza il non-determinismo di **retract**; l'ultima clausola ne assicura il successo sia in caso di assenza di clausole aventi la testa indicata, sia quando esse sono già state tutte cancellate.

L'uso combinato dei predicati **=..**, **retract** ed **assertz** permette di modificare le clausole della base di dati. Per esempio, la seguente procedura è utilizzabile con entrambi gli argomenti istanziati ad atomi, il primo dei quali è il funtore principale di una struttura di molteplicità qualsiasi presente nel programma, mentre il secondo è il funtore da sostituire al precedente:

```
copiatura(Funt_vecchio,Funt_nuovo):- Termine_vecchio =..  
[Funt_vecchio|Arg],Termine_nuovo =.. [Funt_nuovo|Arg],  
retract(Termine_vecchio),assertz(Termine_nuovo),fail.
```

```
copiatura(_,_).
```

Ricerca di clausole.

Il predicato predefinito:

```
clause(T, C)
```

afferma l'esistenza nella base di dati di una clausola di testa **T** e coda **C**. La variabile **T** dev'essere istanziata ad un termine non variabile, altrimenti la meta fallisce. Se **C** è una congiunzione di mete, è necessario racchiuderla all'interno di parentesi supplementari per evitare che le mete vengano considerate come parametri aggiuntivi di **clause**. **C** può essere istanziata o meno; nel secondo caso, se la meta riesce, **C** viene istanziata alla coda della clausola se è una regola, od a **true** se è un fatto. La meta **clause** non presenta effetti collaterali e può venire risoddisfatta (**clause**, **retract**, **call**, **repeat** sono i soli predicati predefiniti non deterministici).

Se nella base di dati sono presenti solo le clausole:

```
pred(arg,arg).  
pred(arg1,arg2):-pr(arg1,arg2,arg3).
```

allora si ha che:

```
clause(pred(arg, arg)). fallisce;
```

```
clause(pred(arg, arg), true). riesce;
```

```
clause(fun(arg)). fallisce;
```

```
clause(pred(arg1, arg2), pr(arg1,arg2,arg3)). riesce;
```

```
clause(pred(_,_), X). istanzia X a true e poi, se vi è ritorno indietro, a pr(arg1, arg2, arg3).
```

PROCEDURA: ricerca_e_scrittura_clausole(T).
 D: T è la testa di una o più clausole presenti nella base di dati.
 P: ricerca_e_scrittura_clausole(<): scrive sul flusso corrente di uscita tutte le clausole di testa T.
 T: consente l'omissione della verifica Corpo $\not\models$ true nella seconda alternativa.
 BD: non modificata.
 C: può essere risoddisfatta mediante ritorno indietro. Eredita questa proprietà da clause/2.
 */
 ricerca_e_scrittura_clausole(Testa):-clause(Testa,Corpo), (Corpo = true,!; writeq(Testa); writeq((Testa:-Corpo))), write(.), nl, fail.

Simulazione dell'assegnamento.

I predicati di aggiunta di clausole consentono di utilizzare la base di dati quale mezzo di scambio di informazioni tra procedure alternativo al passaggio di parametri. Questo vale sia all'interno di una stessa procedura che tra procedure diverse.

Per esempio, un'iterazione **N2 - N1 + 1** volte (con **N2** non minore di **N1**) di un'operazione può essere realizzata secondo il seguente schema:

```
iterazione_con_assert(N1,N2):-assert(contatore(N1)),repeat,retract(contatore(N)),write(N),M is N+1,assert(contatore(M)), M>N2
```

(dove naturalmente al posto delle **write** possono esserci operazioni qualsiasi). Si noti che un solo fatto **contatore(N)** è presente ad ogni istante, cosicché si è usato **assert** senza riguardo alla posizione in cui inserirlo; si può inoltre osservare che, poiché **retract(contatore(N))** nel cancellare **contatore(N)** istanzia **N**, può essere posto subito dopo **repeat** al posto di **contatore(N)**, semplificando lo schema.

L'uso di **assert** e **retract** a tale scopo è tuttavia superfluo; l'iterazione suddetta può essere realizzata più semplicemente come ricorsione in coda:

```
iterazione(N1,N2):-N1>N2,write('FINE').  
iterazione(N1,N2):-write(N1),N is N1+1,iterazione(N,N2).
```

Come esempio di scambio di informazioni tra procedure diverse, supponiamo che una procedura parametro abbia accertato il valore di un parametro (0 o 1) in base al quale un'altra procedura chiamata successivamente debba compiere una scelta fra due alternative:

parametro(Argomenti):-computazione_parametro(Par), assert(valore(Par)),proseguzione(Argomenti).

proseguzione(Argomenti):-procedure_varie(Argomenti),valore(Par), gestione_parametro(Argomenti,Par).

gestione_parametro(Argomenti,0):-primo_cammino(Argomenti).

gestione_parametro(Argomenti,1):-secondo_cammino(Argomenti).

In tal modo si rendono disponibili alle procedure dei dati globali, mediante una sorta di memoria condivisa, ai quali possono per altro virtualmente accedere tutte le procedure del programma.

Naturalmente lo stesso effetto può essere ottenuto con l'usuale passaggio di parametri, aggiungendo un argomento alle sole procedure interessate:

parametro_1(Argomenti):-computazione_parametro(Par), proseguzione(Argomenti,Par).

**prosecuzione(Argomenti,Par):-procedure_varie(Argomenti),
gestione_parametro(Argomenti,Par).**

L'utilizzo della base di dati nel modo sopra esemplificato, con il quale viene in sostanza simulato l'assegnamento dei linguaggi di programmazione procedurali, è in contrasto con le caratteristiche precipue della programmazione logica, basata sull'assenza di variabili globali. Mentre l'interpretazione di una chiamata di **assert** è in sè chiara, il significato logico del programma può diventare oscuro; infatti le clausole aggiunte possono essere invocate da chiamate successive e quindi influenzare il corso dell'esecuzione, e non è più determinabile cosa è conseguenza logica di un programma che si trasforma.

Particolari problemi possono sor4ere quando l'uso di **assert** interagisce con quello di **not**: una meta negata può riuscire in un dato momento dell'esecuzione in quanto non dimostrabile in base alle clausole esistenti in quel momento, ma successivamente vengono aggiunte altre clausole che, se usate, farebbero fallire quella stessa meta.

Anche ragioni di efficienza sconsigliano di utilizzare **assert** e **retract** quando non necessario, in quanto l'aggiunta di una clausola nella base di dati richiede che l'intera struttura venga dapprima copiata, poi sottoposta ad analisi sintattica ed infine indicizzata e memorizzata; inoltre, il recupero dello spazio occupato da una clausola cancellata ha anch'esso un costo.

Queste considerazioni suggeriscono di usare la modifica della base di dati del programma solo nei casi in cui la persistenza di un'informazione tra differenti chiamate del programma o di sue sottoparti è motivata da particolari scopi, riconducibili sostanzialmente ai seguenti:

- Per generare lemmi, cioè aggiungere clausole che sono conseguenza logica di quelle già esistenti, ma possono essere sfruttate per rendere più efficiente l'esecuzione.
- Per memorizzare informazioni fornite dall'utente mediante programmi interattivi, per esempio per tenere traccia di opzioni, comandi od espressioni dell'utente; oppure in basi di dati od in basi di conoscenza il cui contenuto cresce incrementalmente o comunque si modifica durante il suo utilizzo nel tempo; od ancora in programmi il cui compito è di operare su altri programmi, come strumenti usati per sviluppare, eseguire o modificare altri programmi.
- Per generare nuovi simboli o numeri pseudocasuali Questa operazione può essere confinata in una procedura di libreria, considerata come estensione dei predicati predefiniti aventi effetti collaterali, rinunciando ad una interpretazione dichiarativa.
- Quando si vogliono raccogliere in una struttura di dati tutte le soluzioni ad un quesito, ossia tutte le istanze di un oggetto che soddisfano una certa meta o congiunzione di mete.

Generazione di lemmi.

L'aggiunta alla base di dati di una clausola che è conseguenza logica delle clausole già esistenti non altera il contenuto logico del programma, ed è quindi innocua dal punto di vista dichiarativo, mentre risulta utile sotto il profilo dell'efficienza se può essere usata nel seguito dell'elaborazione che in sua assenza dovrebbe ricomputarla. La clausola aggiunta agisce, nella dimostrazione di un quesito, come un lemma, cioè come un'asserzione, già dimostrata in precedenza, che agevola lo sviluppo della dimostrazione in corso. Trasformando una regola del tipo:

regola(Parametri) :- congiunzione_di_mete(Parametri).

nella nuova regola:

regola(Parametri) :- congiunzione_di_mete(Parametri), assert(regola(Parametri)).

ogni volta che la metà:

?- regola(Parametri).

riesce, istanziando il suo argomento, un fatto **regola(Parametri)** con la stessa istanza di **Parametri** viene aggiunto alla base di dati, e può quindi essere usato in un'analogia metà successiva. Ad esempio, nello schema di procedura per il calcolo di derivate del § 5.3, le regole generali di derivazione, che seguono i fatti esprimenti le derivate di base, possono essere scritte nella forma:

derivata(X, Y, Z) :- calcolo(X, Y, Z), asserta(derivata(X, Y, Z)).

aggiungendo, ad ogni invocazione, una nuova derivata già calcolata a quelle di base preesistenti.

Questo tipo di utilizzo di **assert** può essere incapsulato entro una primitiva di generazione di lemmi, della forma:

lemma(P) :- call(P), assert(P).

dove **assert** può essere sostituito da **asserta** o **assertz**, secondo necessità; tale primitiva di più alto livello viene così a costituire una combinazione ragionevole di logica e controllo. Un esempio di applicazione della generazione di lemmi è dato dalla seguente procedura per il calcolo del coefficiente binomiale (il numero di combinazioni di **n** oggetti presi **k** alla volta senza ordine), definito da:

$$\binom{n}{k} = \frac{n!}{k! * (n - k)!}$$

Si può calcolare il fattoriale del minore tra **n**, **k** e (**n** - **k**) ed usarlo come lemma per calcolare il fattoriale del minore tra i due restanti, quindi usare quest'ultimo come lemma per calcolare il fattoriale rimanente; infine i lemmi usati vengono cancellati, per riprendere dall'inizio un eventuale calcolo successivo (si assume la definizione di lemma con **asserta**):

```
lemma(P):-call(P),asserta(P).
coefficiente_binomiale(N,K,Coeff):-N =:= K,! ,Coeff=1.
coefficiente_binomiale(N,K,Coeff):-N-K > K,! , lemma(fattoriale(K,FK)), NK is N-K, lemma(fattoriale(NK,FNK)), fattoriale(N,FN),Coeff is FN//(FK*FNK), retract(fattoriale(K,FK)),retract(fattoriale(NK,FNK)).
coefficiente_binomiale(N,K,Coeff):-N-K =:= K,! , lemma(fattoriale(K,FK)), fattoriale(N,FN),Coeff is FN//(FK*FNK), retract(fattoriale(K,FK)),retract(fattoriale(NK,FNK)).
coefficiente_binomiale(N,K,Coeff):-K < N, N-K < K, NK is N-K,lemma(fattoriale(NK,FNK)), lemma(fattoriale(K,FK)), fattoriale(N,FN),Coeff is FN//(FK*FNK), retract(fattoriale(K,FK)),retract(fattoriale(NK,FNK)).
fattoriale(0,1).
fattoriale(N,F):-N>0,N1 is N-1,fattoriale(N1,F1),F is N*F1.
```

Un altro caso di miglioramento di efficienza dell'esecuzione a parità di contenuto logico del programma è quello in cui vi siano due o più regole con conclusioni diverse e le stesse premesse, cioè del tipo:

a :- c.

b :- c.

Nelle elaborazioni in cui è necessario dimostrare sia **a** che **b**, tali regole portano a dimostrare due volte **c** che può naturalmente essere una meta complessa. Si può evitare questa ridondanza estendendo le due regole suddette nel modo seguente:

a :- c, assert(b).

b :- c, assert(a).

cosicché la dimostrazione con successo di una conclusione aggiunge l'altra come lemma.

Memorizzazione di informazioni durevoli.

In programmi interattivi, o che operano su altri programmi, può essere necessario od utile registrare informazioni in grado di permanere tra una chiamata e l'altra di un programma o di sue sottoparti. Le procedure seguenti sono esempi di alcune possibilità in questa direzione; la loro definizione è essenzialmente procedurale, ed i tagli che vi compaiono sono generalmente rossi.

Per effettuare operazioni (ad esempio di inizializzazione) solo alla prima invocazione di una procedura si può utilizzare **assert** nella maniera seguente:

```
programma_generale(Argomenti):- gia_invocata_una_volta, !,elaborazione(Argomenti),write('OK'),nl.  
programma_generale(Argomenti):-assert(gia_invocata_una_volta), inizializzazione, write(iniz), nl, elaborazione(Argomenti).  
inizializzazione.
```

In occasione della prima chiamata verrà utilizzata la seconda clausola (la prima infatti fallisce), mentre successivamente sarà usata solo la prima, evitando di ripetere le inizializzazioni (è chiaro che la clausola **già_invocata_una_volta** dovrà essere cancellata se si vogliono rieffettuare le operazioni di inizializzazione). La seguente procedura è preposta a segnalare rispettivamente l'attivazione e la disattivazione di un'opzione presente in un programma più ampio:

```
segnalazione:-not segnale,! ,assert(segnale), write('Opzione attivata.'), nl.  
segnalazione.  
ripristino:-retract(segnale),!,write('Opzione disattivata.'),nl.  
ripristino.
```

Le chiamate a **segnalazione** non hanno effetto se l'ultimo messaggio relativo all'opzione ha segnalato che questa è attivata; in caso contrario viene attivata la segnalazione stessa. Inversamente, il messaggio di disattivazione può occorrere solo se la clausola unitaria **segnale** è presente nella base di dati (in tal caso la sottometa **retract(segnale)** riuscirà, mentre in caso contrario fallirà), e non potrà avere ulteriormente luogo senza essere preceduto da un messaggio di attivazione.

La seguente è una procedura per la stampa condizionale di messaggi, in funzione del valore di un opportuno parametro:

```
stampa_condizionale(Messaggio,N):-contatore_stampa(M),M < N,!.  
stampa_condizionale(Messaggio,_):-write(Messaggio),nl.  
modifica_contatore_stampa(Cont_vecchio) :- retract(contatore_stampa(Cont_vecchio)), asserta(contatore_stampa(Cont_nuovo)).  
contatore_stampa(2).
```

La variabile **Messaggio** può essere istanziata ad un qualsiasi termine, mentre **N** è un intero; quando la procedura viene invocata, verrà stampato **Messaggio** se e solo se **N** è inferiore od uguale al valore corrente del contatore di stampa. A tale valore è possibile accedere mediante la clausola unitaria **contatore_stampa**, che può essere reistanziata invocando **modifica_contatore_stampa** con il nuovo valore come argomento. In altri casi può essere utile registrare quante volte viene compiuta

una certa operazione, o quante volte si verifica una certa condizione. Una procedura per tale scopo è:

```
conteggio_operazioni :- eventi_occorsi(N), !, N1 is N+1, retract(eventi_occorsi(N)), assert(eventi_occorsi(N1)), write(N1), nl.  
conteggio_operazioni :- assert(eventi_occorsi(1)), write(1), nl.
```

Alla prima chiamata della procedura, la meta **eventi_occorsi(N)** fallisce, in quanto nella base di dati non è ancora presente un fatto con tale funtore, e viene utilizzata la seconda clausola; nelle chiamate successive l'unificazione riesce e l'argomento di **eventi_occorsi** viene aggiornato. Si noti che la formulazione alternativa:

```
conteggio_operazioni_1 :- write(no), nl , not eventi_occorsi(_), !, assert(eventi_occorsi(1)), write(1), nl.  
conteggio_operazioni_1 :- eventi_occorsi(N), N1 is N+1, retract(eventi_occorsi(N)), assert(eventi_occorsi(N1)), write(N1), nl.
```

è meno efficiente, in quanto comporta l'effettuazione del ritorno indietro alla seconda clausola per tutte le chiamate della procedura tranne la prima, mentre la formulazione precedente richiede l'utilizzo della seconda clausola soltanto all'atto della prima chiamata. La procedura seguente ha la particolare proprietà di riuscire e fallire alternativamente, invocazione dopo invocazione:

```
interruttore:-retract(interruttore_aperto),!,assert(interruttore_chiuso), write(chiuso_n), nl.  
interruttore:-retract(interruttore_chiuso),assert(interruttore_aperto), write(aperto),nl,!,-fail.  
interruttore:-assert(interruttore_chiuso),write(chiuso_1),nl.
```

Alla prima invocazione della procedura, il fallimento della prima sottometa delle prime due clausole attiva il ritorno indietro e porta all'utilizzo della terza clausola, che memorizza il fatto **interruttore_chiuso**. Alla seconda chiamata, ed a tutte le eventuali successive chiamate di ordine pari, l'unificazione con la prima clausola fallisce (**interruttore_aperto** non è presente nella base di dati), ma quella con la seconda riesce; la combinazione "**!, fail**" viene qui utilizzata per forzare il fallimento della meta principale. Alla terza invocazione, ed a tutte le eventuali successive di ordine dispari, riesce l'unificazione con la prima clausola, e la chiamata termina con successo. La terza clausola viene dunque utilizzata soltanto nella prima invocazione di procedura, mentre in tutte le successive viene usata la prima o la seconda, alternativamente.

Generazione di costanti.

Due casi tipici che richiedono la memorizzazione di informazioni da una chiamata all'altra di una procedura sono la generazione di nuove costanti e la generazione di numeri pseudocasuali, illustrate dalle due procedure seguenti.

```
/*  
PROCEDURA: generazione_costanti(Radice, N_iniz, Costante).  
D: Costante è una costante (atom o numero) non preesistente di prefisso Radice; Radice è una costante, N_Iniz è un numero naturale.  
P: generazione_costanti(<, <, >): ad ogni chiamata produce una nuova costante, concatenando a Radice un numero sempre crescente  
a partire dal successore di N_iniz.  
BD: un'istanza della clausola ultimo_numero(Radice, N) è sempre presente.  
U: concatenazione_costanti  
*/  
generazione_costanti(Radice, Num_iniz,Costante):- atomic(Radice),var(Costante), generazione_numero(Radice, Num_iniz, N), N1 is  
N+1,assert(ultimo_numero(Radice, N1)), concatenazione_costanti(Radice, N1, Costante).  
generazione_numero(Radice,_,N):-retract(ultimo_numero(Radice,N)).!  
generazione_numero(_,N,N).  
concatenazione_costanti(C1,C2,C):-name(C1,L1),name(C2,L2), concatenazione(L1,L2,L), name(C,L).  
concatenazione([],L,L).  
concatenazione([T|L1],L2,[T|L3]):-concatenazione(L1,L2,L3).  
/*  
PROCEDURA: gener_numeri_casuali(L, N).  
D: N è un numero compreso tra 1 ed L.  
P: gener_numeri_casuali(<, >): genera in maniera pseudocasuale un numero compreso tra 1 ed il limite indicato con il primo  
argomento.  
BD: un'istanza di numero_base è persistente.
```

```
*/  
numero_base(13).  
gener_numeri_casuali(Limite,Num_casuale) :- retract(numero_base(Num)),Num_casuale is (Num mod Limite)+1, Nuovo_num is (125*Num+1) mod 4096,assert(numero_base(Nuovo_num)).
```

Con il quesito:

?- repeat, gener_numeri_casuali(10,N), write(N), nl, fail.

si ottiene una serie infinita di numeri interi compresi fra **1** e **10**. Come esempio di utilizzo di **gener_numeri_casuali**, la seguente procedura può essere usata per fornire un elemento scelto a caso da una lista assegnata:

```
elemento_casuale(Lista,Elemento) :- lunghezza(Lista,Num),  
gener_numeri_casuali(Num,Num_casuale),ennesimo(Num_casuale,Lista,Elemento).  
ennesimo(1,[E|_],E).  
ennesimo(N,[C|C'],E):-N1 is N-1, ennesimo(N1,C,E).  
numero_base(13).  
gener_numeri_casuali(Limite,Num_casuale) :- retract(numero_base(Num)),Num_casuale is (Num mod Limite)+1, Nuovo_num is (125*Num+1) mod 4096,assert(numero_base(Nuovo_num)).
```

Raccolta di tutte le soluzioni.

Nella [Strutturazione dei dati](#) si è visto come strutture di dati rappresentate con termini possono essere elaborate da procedure ricorsive che mantengono gli istanziamenti da una chiamata all'altra, mentre se sono rappresentate con clausole possono essere scandite da ritorno indietro implicito o esplicito, che però non conserva gli istanziamenti dei rami alternativi dell'albero di ricerca.

Alcuni problemi non sono esprimibili senza la possibilità di memorizzare tutte le soluzioni dell'esecuzione non deterministica di una meta; anche il conoscere solo il numero degli oggetti che soddisfano una certa condizione richiede di raccoglierli in una struttura di dati in cui potere contarli. Esaminiamo quindi quali caratteristiche presentano i due metodi suddetti rispetto allo scopo di raccogliere tutte le soluzioni di una meta, considerando, per esemplificare, la seguente procedura a clausole unitarie:

targa('MI', '42454A').

targa('MI', '54121L').

targa('MI', '42427A').

targa('MI', '54444M').

targa('TO', '76392P').

targa('FI', '23381L').

targa('BO', '67566W').

targa('MI', '42411D').

targa('MI', '54327S').

targa('MI', '42443K').

targa('MI', '54321L').

targa('FO', '76561L').

targa('FI', '23451M').

Volendo conoscere tutti i numeri delle targhe di una data sigla, si può definire una procedura ricorsiva che accumula le soluzioni in un argomento aggiuntivo:

```
targhe(Sigla,Lista_targhe) :- targhe_1(Sigla,[],Lista_targhe).
targhe_1(Sigla,L_accum,Lista_targhe) :- targa(Sigla,Num_targa), not appartenenza(Num_targa,L_accum), !,
targhe_1(Sigla,[Num_targa|L_accum],Lista_targhe).
targhe_1(_,Lista_targhe,Lista_targhe).

targa('MI', '42454A').
targa('MI', '54121L').
targa('MI', '42427A').
targa('MI', '54444M').
targa('TO', '76392P').
targa('FI', '23381L').
targa('BO', '67566W').
targa('MI', '42411D').
targa('MI', '54327S').
targa('MI', '42443K').
targa('MI', '54321L').
targa('FO', '76561L').
targa('FI', '23451M').
appartenenza(X,[X|_]).
appartenenza(X,[_|L]) :- appartenenza(X,L).
```

A quesiti del tipo:

?- targhe('FI', Targhe_FI).

?- targhe('PA', Targhe_PA).

la procedura risponde:

Targhe_FI = [23451M, 23381L]

Targhe_PA = []

Questo approccio presenta però due inconvenienti. Il primo è che ogni volta che viene ricercata un'ulteriore soluzione, l'albero di ricerca viene ripercorso da capo: la procedura ritrova una soluzione già ottenuta, scopre di averla già considerata perché è presente nella lista costruita fino a quel punto, quindi fallisce e ritorna indietro, per ognuna delle soluzioni precedenti, prima di arrivare alla successiva; questo comportamento rende la procedura inefficiente in un modo che può anche diventare proibitivo. Il secondo problema è che non è possibile ottenere tutte le coppie **Sigla/Lista_targhe**, in quanto la metà:

?- targhe(Sigla, Lista_targhe).

è deterministica a causa del taglio, e quindi dà la prima coppia:

Sigla = MI

Lista_targhe = [54321L, 42443K, 54327S, 42411D, 54444M, 42427A, 54121L, 42454A]

e termina senza fornire le successive.

La rimozione del taglio risulterebbe viceversa in un comportamento indesiderato della procedura se usata con il primo argomento istanziato ed il secondo libero, in quanto nel caso di successivi ritorni indietro darebbe come soluzioni tutte le permutazioni e tutte le sottoliste della lista completa.

Per raccogliere tutte le soluzioni mediante il ritorno indietro è necessario usare i predicati di sistema di aggiunta e cancellazione delle clausole, per memorizzare le soluzioni via via trovate e poi raccoglierle. Nell'esempio, volendo conoscere tutti i numeri delle targhe di Firenze:

```
targa('MI', '42454A').  
targa('MI', '54121L').  
targa('MI', '42427A').  
targa('MI', '54444M').  
targa('TO', '76392P').  
targa('FI', '23381L').  
targa('BO', '67566W').  
targa('MI', '42411D').  
targa('MI', '54327S').  
targa('MI', '42443K').  
targa('MI', '54321L').  
targa('FO', '76561L').  
targa('FI', '23451M').  
/* si possono memorizzare nella base di dati quelli che hanno targa 'FI' con la procedura: */  
ricerca_targhe:-targa(Città, Numero), selezione_targhe(Città, Numero), fail.  
ricerca_targhe.  
selezione_targhe('FI', Numero):-!, assertz(targa_Firenze('FI', Numero)).  
selezione_targhe(_).  
/* Dopo aver eseguito la procedura precedente, la lista delle targhe fiorentine può venire raccolta con la procedura: */  
raccolta_targhe(Lista_provv, Lista_finale) :- retract(targa_Firenze('FI', Num)), !, raccolta_targhe([(FI, Num)|Lista_provv], Lista_finale).  
raccolta_targhe(Lista_finale, Lista_finale).
```

L'invocazione:

```
?- ricerca_targhe, raccolta_targhe([], Lista), write(Lista), write(., nl).
```

fornisce la soluzione:

```
[FI, 23451M, FI, 23381L]
```

Poiché il problema è generale, piuttosto che definire procedure apposite caso per caso, si può sistematizzare un procedimento applicabile a qualunque meta; a questo scopo è devoluto al modulo:

```
/*  
MODULO: tutte_le_soluzioni.  
ESPORTA: raccolta_soluzioni(Oggetto, Meta, Lista).  
IMPORTA: -  
USA: asserta/i, retract/1, call/I, fail/0,=/2, !/O, \==/2.  
Esegue non deterministicamente la meta specificata, inserisce in Lista tutte le istanze di Oggetto dimostrabili con Meta in modo finito, nell'ordine in cui sono trovate nella ricerca e con eventuali duplicati. Quindi non termina se l'esecuzione di Meta non termina; istanzia Lista alla lista vuota se Meta non ha soluzioni. Il termine Meta rappresenta una meta od una congiunzione di mete, specificate come se si trovasse nel corpo di una clausola, ma in parentesi se più di una. In particolare, raccolta_soluzioni stessa può comparire (ricorsivamente) in Meta; in tal caso, Lista diventa una lista di liste. Poiché raccolta_soluzioni ha senso solo se Oggetto e Meta hanno qualche variabile in comune, il termine Oggetto deve essere o una variabile che compare in Meta o una struttura che contiene variabili presenti in Meta; tali variabili non devono figurare in nessun'altra parte della clausola contenente raccolta_soluzioni. Eventuali variabili libere in Meta, che non siano in condivisione con Oggetto, vengono trattate come variabili anonime, nel senso che il tentativo di soddisfare Meta non terrà conto degli istanziamimenti che, in corrispondenza di questo, si verificheranno per tali variabili. Ogni eventuale soluzione di Meta che contenga variabili è rappresentata, nel corrispondente elemento di Lista, da un termine con variabili, che costituisce un rappresentante singolo ed arbitrario degli infiniti modi diversi di istanziare tali variabili.  
*/  
/*  
PROCEDURA: raccolta_soluzioni(Oggetto, Meta, Lista).  
D: Lista ha come elementi tutte le istanze del termine Oggetto che soddisfano Meta.  
P1: raccolta_soluzioni(<, <, >) (con il derivato (<, <, <)).  
P2: raccolta_soluzioni(>, <, >).  
T: impedisce di risoddisfare raccolta_istanze in caso di ritorno indietro della procedura che usa raccolta_soluzioni.
```

BD: tutte le clausole aggiunte saranno poi cancellate.
C: cerca, mediante ritorno indietro forzato, tutti gli istanziamenti di Oggetto, e memorizza ognuno di essi nella base di dati in cima ad una pila di fatti istanza(Oggetto), a partire dalla marca iniziale istanza(limite_inferiore), fino a quando il fallimento di Meta attiva la seconda clausola che, chiamando deterministicamente raccolta_istanze([], Lista), istanzia Lista alla lista di tutti i termini Oggetto della pila.
*/
raccolta_soluzioni(Oggetto,Meta,_):-asserta(istanza(limite_inferiore)), call(Meta), asserta(istanza(Oggetto)), fail.
raccolta_soluzioni(_,_,[List]):-raccolta_istanze([],List),!.

/*
PROCEDURA: raccolta_istanze(L_parziale,L_finale).
P: raccolta_istanze(<, >).
T: rosso.
C: scandisce la pila di fatti istanza(Oggetto), inserendo progressivamente in L_parziale (inizialmente vuota) le istanze di Oggetto ivi trovate, fino a quando, esaurita la pila, istanzia con la seconda clausola L_finale alla lista così costruita.
*/
raccolta_istanze(Lista_parziale,Lista_finale) :- cancellazione_istanza(Oggetto),!, raccolta_istanze([Oggetto|Lista_parziale],Lista_finale).
raccolta_istanze(Lista,Lista).

/*
PROCEDURA: cancellazione_istanza(Oggetto).
P: cancellazione_istanza(>).
T: impedisce il ritorno indietro a retract(istanza(Oggetto)) che porterebbe alla ricerca di altri fatti istanza(Oggetto) posti inferiormente ad istanza(limite_inferiore): tali fatti o non esistono (la loro ricerca sarebbe dunque inutile) o, se esistono, non sono stati creati con l'invocazione di raccolta_soluzioni (occorre perciò ignorarli).
BD: riporta la base di dati nella situazione iniziale.
*/
cancellazione_istanza(Oggetto) :- retract(istanza(Oggetto)),!, Oggetto \== limite_inferiore.
lunghezza([],0).
lunghezza([_|C],N):-lunghezza(C,N1), N is N1+1.
/* Dati i seguenti fatti: */
pratica('Paolo',tennis).
pratica('Anna',tennis).
pratica('Gianni',calcio).
pratica('Linda',corsa).
pratica('Marco',calcio).
pratica('Grazia',nuoto).
pratica('Lia',tennis).
pratica('Grazia',salto).
/* Naturalmente raccolta_soluzioni può essere usata per raccogliere le soluzioni di un quesito relativo ad una qualsiasi relazione definita intensionalmente. Per esempio, definita la relazione addendi(L1, N, L2) (L2 è una lista di numeri naturali appartenenti alla lista L1 tali che la loro somma è N): */
addendi(_,0,[]).
addendi([T|C1],N,[T|C2]):-T=< N, N1 is N-T, addendi(C1,N1,C2).
addendi([_|C],N,I):-addendi(C,N,I).
addendi([],_,_):-fail.
/* Si può anche osservare che l'applicazione della procedura raccolta_soluzioni può riguardare qualsiasi combinazione di argomenti di una relazione. Date, per esempio, le seguenti clausole: */
pred(a,c,l,m).
pred(b,e,f,v).
pred(b,c,d,i).
pred(g,k,o,r).

sport_praticato_da(Sport, N) :- bagof(Persona,pratica(Persona,Sport), Lis), lunghezza(Lis, N).

esempi di quesiti e relative risposte sono:

?- raccolta_soluzioni(Persona,pratica(Persona,tennis),Tennisti).

Tennisti = [Paolo, Anna, Lia]

?- raccolta_soluzioni(Persona,pratica(Persona,sci),Sciatori).

Sciatori = []

**?- raccolta_soluzioni(Persona,pratica(Persona,calcio),Calciatori),
lunghezza(Calciatori,N_calc).**

N_calc =2

?- **raccolta_soluzioni(Persona,pratica(Persona,Sport),Praticanti).**

Praticanti = [Paolo, Anna, Gianni, Linda, Marco, Grazia, Lia, Grazia]

?- **raccolta_soluzioni(Sport,pratica(Persona,Sport),Sport_praticati).**

Sport_praticati = [tennis, tennis, calcio, corsa, calcio, nuoto, tennis, salto]

?- **raccolta_soluzioni(Sport,pratica('Grazia',Sport),Sport_Grazia).**

Sport_Grazia = [nuoto, salto]

il quesito:

?- **raccolta_soluzioni(coppia (Primo, Quarto), pred(Primo, _, _, Quarto), Coppie).**

porta ad ottenere:

Copie = [coppia(a, m), coppia(b, v), coppia(b, i), coppia(g, r)]

il quesito:

?- **raccolta_soluzioni(Z,addendi([1,2,3,5,7],10, Z),L).**

dà la soluzione:

L = [[1,2,7],[2,3,5],[3,7]]

Si noti che per evitare di ottenere elementi duplicati è sufficiente sostituire la sottometa:

asserta(istanza(Oggetto))

entro la prima clausola per raccolta_soluzioni, con la sottometa:

controllo_presenza(Oggetto)

ed aggiungere la procedura **controllo_presenza(Oggetto)**:

controllo_presenza(Oggetto):-istanza(Oggetto),!.

controllo_presenza(Oggetto):-asserta(istanza(Oggetto)).

L'uso ricorsivo di **raccolta_soluzioni** è reso possibile dalla presenza del delimitatore **limite_inferiore**, nel senso che i fatti aggiunti nella base di dati per effetto della chiamata ricorsiva risultano separati (ed in posizione precedente) da quelli aggiunti con la chiamata non ricorsiva. Un esempio di tale uso ricorsivo è la seguente procedura che, assegnata a L1 una lista qualsiasi, istanzia L2 alla lista di tutti e soli gli elementi della prima lista che sono strutture ad un solo argomento diverso da un atomo:

ricerca_elementi(L1,L2) :- raccolta_soluzioni(E,(appartenenza(E,L1), functor(E,_,1)), raccolta_soluzioni(X,(arg(1,E,X),atom(X)),[]),L2).

In versioni di Prolog diverse si possono avere differenti implementazioni di predicati di sistema per la raccolta di tutte le soluzioni di una meta. In Prolog/DEC-10 si hanno i seguenti:

bagof(Oggetto, Meta, Lista)

setof(Oggetto, Meta, Lista)

Differiscono tra loro per il fatto che **Lista** rappresenta in **bagof** un multinsieme (bag), cioè un insieme di elementi non ordinati con possibili duplicati, mentre in **setof** rappresenta un insieme di elementi senza ripetizioni e secondo l'ordinamento lessicografico standard adottato dal linguaggio.

Differiscono da **raccolta_solutions** per il fatto che, se non esistono soluzioni di **Meta**, falliscono anziché istanziare **Lista** alla lista vuota, e per il fatto che, mentre con **raccolta_solutions** tutti gli oggetti sono inseriti nella lista indipendentemente da eventuali istanziamenti alternativi delle variabili in **Meta** che non sono in condivisione con **Oggetto**, con **bagof** e **setof** questi possono essere raccolti in liste separate. Nell'esempio, la meta:

```
?- bagof(N, pratica(N, S), L).
```

produce, tramite ritorno indietro, le soluzioni alternative:

S = tennis

L = [Paolo,Anna,Lia];

S = Calcio

L = [Gianni,Marco];

S = corsa

L = [Linda];

S = nuoto

L = [Grazia];

S = salto

L = [Grazia]

Per avere lo stesso comportamento di **raccolta_solutions**, ossia una sola lista di tutte le soluzioni, il quesito va posto nella forma:

```
?- bagof(N, S^pratica(N, S), L).
```

dove " \wedge " denota un quantificatore esistenziale che lega esplicitamente le variabili in **S**, che non vengono quindi più trattate come variabili libere come nel caso precedente (l'uso ditale quantificatore esistenziale è superfluo all'esterno di **setof** e **bagof**).

Definendo:

```
sport_praticato_da(Sport, N) :- bagof(Persona, pratica(Persona, Sport), Lis), lunghezza(Lis, N).
```

oltre che:

```
?- sport_praticato_da(tennis, N).
```

è ora possibile il quesito:

```
?- sport_praticato_da(Sport, 1).
```

che porterà a produrre, mediante ritorno indietro, le tre soluzioni alternative:

Sport = corsa;

Sport = nuoto;

Sport = salto

sfruttando così la possibilità di usare il predicato **bagof** nonostante che la variabile **Sport** sia, al momento della sua chiamata, libera.

La possibilità di ottenere mediante ritorno indietro soluzioni alternative per le variabili libere in **bagof** e **setof** è correlata alla restrizione che la lista delle soluzioni non sia vuota: in caso contrario sarebbe infatti possibile un'infinità di alternative, portando ad un ciclo infinito.

Si noti che questa possibilità consente di fare verifiche sulla congruenza dei fatti espressi. Date ad esempio le seguenti clausole:

```
luogo_nascita( 'Rossi', 'Roma').
```

```
luogo_nascita( 'Verdi', 'Venezia').
```

...

```
luogo_nascita('Rossi', 'Napoli').
```

con associata a **luogo_nascita(Persona, Luogo)** l'interpretazione: "Persona è nato a Luogo", l'occorrenza di due fatti aventi lo stesso primo argomento e il secondo diverso è da considerare (prescindendo dalle omonimie, che si possono superare qualificando più ampiamente i dati anagrafici) un errore nei dati; può essere rilevato formulando ad esempio il quesito:

```
?- bagof(Luogo, luogo_nascita(Persona, Luogo), Luoghi), lunghezza(Luoghi, N), N \== 1,  
write('Errore di dati per: '), write(Persona), nl.
```

Alternativamente, ma un po' più laboriosamente, si può controllare che nella lista **Persone** costruita con il quesito:

```
?- raccolta_soluzioni(Persona, luogo_nascita(Persona, Luogo), Persone).
```

non siano presenti duplicati.

Note bibliografiche.

O'Keefe (1983) segnala l'insieme delle circostanze in cui è opportuno fare ricorso ai predicati di manipolazione della base di dati.

Il concetto e le modalità di utilizzo della generazione dilemmi sono discussi nei libri di Kowalski (1979a) e di Hogger (1984), e nell'articolo di Clark e McCabe (1982).

L'idea di realizzare una procedura priva di effetti collaterali e con la caratteristica di riuscire e fallire alternatamente, come interruttore, compare in Byrd (1980). Da Bundy e Welham (1977) è stata invece ripresa la procedura **stampa_condizionale**.

Warren (1982) mostra come l'aggiunta al Prolog di predicati come **setof** e **bagof** non costituisce una semplice estensione sintattica di comodo, ma rende il linguaggio effettivamente più potente. Un'implementazione del predicato **setof** in Prolog è stata presentata in Pereira e Porto (1981). La questione viene anche discussa in Clocksin e Mellissli (1981), ed in Sterling e Shapiro (1986). Una panoramica delle implementazioni esistenti per i predicati di raccolta di tutte le soluzioni di una meta compare in Naish (1985).

Sommario.

Dovrebbe a questo punto essere chiaro che i predicati di modifica della base di dati, e conseguentemente le procedure che li usano, si collocano ad un livello diverso rispetto a quello delle descrizioni effettuate nel solo linguaggio logico delle clausole di Horn; infatti essi influenzano la dimostrazione nel corso stesso della sua esecuzione, oppure - come nel caso di **raccolta_soluzioni** - fanno riferimento a più dimostrazioni.

Lo studente è dunque ora pienamente consapevole di come il Prolog, costituito dall'insieme delle clausole di Horn e dei predicati predefiniti extra-logici e meta-logici, consente di costruire programmi a vari livelli di sofisticazione, ed anche con vario grado di bilanciamento tra aspetti procedurali ed aspetti dichiarativi, tra logica e controllo.

15. Prova dei programmi

Dove si presenta il modello di Byrd del comportamento di una procedura ai fini della comprensione dell'esito della sua esecuzione, e si descrivono i predicati di sistema che facilitano la ricerca degli errori nei programmi. E dove si discutono i principali problemi ai quali si può andare incontro nella messa a punto di un programma Prolog, ed alcune indicazioni sul modo di strutturare i casi di prova.

Il modello di Byrd.

Il modello di Byrd, o modello della scatola a quattro porte, rappresenta il flusso di controllo attraverso una procedura Prolog. Le clausole della procedura sono immaginate disposte entro una scatola a quattro porte, corrispondenti alle quattro differenti possibilità che si presentano nel flusso di controllo in entrata od in uscita alla procedura. Di conseguenza il flusso di controllo viene visualizzato come un insieme di spostamenti all'interno od all'esterno delle varie scatole associate alle procedure del programma, attraverso tali punti di passaggio. Nella figura successiva le quattro porte sono state etichettate con i nomi **call**, **exit**, **redo** e **fail**, con il significato che segue.



Figura 15.1.

call. L'entrata nella scatola attraverso la porta cali indica l'invocazione della procedura, quando viene chiamata come meta: il sistema Prolog utilizza una clausola della procedura, tentando di soddisfare le sottomete presenti in essa. Le procedure invocate da tali sottomete vanno intese come scatole incorporate nella scatola principale. Questa azione prescinde completamente dal fatto che le varie operazioni di unificazione coinvolte siano possibili oppure no, e nulla è ipotizzato riguardo al risultato della chiamata.

exit. La freccia relativa a questa etichetta rappresenta una terminazione con successo della procedura interessata, ossia il soddisfacimento della meta per la quale essa era stata invocata. Il controllo esce dalla scatola per tornare al punto di provenienza.

redo (o back to). Indica il fallimento di una meta: il sistema Prolog sta dunque effettuando un ritorno indietro nel tentativo di trovare soluzioni alternative per le mete in precedenza soddisfatte. Questa porta riguarda la possibilità che l'interprete abbia in precedenza soddisfatto la procedura in questione ma debba ora riconsiderarla come conseguenza del fallimento di una meta successiva.

fail. La procedura fallisce, e la sua scatola viene abbandonata attraverso questa porta, se non è possibile alcun (ulteriore) soddisfacimento della meta per la quale la procedura è stata invocata. Il controllo passa allora all'esterno, ed il sistema continua l'operazione di ritorno indietro.

Se in un programma vi sono più chiamate alla stessa procedura, per distinguerle occorre immaginare di disporre di una nuova scatola in corrispondenza ad ogni nuova chiamata (in tal modo restano giustificati i concetti di **redo**, di **exit** o di **fail** della stessa scatola).

Gli strumenti di debugging disponibili nelle versioni del linguaggio che si ispirano al Prolog/DEC-10 forniscono un numero di invocazione, che compare nei messaggi in parentesi; è il numero assegnato ad una scatola quando si entra in essa attraverso una porta **call**: alla prima viene assegnato il numero 1, alla seconda il 2 e così via. Come si è detto, può trattarsi della stessa scatola invocata più volte, come per esempio nel caso di una procedura ricorsiva. Consideriamo, per fissare le idee, la seguente procedura:

p :- q, r.

q :- s.

q :- t.

r :- a, b.

s.

a.

La seguente sequenza rappresenta, con commenti aggiuntivi, i messaggi emessi da uno strumento di debugging avente le suddette caratteristiche, durante l'esecuzione di questa procedura attivata con il quesito:

?- p.

- (1) **Call p** - il quesito attiva la meta **p**,
- (2) **Call q** - che attiva la sua prima sottometà **q**,
- (3) **Call s** - che attiva con la prima clausola la sua sottometà **s**;
- (3) **Exit s** - **s** riesce,
- (2) **Exit q** - ed essendo l'unica sottometà di **q**, **q** riesce,
- (4) **Call r** - attivando la seconda sottometà **r** di **p**,
- (5) **Call a** - che attiva la sua prima sottometà **a**;
- (5) **Exit a** - **a** riesce,
- (6) **Call b** - attivando la seconda sottometà **b** di **r**;
- (6) **Fail b** - non essendovi clausole, **b** fallisce,
- (5) **Redo a** - attivando il ritorno indietro ad **a**;

- (5) **Fail a** - non essendovi altre clausole, **a** fallisce;
- (4) **Fail r** non essendovi altre clausole, **r** fallisce,
- (2) **Redo q** - attivando il ritorno indietro a **q**,
- (3) **Redo s** - che riprova **s**;
- (3) **Fail s** - non essendovi altre clausole, **s** fallisce,
- (7) **Call t** - attivando il ritorno indietro a **q**, che attiva **t** con la seconda clausola;
- (7) **Fail t** non essendovi clausole, **t** fallisce;
- (2) **Fail q** non essendovi altre clausole, **q** fallisce;
- (1) **Fail p** non essendovi altre clausole, **p** fallisce.

La successiva mostra l'albero AND OR della dimostrazione, in cui ora i nodi sono costituiti dalle scatole contenenti le procedure invocate: esse sono tante quante sono le sottomete (comprese quelle che non hanno procedure di definizione, e le cui scatole sono quindi vuote), più la metà iniziale; su ogni scatola è indicato a sinistra il numero di invocazione e a destra la sottometa (entrambi corrispondenti a quelli che compaiono nella sequenza sopra indicata). Gli archi orientati che connettono le diverse porte delle scatole visualizzano l'andamento del flusso di controllo; ciascuno di essi è etichettato dal numero di invocazione, seguito da **C** per **Call**, oppure **E** per **Exit**, o **R** per **Redo**, o **F** per **Fail**.

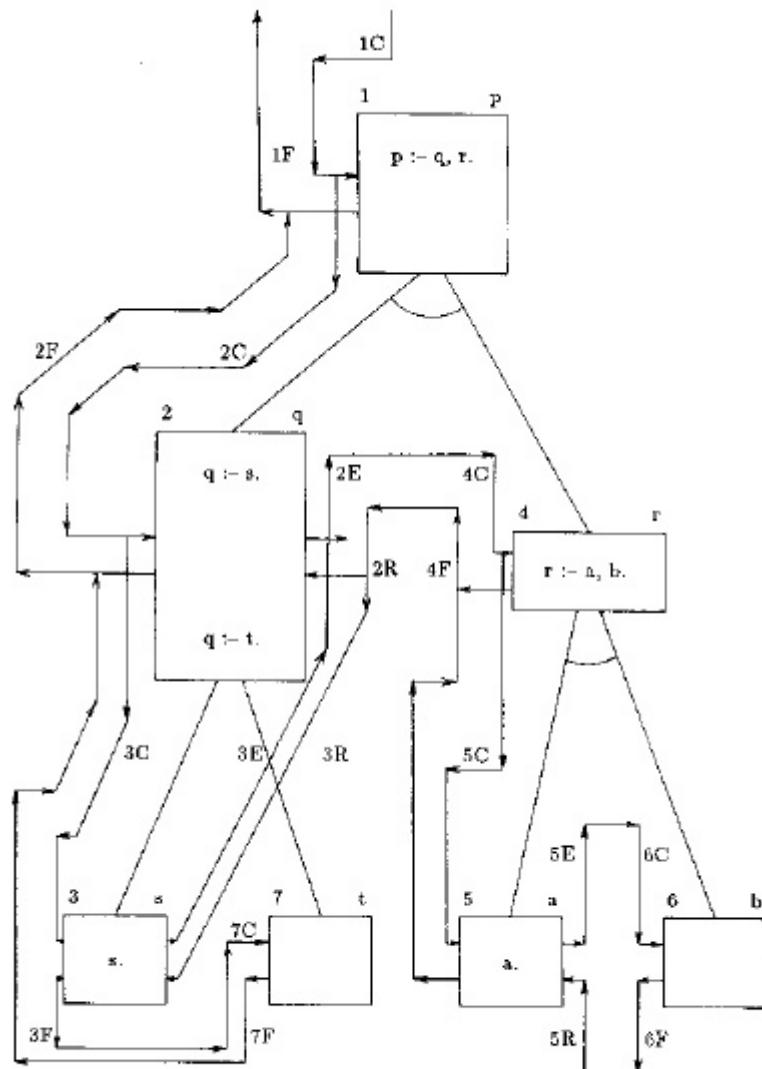


Figura 15.2.

Strumenti per la ricerca degli errori.

Il modello di Byrd individua la procedura quale primo elemento di attenzione e costituisce un concreto punto di partenza per gli strumenti di ausilio alla ricerca degli errori. Tutti i sistemi Prolog mettono a disposizione un insieme di procedure di debugging, espresse da predicati predefiniti, che possono però variare a seconda delle implementazioni. In generale, esse costituiscono mete che riescono sempre e non possono venire risoddisfatte; al momento del loro soddisfacimento viene emesso sul flusso standard di uscita un messaggio di conferma. Il loro scopo è di ottenere come effetto collaterale la memorizzazione e la visualizzazione delle informazioni specificate; di conseguenza, le esecuzioni richiederanno un maggiore spazio di memoria.

Di solito, per ciascun predicato di attivazione, per esempio:

debug

che attiva il debug mode, vi è un corrispondente predicato di disattivazione, come:

nodebug

che, al contrario, disattiva il debug mode (non memorizzando più alcuna informazione). In qualche caso vi è anche un predicato per conoscere lo stato corrente, per esempio:

debugging

il cui effetto collaterale è di fornire informazioni concernenti lo stato del debugging, in particolare indicando se il debug mode è attivato o meno.

Nel seguito vengono indicate le principali funzionalità che tali predicati forniscono, esemplificandone qualcuno, in una forma necessariamente specifica. Per conoscere quali predicati sono disponibili in una determinata versione di Prolog, e con quali modalità precise di utilizzo, va evidentemente consultato il relativo manuale di utente.

Predicati di tracciamento.

In generale sono disponibili procedure per il tracciamento dell'esecuzione di un programma; esso può essere esaustivo (relativo a tutte le mete) o selettivo (solo quelle indicate), e può essere compiuto a partire dal quesito, o direttiva, sino al termine dell'esecuzione oppure solo per parti di essa, secondo le opzioni dell'utente. Nella condizione normale non viene effettuato alcun tracciamento. I predicati di tracciamento completo, ad esempio **trace** per l'attivazione e **notrace** per la disattivazione, consentono all'utente di mantenersi informato dinamicamente su tutte le mete che il sistema Prolog cerca di dimostrare e su quelle che riesce effettivamente a dimostrare. Il relativo stato di sistema, che assume i valori on (abilitato) ed off (disabilitato), può venire esaminato e modificato.

I punti di spia consentono all'utente di restringere il tracciamento al sottoinsieme di predicati che lui stesso può indicare. Posizionando un punto di spia su un predicato si ottengono messaggi di tracciamento ogni volta che il sistema tenta di provare, o riesce a provare, mete aventi quel nome di predicato. Ad esempio, l'unico argomento della procedura:

spy Spec (o spy(Spec))

è uno specificatore (od una lista di specificatori) di punto di spia, cioè un nome di predicato, eventualmente seguito da un numero che ne rappresenta la molteplicità. Uno specificatore che consiste solo di un nome di predicato seleziona tutte le clausole la cui testa ha quel nome; uno specificatore che include anche un numero seleziona tutte le clausole la cui testa ha quel nome di predicato ed il cui numero di argomenti è uguale a quel numero. Il flusso di controllo attraverso le porte delle procedure specificate verrà, come conseguenza dell'esecuzione della meta **spy Spec**, sottoposto a tracciamento.

La procedura:

nospy Spec (o nospy (Spec))

cancella i punti di spia da tutte le procedure specificate da **Spec**, mentre invocando:

nospyall

vengono rimossi tutti i punti di spia precedentemente attivati.

Antenati di una meta.

Gli antenati (ancestors) di una meta sono le mete al cui soddisfacimento tale meta contribuisce se viene a sua volta soddisfatta. Il primo antenato di una meta **m**, detto meta genitrice di **m**, è la meta che ha determinato l'unificazione con la testa della clausola contenente **m**; l'**n**-esimo antenato di **m** è la meta genitrice dell'**n-1**-esimo antenato di **m**.

In alcuni sistemi Prolog sono disponibili predicati predefiniti che consentono di ottenere un elenco di tutti o di alcuni antenati di una certa meta. Per esempio, un'invocazione della procedura:

ancestor(N, X)

termina con successo se l'intero **N** è maggiore od uguale a zero e se la meta **ancestor** ha un antenato **N**-esimo; in caso positivo la variabile **X**, inizialmente libera, viene istanziata all'**N**-esimo antenato della meta. La meta:

ancestors(L)

viene soddisfatta unificando la variabile **L**, inizialmente non istanziata, con la lista delle mete antenate della clausola nella quale si trova **ancestors**; tale lista ha come primo elemento la meta genitrice e termina con l'antenato più remoto. La meta:

backtrace(N)

riesce se **N** è un numero intero, fornendo i primi **N** antenati della meta stessa (o tutti i suoi antenati, se questi sono in numero inferiore ad **N**), con qualsiasi istanziamento che sia già stato effettuato, in ordine inverso. Di conseguenza, **backtrace** può venire utilizzato come direttiva per esaminare dove ha avuto luogo un errore di esecuzione.

Informazioni sul programma.

La funzione principale di questo insieme di predicati consiste nel controllare e segnalare quali clausole o quali termini si trovano nella base di dati del sistema, in un momento qualsiasi dell'elaborazione.

Ad esempio, nella procedura:

listing(P)

l'argomento **P** può essere omesso, oppure specificare il nome ed eventualmente la molteplicità di uno o più predicati. Nel primo caso vengono elencate, sul flusso corrente di uscita, tutte le clausole del programma; nel secondo caso soltanto le clausole la cui meta di testa ha quel nome di predicato, nel terzo caso le clausole con quel nome e quella molteplicità del predicato di testa. La meta:

current_atom(Var)

permette di generare uno alla volta, mediante ritorno indietro, tutti gli atomi noti al sistema in quel momento dell'elaborazione; essi vengono mostrati istanziando via via ad essi la variabile **Var**.

La meta:

current_functor(N, F)

genera invece uno alla volta, mediante ritorno indietro, tutti i funtori noti al sistema in quel momento, e per ognuno di essi restituisce il suo nome ed il suo termine più generale, istanziando ad essi rispettivamente le variabili **N** ed **F**. Se **N** viene assegnato dall'utente, vengono generati soltanto i funtori dotati di quel nome, tramite l'istanziamento della variabile **F**.

La procedura:

current_predicate(N, F)

è simile a **current_functor**, ma genera soltanto i funtori che corrispondono a predici per i quali esiste una procedura: **N** è il nome di un predicato **F**.

Controllo dell'esecuzione.

Una volta che un programma è stato letto e caricato nella base di dati del sistema, l'interprete ha disponibili tutte le informazioni necessarie per la sua esecuzione; esse vengono dette stato di programma o stato di esecuzione. Lo stato corrente del programma può venire memorizzato su un file, per una futura esecuzione, mediante la direttiva:

?- save(file).

Tale procedura può essere invocata al livello dell'interprete od in una qualsiasi parte di un programma utente. Lo stato memorizzato in file può essere ripristinato assegnando il nome del file come argomento al comando **prolog**, come in:

?- prolog file.

Dopo la sua esecuzione l'interprete si troverà esattamente nello stesso stato precedente alla chiamata di **save**, ad eccezione degli eventuali files in quel momento aperti, che vengono automaticamente chiusi da **save**. In tal modo l'esecuzione avrà inizio a partire dalla metà immediatamente successiva alla chiamata di **save**.

È possibile interrompere la normale esecuzione di una direttiva agendo sul carattere di interruzione (interrupt) del proprio terminale (delete, CTRL Y e così via). Quando ha luogo un'interruzione, compare sul terminale un messaggio che richiede all'utente di scegliere una tra le opzioni previste, che solitamente sono le seguenti (od un loro sottoinsieme):

a (per abort) riporta il controllo all'interprete principale;

b equivale all'invocazione della procedura di sistema **break**;

c (per continue) continua semplicemente l'esecuzione, come se l'interrupt non avesse avuto luogo;

d attiva il debugging e continua l'esecuzione;

e (per exit) termina la sessione Prolog;

n equivale all'invocazione della meta **notrace**;

t equivale all'invocazione della meta **trace**.

I sistemi Prolog consentono di sospendere l'esecuzione di un programma alla chiamata di procedura successiva alla meta in corso, preservando il contesto dell'esecuzione e determinando una nuova invocazione dell'interprete principale. Chiamando la procedura predefinita **break** viene scritto sul terminale un messaggio del tipo:

Entering break

seguito da:

[Break (level 1)]

Il sistema si trova da quel momento in una nuova configurazione, nella quale l'utente può emettere direttive per la risoluzione di mete e fornire gli ingressi che desidera, esattamente come avviene per l'interprete principale. Se **break** viene invocato all'interno di un break-level, viene dato inizio ad un altro break-level, che avrà un numero superiore, e così via a qualunque grado di innestamento; il break-level indica pertanto il numero di invocazioni sospese per le quali è stato preservato il contesto. Si può entrare in uno stato di **break** in uno dei tre modi seguenti: eseguendo la meta **break**; selezionando l'opzione di interruzione **b**; causando l'insorgere di alcuni tipi di errori di esecuzione (dei quali si dirà più avanti).

Per chiudere uno stato di **break** occorre digitare il carattere di end-of-file (CTRL-Z o CTRL-D nella maggioranza dei sistemi) in risposta al normale prompt per direttive, "?-". L'uscita da uno stato di **break** riduce di una unità il break-level; uscendo dal break-level di primo livello viene ripristinata l'esecuzione precedentemente sospesa, a partire dalla chiamata di procedura in corrispondenza alla quale si era verificata la sospensione. Alternativamente, l'esecuzione interrotta può essere terminata invocando la procedura di sistema **abort**. In quest'ultimo caso non c'è bisogno del carattere di end-of- file per la chiusura del **break**, in quanto il controllo ritorna direttamente all'interprete principale.

Alcune versioni del linguaggio consentono l'accesso ai comandi del sistema operativo durante la sessione Prolog. Un caso tipico è la disponibilità della procedura:

system(S)

che effettua una chiamata al sistema operativo sottponendo ad esso come argomento **S** un comando.

Uso delle risorse.

In tutte le implementazioni sono presenti dei predicati predefiniti che consentono di ottenere informazioni sull'occupazione di memoria e sui tempi di esecuzione di un programma. Per esempio, un predicato quale:

cputime

segnala, sul file di uscita corrente, il tempo di CPU utente in secondi a partire dall'invocazione dell'interprete. Può essere usato per conoscere il tempo speso dal sistema nel tentativo di soddisfare una determinata meta o gruppo di mete oppure, invocato quale ultima meta di un programma, per ottenere il tempo di esecuzione globale dello stesso. L'invocazione di un predicato del tipo:

core (o heapused)

fornisce in uscita informazioni relative all'utilizzo della memoria a quel punto dell'elaborazione o della sessione, quali l'ammontare della memoria libera e l'ampiezza delle aree di memoria utilizzate dalle strutture di dati dinamiche interne. Alcuni sistemi dispongono di un'operazione di recupero dinamico della memoria occupata ma non più utilizzata (garbage collection). L'effettuazione di tale operazione si spiega osservando che l'interprete Prolog memorizza, ogni volta che una meta viene soddisfatta in maniera non deterministica, una serie di informazioni, che utilizza per il controllo del ritorno indietro, relative a mete già soddisfatte ed a variabili già istanziate nel corso del processo di unificazione. Come si ricorderà dai capitoli precedenti, la quantità di informazione conservata per la gestione del ritorno indietro può essere minimizzata mediante un adeguato utilizzo nel programma dei predicati "!" e **fail** e limitando, per quanto possibile, la realizzazione di profondi innestamenti di mete e sottomete.

L'operazione di garbage collection consiste nel liberare, nelle tavole di nomi del sistema, lo spazio occupato da atomi ai quali non viene fatto più riferimento in nessuna delle clausole presenti nella base di dati. Ogni volta che una clausola viene cancellata, la sua rappresentazione in memoria viene marcata ma la clausola non viene rimossa fisicamente; analogamente, quando un file viene chiuso vengono marcate, ma non rimosse, le aree di memoria relative alla sua gestione. Queste aree vengono raccolte durante la garbage collection, e lo spazio recuperato viene reso nuovamente disponibile.

Nei sistemi che la ammettono, la garbage collection viene invocata automaticamente dall'interprete dopo ogni direttiva a livello principale, dopo **abort**, **consult** o **reconsult**, ma può anche essere abilitata da una meta del programma effettuando una chiamata ad un predicato predefinito, come **gc** o **trimcore**. Questo può avere un argomento, da istanziare ad un intero, che specifica la soglia desiderata di memoria occupata sotto la quale la garbage collection non è richiesta (essendo un'operazione che ovviamente impegna un certo tempo di esecuzione). La disabilitazione si ottiene invocando un predicato con effetto contrario, ad esempio **nogc**.

Trattamento degli errori.

Eventuali errori sintattici vengono rilevati durante la fase di consultazione o riconsultazione di un programma, od all'interno di direttive da eseguire. Ogni clausola, direttiva od in generale qualsiasi termine letto mediante la procedura di sistema **read** che non si accorda con i requisiti sintattici viene scritta sul terminale non appena viene letta, con l'indicazione del punto di occorrenza ed eventualmente della natura dell'errore.

Gli errori sintattici non compromettono l'operazione di consultazione o di riconsultazione se non per il fatto che le clausole od i comandi contenenti errori sintattici vengono ignorati; tutte le altre clausole presenti nel file interessato vengono lette e consultate correttamente. Se l'errore sintattico avviene invece durante la formulazione di un quesito o di un comando, si rende necessario ripeterlo. Data la particolare semplicità della sintassi Prolog, è di solito immediato individuare la fonte d'errore e porvi rimedio.

Per errori di esecuzione si intendono invece gli errori, che avvengono durante l'esecuzione di un quesito o di una direttiva, diversi da quelli generati dal predicato di sistema **read**. Il tipo più ricorrente di errore di esecuzione è il tentativo di eseguire una procedura di sistema con argomenti scorretti od incongruenti; altri possibili errori riguardano l'esaurimento della memoria, che può avversi ad esempio quando il programma entra in un ciclo infinito, od il tentativo di effettuare divisioni per zero.

In generale, dopo l'emissione di una breve spiegazione dell'errore, l'interprete presenta i seguenti possibili comportamenti. Se l'errore è di notevole gravità (un caso tipico è l'esaurimento della memoria), l'esecuzione viene terminata, con segnalazione su terminale. Alcuni errori sono di gravità tale da forzare la terminazione della sessione Prolog; fra questi vanno segnalati errori di ingresso/uscita sul flusso corrispondente al file di sistema **user**, o la digitazione del tasto di interrupt durante il caricamento del sistema, od ancora il superamento del massimo numero possibile di atomi nella base di dati. Se invece l'errore è meno grave, il sistema entra automaticamente in uno stato di break. Questo è utile per l'esplorazione e l'eventuale modifica della base di dati, e consente un'operazione di ritorno indietro degli antenati della meta che ha provocato la segnalazione dell'errore. L'uscita dallo stato di **break** determina la ripresa dell'esecuzione come se la meta che ha causato l'errore fosse fallita.

Verifica di un programma.

In Prolog il fallimento di una meta può essere imputabile a diverse cause: l'assenza di clausole, la mancata riuscita dell'unificazione con le clausole disponibili, il mancato soddisfacimento di una o più sottomete, il fatto che le soluzioni prodotte vengano poi respinte, o che tutte le soluzioni possibili siano già state trovate. Di conseguenza gli errori sono a volte molto difficili da rilevare e da comprendere. Si veda come esempio la seguente definizione erronea della relazione di inversione di liste (Si può confrontarla con [quella corretta](#)):

```
inversione([], []).  
inversione([A | B], Q) :- inversione(B, Z), concatenazione(Z, A, Q).  
concatenazione([], L, L).  
concatenazione([T|L1], L2, [T|L3]) :- concatenazione(L1, L2, L3).
```

Il quesito **inversione([a, b], [b, a])** fallisce: l'utente deve decifrare l'origine dell'errore. Il problema è acuito dal fatto che nei programmi con un ampio spazio di ricerca un semplice errore di scrittura, ovvero di incompatibilità fra termini, può condurre ad una enorme quantità di computazione o addirittura provocare la non terminazione del programma. Si veda, come esempio:

fattoriale(N, F), uguale(F, vuoto)

in cui si considera un **N** tale che il suo fattoriale risulti uguale al termine vuoto (una costante non numerica!); la computazione risultante entrerà in un ciclo infinito.

L'individuazione di un metodo di verifica sistematico è connessa con la questione della realizzazione di una biblioteca di moduli da una parte, e con la strutturazione a livelli di un programma Prolog dall'altra, che può venire utilmente sfruttata nella fase di verifica. Si può sviluppare la prova (testing) di un programma a partire dalle procedure di più basso livello: se queste sono procedure di libreria, il loro comportamento è già completamente noto in partenza (dunque non è necessario sviluppare alcuna verifica), altrimenti è necessario procedere accertando che i risultati forniti siano corretti e che sia noto se il comportamento è deterministico o meno, cioè se in caso di ritorno indietro forniranno, dove possibile, un'altra soluzione oppure no. Quando è stata completata la prova di un insieme di procedure che, opportunamente combinate, consentono di realizzare una funzionalità di livello più alto, si passa a provare quest'ultima, seguendo gli stessi criteri: presenza della soluzione desiderata, comportamento noto in caso di ritorno indietro. La verifica prosegue fino a che viene provata la funzionalità di livello più alto, che corrisponde alla meta principale, ossia al quesito formulato dall'utente.

Nella prova delle procedure si possono seguire alcuni criteri pragmatici, quali i seguenti. Se le procedure sono organizzate per casi, vanno provati i singoli casi (in dipendenza dalla forma degli

ingressi). Va accertata, nel caso di procedure ricorsive, la presenza di almeno una condizione limite. È necessario curare particolarmente le procedure che modificano la base di dati: esse sono spesso fonte di errori, in quanto ci si può attendere che vengano effettuate unificazioni con clausole che sono state precedentemente cancellate dalla base di dati.

Un esempio di un insieme di casi di prova basato su ingressi tali da esercitare ciascuno una delle clausole della procedura è il seguente, che si riferisce alla procedura coefficiente_binomiale (i commenti indicano le risposte attese):

```
?- coefficiente_binomiale(3, 3, C). /* 1 */
```

```
?- coefficiente_binomiale(7, 3, C). /* 35 */
```

```
?- coefficiente_binomiale(6, 3, C). /* 20 */
```

```
?- coefficiente_binomiale(7, 5, C). /* 21 */
```

Essi attivano, nell'ordine, le quattro regole che definiscono la procedura.

Sfruttando il fatto che ogni quesito può essere scritto come regola, si possono predisporre i casi di prova in forma di regole, in modo da eseguirli con maggiore facilità. Ciò risulta particolarmente comodo quando le strutture che compaiono come argomenti di un quesito sono complesse, ed è quindi scomodo scriverle interattivamente a livello dell'interprete principale (con la possibilità di dovere riscrivere tutto per un banale errore di battitura). Consideriamo ad esempio i quesiti relativi alla procedura appiattimento_lista. Si può definire la regola:

```
prova(L) :- appiattimento_lista([1,2,3, (4,5, [[6], 7]],8], L).
```

e poi formulare il più semplice quesito:

```
?- prova(L).
```

Più comoda ancora può risultare una definizione separata della struttura che si vuole usare mediante un fatto, utilizzabile poi in più regole diverse. Nell'esempio, definendo:

```
lista([1,2,3,[4,5,[[6],7]],8], L).
```

si possono poi avere le seguenti regole per provare per questo stesso caso le tre versioni della procedura:

```
prova_1(L) :- lista(L1), appiattimento_lista(L1, L).
```

```
prova_2(L) :- lista (L1), appiattimento_lista_1(L1, L).
```

```
prova_3(L) :- lista(L1), appiattimento_lista_3(L1, L).
```

L'unificazione provvede a fornire l'ingresso voluto, mentre nel quesito comparirà solo la variabile d'uscita, dalla quale si otterrà la risposta.

Ancora, poiché anche le risposte possono coinvolgere strutture complesse e quindi di non immediata lettura, e poiché per controllarne la correttezza occorre comunque conoscerle in anticipo,

si può nuovamente trarre profitto dalle caratteristiche del linguaggio per semplificare il lavoro. In questo caso si tratta del fatto che le procedure che definiscono relazioni (tranne quelle il cui scopo consiste solo nel creare effetti collaterali) possono sempre essere utilizzate per verificare la relazione definita; usandole in questo modo, occorre fornire come argomenti non solo gli ingressi ma anche le uscite. Il quesito avrà pertanto una risposta unicamente di tipo sì o no, fornendo così un controllo automatico dei risultati. Per la procedura in esame avremo ad esempio:

prova :- lista_1(L1), lista_2(L2), appiattimento_lista(L1, L2).

lista_1([atomo, [elemento,1], a, (c, (d, [e, f]))]).

lista_2([atorno, elemento, 1, a, c, d, e, f]).

Una volta provate le procedure (particolarmente quelle che fanno parte di una biblioteca), può essere conveniente conservare i casi di prova utilizzati, da riusare in caso di successive modifiche e quindi necessità di riverifica (che viene chiamata prova di regressione). A tale scopo si possono definire delle regole che collegano insieme diverse procedure, in modo tale da esercitarle tutte in un'unica esecuzione, con controllo automatico dei risultati. Per esempio, partendo dall'albero (definito come fatto) corrispondente alla [Figura 9.1](#), la regola che segue esercita alcune delle procedure definite nelle [Strutture di dati e programmi](#), ottenendo come risultato finale quello di [Figura 9.2](#).

alb(albero(albero(albero (vuoto, 4, vuoto), 5, albero(vuoto, 6, vuoto)), 7, albero(vuoto, 8, vuoto))).

prove: alb(A), albero_binario(A), in_albero_ordinato(5, A), inser_in_albero(A, 3, A1), cancell_da_albero(A1, 3, A2), scrittura_albero(A2).

Note bibliografiche.

Il modello della scatola a quattro porte è stato proposto in Byrd (1980).

Sommario.

Lo studente ha ora una conoscenza di massima delle possibilità di uso degli strumenti indirizzati alla ricerca degli errori, ed anche uno schema generale di organizzazione della verifica di un programma.

Questo libro ha cercato di fornire gli elementi introduttivi necessari ad iniziare un'attività di programmazione logica, e di fornire degli spunti riguardo ai vari aspetti coinvolti. Sta ora alla volontà ed alla fantasia dello studente applicarli, approfondirli ed integrarli.