Laurea in Ingegneria Informatica
Facoltà di Ingegneria
Università di Roma "La Sapienza"

Dispense del Corso di
Intelligenza Artificiale

Anno Accademico 2003/04

Linguaggi per l'Intelligenza Artificiale

Daniele Nardi *Giorgio Grisetti*

Dipartimento di Informatica e Sistemistica
Università di Roma "La Sapienza"
via Salaria 113, 00198 Roma, Italia
{nardi,grisetti}@dis.uniroma1.it

Indice

1 LISP e linguaggi funzionali **7**
1.1 Principi di programmazione funzionale 7
1.2 Il tipo di dato lista 8
1.3 Il nucleo applicativo del LISP 11
1.4 Programmazione tramite funzioni ricorsive 18
1.5 Altre caratteristiche del LISP 20

2 PROLOG e programmazione logica **23**
2.1 Principi di programmazione logica 24
2.1.1 La base delle conoscenze 26
2.1.2 Interrogazione del sistema 27
2.1.3 Regole ricorsive 28
2.1.4 L'esecuzione dei programmi PROLOG 29
2.1.5 Modello operazionale 31
2.1.6 Esercizi 35
2.2 Termini e strutture dati 36
2.2.1 Termini 36
2.2.2 Liste 36
2.2.3 Modello computazionale per il linguaggio con i termini 37
2.3 Programmazione in PROLOG 39
2.3.1 Definizione dei numeri naturali 39
2.3.2 Operatori per dati di tipo numerico: il predicato `is` 40
2.3.3 Primitive per la manipolazione di liste 41
2.3.4 Alberi binari 44
2.4 Negazione e taglio 46
2.4.1 Modificare l'ordine di ricerca: il taglio 46
2.4.2 Negazione in PROLOG 48
2.5 Esercizi Riepilogativi 49

3 Applicazioni AI in PROLOG **51**
3.1 Problemi di ricerca in PROLOG 51
3.1.1 Esercizi 56
3.2 Classici AI 57

Introduzione

Lo sviluppo dell'Intelligenza Artificiale (IA) ha avuto molta influenza in diversi campi dell'Informatica, poiché la realizzazione dei sistemi di IA ha portato allo sviluppo di tecniche e strumenti innovativi. Questo è certamente il caso dei linguaggi di programmazione: un campo nel quale molti linguaggi sono stati sviluppati inizialmente per rendere più efficiente la realizzazione di sistemi di IA.

In particolare lo studio di metodi e tecniche per la rappresentazione della conoscenza ha portato alla realizzazione di ambienti di sviluppo per sistemi basati sulla conoscenza, che offrono al programmatore la possibilità di sviluppare un sistema formalizzando nel linguaggio di rappresentazione le conoscenze sul dominio e offrendo strumenti per derivare le conseguenze delle conoscenze specificate. I linguaggi utilizzati per rappresentare la conoscenza si potrebbero anch'essi considerare linguaggi di programmazione *dichiarativa* nel senso che consentono la specifica del problema, lasciando agli strumenti offerti dal sistema il compito di trovare algoritmicamente la soluzione. I linguaggi logici e funzionali LISP e PROLOG si trovano in una posizione intermedia, in quanto consentono forme di programmazione dichiarativa, ma presentano, a tutti gli effetti anche le caratteristiche di un linguaggio di programmazione procedurale. In questa dispensa l'attenzione è rivolta ai linguaggi ed alle tecniche di programmazione, trascurando i linguaggi di rappresentazione della conoscenza e tutte le problematiche di formalizzazione delle conoscenze relative ad un problema.

Nell'ambito della realizzazione di sistemi IA si pone spesso la necessità di sviluppare in tempi molto rapidi delle soluzioni prototipali al problema di interesse. Infatti molto spesso la realizzazione finale si basa sulla sperimentazione effettuata tramite prototipi. Per far fronte a questa necessità occorrono tecniche e strumenti che favoriscano la prototipazione rapida. I linguaggi di programmazione per l'IA sono quindi incorporati in ambienti di sviluppo in cui l'interprete diventa lo strumento principale di programmazione e fornisce funzionalità di *debugging* interattivo.

Classificazione dei Linguaggi di programmazione

Molti linguaggi di programmazione sono stati sviluppati negli anni, e per questa ragione si parla di Torre di Babele, riferendosi ai problemi di diffusione di strumenti informatici legati all'uso, in fase di sviluppo, di diversi linguaggi di programmazione. D'altra parte questo fenomeno è inevitabile ed inarrestabile, risultando intrinseco alla natura del linguaggio come strumento di comunicazione. L'evoluzione della comunicazione uomo-elaboratore crea dialetti e gerghi, allo stesso modo in cui evolve e si adatta nella comunicazione tra gli uomini il linguaggio naturale che diventa lessico familiare e gergo di cerchie più o meno ristrette di persone.

In genere un elaboratore offre la possibilità di usare uno o più linguaggi di comandi ed uno o più linguaggi di programmazione. Questi due tipi di linguaggi hanno, nella maggior parte dei casi, scopi distinti: i linguaggi di comandi vengono utilizzati per la creazione, l'esecuzione e la gestione dei programmi; i linguaggi di programmazione vengono utilizzati dall'utente per scrivere i programmi che risolvono i problemi specifici alle sue applicazioni. La distinzione tra linguaggio di comandi e linguaggio

di programmazione deriva da esigenze di costruzione piuttosto che da un reale vantaggio per l'utente. Infatti, da un punto di vista realizzativo, è conveniente separare le operazioni che consentono l'utilizzo dell'elaboratore (linguaggio di comandi), dal linguaggio di programmazione, che permette la specifica di operazioni di carattere più generale.

I linguaggi di programmazione evoluti che sono stati introdotti a partire dagli anni '50 sono detti linguaggi ad alto livello; essi sono stati progettati con lo scopo di rendere il linguaggio indipendente dalle caratteristiche dell'elaboratore e più vicino alla logica del programmatore. I programmmi scritti in un linguaggio ad alto livello non sono direttamente eseguibili dall'elaboratore, ma devono essere prima tradotti da un interprete o da un compilatore nel linguaggio macchina dell'elaboratore. Osserviamo inoltre che i linguaggi ad alto livello consentono di utilizzare lo stesso programma anche su macchine di tipo diverso.

La maggior parte dei linguaggi ad alto livello sono basati sull'interpretazione delle variabili come celle di memoria dell'elaboratore. Essi pertanto consentono istruzioni di due diversi tipi:

1. istruzioni che operano sulle variabili, o modificando il contenuto della memoria tramite l'assegnazione di un valore ad una variabile (istruzione di lettura e di istruzione di assegnazione), o utilizzando il valore della variabile (istruzione di stampa);

2. istruzioni che determinano le modalità di esecuzione di altre istruzioni.

Linguaggi di questo tipo sono detti linguaggi imperativi o anche linguaggi di Von Neumann, perchè fanno riferimento all'omonimo modello di calcolo. Come i linguaggi di basso livello essi fanno riferimento al modello di esecuzione delle istruzioni fornito dall'elaboratore: un programma è cioè costituito da una sequenza di istruzioni il cui effetto è quello di modificare il contenuto della memoria dell'elaboratore; in questo modello assume un ruolo fondamentale l'istruzione di assegnazione. A questo proposito J. Backus (che è considerato il padre del FORTRAN) scrive:
Le differenze tra il FORTRAN e l'ALGOL 68, sebbene considerevoli, sono meno significative del fatto che entrambi sono basati sullo stile di programmazione proprio dell'elaboratore di Von Neumann.

I linguaggi per l'IA appartengono ad altre categorie di linguaggi di programmazione che prescindono dal modello di funzionamento dell'elaboratore, cercando di fornire un mezzo espressivo per specificare il compito da eseguire in modo semplice e sintetico. Un programma può essere allora considerato come l'effetto dell'interazione di un insieme di oggetti (linguaggi orientati ad oggetti), oppure come il calcolo del valore di una funzione (linguaggi funzionali), oppure ancora come la dimostrazione della verità di una asserzione (linguaggi logici o dichiarativi). Questi tipi di linguaggi vengono detti non Von Neumann , proprio perchè svincolati dal modello su cui si basano i linguaggi imperativi.

I linguaggi per l'IA sono linguaggi non Von Neumann, in particolare il il LISP è il primo linguaggio funzionale e nasce alla fine degli anni '50 (nello stesso periodo del FORTRAN); la programmazione Logica si sviluppa negli anni '70 nell'ambito

di progetti per la dimostrazione automatica di teoremi el'analisi del linguaggio naturale. Anche per qualto riguarda i linguaggi orientati ad oggetti esso sono stati oggetto di sviluppo e di applicazione nell'ambito dell'IA. In particolare, SMALLTALK, uno tra i primi linguaggi ad oggetti ha portato alla realiazzazione di interfacce uomo-macchina. Tuttavia, con il passare degli anni la programmazione ad oggetti ha assunto un ruolo complementare rispetto a quello delle altre tecniche di programmazione, in quanto rivolto principalmente alla organizzazione dei dati e non al controllo del flusso di esecuzione o al modello computazionale. In questa dispensa ci limiteremo quindi a considerare i linguaggi funzionali ed i linguaggi logici.

Scopi e sommario

L'obiettivo di questa dispensa è di fornire una introduzione ai due linguaggi di programmazione sviluppati nel campo dell'IA che sono alla base di prototipi sviluppati nei laboratori di ricerca e di molte applicazioni realizzate in diversi settori dell'IA: il LISP ed il PROLOG. La conoscenza di questi linguaggi non solo consente di approfondire gli aspetti realizzativi dei sistemi di Intelligenza Artificiale, ma offre anche la possibilità di approfondire le tecniche di programmazione su cui si basano tali linguaggi.

La parte relativa al LISP ed alla programmazione funzionale è limitata ad una introduzione del principio della programmazione funzionale attraverso il nucleo applicativo del LISP che viene sviluppato nel Capitolo 1.

Per quanto riguarda la programmazione logica ed il PROLOG, vengono dapprima illustrate le caratteristiche di base del linguaggio, restringendo i termini a costanti e variabili, in modo da poter introdurre un modello computazionale semplificato. Vengono quindi introdotti i termini con simboli di funzione e sviluppate le tecniche di costruzione e manipolazione delle strutture dati. Vengono infine approfondite le caratteristiche del modello computazionale, con gli operatori di taglio e negazione (Capitolo 2).

Infine, vengono presentati degli esempi di programmi PROLOG relativi a tecniche di ricerca nello spazio delle soluzioni e a problemi classici di IA (Capitolo 3).

Ringraziamenti

Desideriamo ringraziare l'Ing. Marco Benedetti per il materiale messo a disposizione per la preparazione delle esercitazioni del corso. In particolare, nella parte relativa all'uso delle tecniche di ricerca in Prolog abbiamo attinto dalle sue esercitazioni. Inoltre, alcuni esercizi sono presi dal testo [5], specialmente nella parte delle applicazioni IA in Prolog.

Capitolo 1

LISP e linguaggi funzionali

La classe dei *linguaggi funzionali* comprende i linguaggi basati sul concetto di funzione e di applicazione di una funzione ad argomenti; per questa ragione essi sono anche detti *linguaggi applicativi*. L'origine dei linguaggi funzionali risale alla fine degli anni '50, quando J. Mc Carthy definì il linguaggio LISP (acronimo per LISt Processing), con l'obbiettivo di rendere agevole la manipolazione di informazione simbolica, cioè informazioni di tipo non numerico. In questo capitolo vengono inizialmente discussi i principi su cui si basa la programazione funzionale. Viene poi introdotto il tipo di dato lista usando la notazione parentetica del LISP, presentata la sintassi del nucleo applicativo del LISP e discussi i metodi di valutazione. Vengono quindi presentati alcuni esempi di programmazione di funzioni ricorsive, ed, infine, vengono discusse brevemente altre caratteristiche del LISP e dei linguaggi funzionali in generale.

1.1 Principi di programmazione funzionale

Un programma scritto in un linguaggio funzionale è costituito da un insieme di definizioni di funzioni. L'esecuzione del programma consiste in genere nel calcolo del valore di una o più funzioni. Come nell'usuale calcolo matematico, per ottenere il valore di una funzione è necessario specificare il valore dei suoi argomenti; in questo caso si dice che la funzione viene applicata agli argomenti, e la coppia

$$< funzione, listadiargomenti >$$

è un'espressione che prende il nome di *applicazione*. Pertanto, la valutazione di un'applicazione comporta la valutazione delle espressioni che fungono da argomenti. Queste possono essere costanti o variabili, ed in questo caso la valutazione restituisce il valore ad essi associato, oppure a loro volta applicazioni di cui occorre calcolare il valore per poterlo usare nella valutazione della applicazione più esterna. Supponiamo, ad esempio, che sia definita la funzione *succ* che ha per argomento un numero naturale e ne calcola il successore. Pertanto, la valutazione dell'applicazione $succ(5)$ restituisce 6 come risultato, mentre la valutazione dell'espressione $succ(succ(3))$ dà come risultato 5 (ottenuto applicando la funzione *succ* al risultato della valutazione dell'espressione $succ(3)$). In generale, il calcolo del valore di una funzione può anche

richiedere la valutazione di una applicazione della funzione stessa: in tal caso si ha una definizione *ricorsiva* della funzione. L'insieme delle espressioni che caratterizzano un linguaggio funzionale utilizza un insieme di funzioni base, predefinite nel linguaggio (ad esempio le funzioni definite sui numeri naturali) ed è completato dal condizionale. Esso è un'espressione del tipo

if *predicato* **then** *espressioneThen* **else** *espressioneElse*

in cui il *predicato* è un'espressione che può assumere i valori di verità VERO e FALSO. La valutazione del condizionale è data dalla valutazione dell'*espressioneThen* se il valore del predicato è VERO e dalla valutazione dell'*espressioneElse* se il valore del predicato è FALSO. Per poter usare il linguaggio occorre specificare che cosa sono gli argomenti ed i risultati della valutazione delle espressioni, o, più precisamente, il tipo di dato su cui si opera. Se consideriamo ad esempio il tipo di dato dei numeri naturali possiamo definire la funzione per il calcolo del fattoriale di un numero n nel seguente modo:

$$\textit{fatt} \leftarrow \textbf{if } n = 0 \textbf{ then } 1 \textbf{ else } n * \textit{fatt}(n-1)$$

Il simbolo $\leftarrow$ significa "definita come". L'occorrenza del nome *fatt* nella parte destra della definizione mostra che la funzione fattoriale è definita ricorsivamente. In essa compaiono i simboli "=", "*" e "-" che si riferiscono rispettivamente al predicato di uguaglianza ed alle funzioni di moltiplicazione e sottrazione definite nel tipo di dato dei numeri naturali.La valutazione dell'applicazione *fatt*(3) richiede la valutazione dell'espressione 3 * *fatt*(2), che a sua volta richiede quella di 3 * 2 * *fatt*(1). La valutazione termina quando l'argomento della applicazione della funzione *fatt* è 0: la valutazione di *fatt*(0) fornisce infatti il valore 1, senza richiedere la valutazione di altre applicazioni. Pertanto, il risultato complessivo è dato dal prodotto 3 * 2 * 1 * 1. Usualmente i linguaggi funzionali sono definiti sul tipo di dato delle liste. Queste garantiscono la flessibilità necessaria per esprimere adeguatamente strutture dati anche molto complesse.

1.2 Il tipo di dato lista

Sia dato un universo di atomi $\mathcal{A}$, il tipo di dato lista $\mathcal{L}$ è definito induttivamente nel modo seguente:

- () appartiene ad $\mathcal{L}$;
- $(e\ l)$ appartiene ad $\mathcal{L}$, se l è una lista ed e è un atomo o una lista.

Con questa definizione si possono costruire liste di atomi

$$(a_1 ... a_n)$$

e liste di liste o alberi, come ad esempio,

$$((a_{11}a_{12})((a_{211}a_{212})(a_{221}a_{222})))$$

In LISP sono definiti come atomi sia i numeri, che qualsiasi identificatore preceduto dal simbolo `'` (`QUOTE` nel gergo LISP). Ad esempio `(3 4 5)` e `('a 'b)` sono due liste ammesse dal LISP, la seconda delle quali è una lista contenente i due atomi `a` e `b`. Inoltre esistono due atomi denominati `T` e `NIL` che venono usati per rappresentare i valori di verità True e False, e che in seguito useremo per definire funzioni che hanno valori booleani, cioè predicati. Va tuttavia osservato che l'atomo `NIL` rappresenta anche la lista vuota, viene cioè usato come sinonimo di `()`. Questo duplice uso dell'atomo `NIL` è spesso fonte di confusione, ed occorre prestare un po' di attenzione per interpretarlo correttamente.

Consideriamo ora gli operatori definiti sulle liste iniziando ad usare la notazione parentetica del LISP, che prevede la scrittura di una applicazione con le parentesi che racchiudono sia il nome della funzione che gli argomenti. Pertanto l'applicazione della funzione *succ* al valore 4, che in notazione matematica di solito si scrive *succ*(4), in LISP diventa `(succ 4)`. Indicheremo inoltre con $\mathcal{E}$ l'insieme degli atomi e delle liste, con $\mathcal{O}$ l'insieme di tutti i dati ammessi dal linguaggio e con $\mathcal{B}$ l'insieme dei valori di verità True e False.

Costruzione

L'operatore `CONS` permette di definire una lista. In particolare,

$$\texttt{CONS} : \mathcal{E} \times \mathcal{L} \rightarrow \mathcal{L}$$

ha due argomenti e costruisce una nuova lista aggiungendo l'elemento corrispondente al primo argomento in testa ad una lista uguale al secondo argomento.

$$(\texttt{CONS}\ e\ (e_1 ... e_n)) = (e\ e_1 ... e_n)$$

Una lista si costruisce quindi a partire dalla lista vuota tramite la funzione `CONS`. Ad esempio

```
(CONS 3 ()) = (3)
(CONS 4 (CONS 3 ())) = (4 3)
```

Una lista di liste si costruisce come una lista di atomi usando liste anziché atomi come primo argomento della funzione CONS. Ad esempio

```
(CONS (CONS 3 ()) (CONS 4 ())) = ((3) 4)
```

Selezione

In LISP sono definite due funzioni di selezione su liste che restituiscono, rispettivamente, il primo elemento di una lista e la lista a cui questo è stato aggiunto. In particolare la prima funzione

$$\texttt{FIRST} : \mathcal{L} \rightarrow \mathcal{E}$$

restituisce il primo elemento di una lista non vuota

$$(\texttt{FIRST}\ (e\ e_1...e_n)) = e$$

Ad esempio, se consideriamo la lista `(4 3)` il primo elemento si ottiene applicando la funzione

```
(FIRST (CONS 4 (CONS 3 ()))) = 4
```

Si noti che se consideriamo la lista `((4) 3)` per selezionare l'atomo `4` occorre applicare due volte la funzione `FIRST`, poiché il primo elemento della lista è a sua volta una lista

```
(FIRST (FIRST (CONS (CONS 3 ()) (CONS 4 ())))) = 3
```

La seconda funzione di selezione è la funzione

$$\texttt{REST} : \mathcal{L} \rightarrow \mathcal{L}$$

che restituisce il resto di una lista non vuota, cioè una lista ottenuta da quella in ingresso eliminando il primo elemento

$$(\texttt{REST}(e\ e_1...e_n)) = (e_1...e_n)$$

Se consideriamo di nuovo la lista (4 3) applicando la funzione `REST` si ottiene la lista privata del primo elemento

```
(REST (CONS 4 (CONS 3 ()))) = (3)
```

Se consideriamo la lista `((4) 3)` per selezionare l'atomo `3` occorre applicare prima la funzione `REST` e poi la funzione

```
(FIRST (REST (CONS (CONS 3 ()) (CONS 4 ())))) = 4
```

<u>Test</u>
Introduciamo ora i predicati del tipo di dato lista, che in LISP corrispondono a funzioni che restituiscono valore `T` o `NIL`, ed alcuni predicati che riguardano gli atomi.

$$\texttt{ATOM} : \mathcal{O} \rightarrow \mathcal{B}$$

restituisce il valore `T` se l'argomento è un atomo, altrimenti `NIL`. In particolare

```
(ATOM ()) = NIL
```

poiché la lista vuota `()` è sinonimo dell'atomo NIL.

$$\texttt{NULL} : \mathcal{O} \rightarrow \mathcal{B}$$

restituisce il valore `T` se l'argomento è la lista vuota, altrimenti `NIL`.

$$\texttt{LISTP} : \mathcal{O} \rightarrow \mathcal{B}$$

restituisce il valore `T` se l'argomento è una lista, altrimenti `NIL`.

$$\texttt{EQ} : \mathcal{A} \times \mathcal{A} \rightarrow \mathcal{B}$$

restituisce il valore `T` se i due argomenti sono lo stesso atomo, altrimenti `NIL`.

$$\texttt{EQUAL} : \mathcal{L} \times \mathcal{L} \rightarrow \mathcal{B}$$

restituisce il valore `T` se i due argomenti sono liste uguali, altrimenti `NIL`. Si noti che la definizione di lista considerata in questo capitolo prevede che la funzione `CONS` abbia sempre una lista come secondo argomento, in modo tale che la costruzione di una lista richiede sempre di partire dalla lista vuota. In effetti, il LISP prevede anche la possibilità di applicare la funzione `CONS` con un atomo come secondo argomento. Ad esempio

```
(CONS 'a 'b) = (a. b)
```

In questo caso il risultato è una struttura dati che prende il nome di *coppia puntata*, e permette una gestione della memoria a livello di locazione. Non entreremo qui in ulteriori dettagli sulla distinzione tra liste e coppie puntate, che ha non poche implicazioni nella programmazione in LISP nostra attenzione sulle liste. Si noti soltanto che la funzione LISTP verifica se il suo argomento è una struttura costruita tramite `CONS`, oppure la lista vuota, quindi non distingue correttamente tra liste e coppie puntate. Ricordiamo inoltre che nel caso della coppia puntata le funzioni di selezione prendono il nome di `CAR` e `CDR`, che restituiscono il primo ed il secondo elemento della coppia puntata, rispettivamente. `CAR` e `CDR` possono essere usate in maniera del tutto analoga a `FIRST` e `REST` quando hanno gli stessi tipi di argomenti previsti per `FIRST` e `REST`. I nomi `CAR` e `CDR` derivano dalla prima implementazione del LISP in cui `CAR` corrispondeva alla sigla "Content of Address Register" e `CDR` a "Content of Decrement Register". Questi nomi sono poi rimasti nel gergo LISP, anche se il loro significato originario si è del tutto perduto.

1.3 Il nucleo applicativo del LISP

In questo capitolo definiamo un linguaggio, che denominiamo Micro-LISP applicativo e che risulta esere un sottoinsieme del LISP. Più precisamente esso corrisponde al nucleo del LISP basato sull'applicazione di funzioni come unica tecnica di programmazione, in modo analogo a quanto discusso nel Paragrafo 1.1. Si noti che, anche se il Micro-LISP non è un linguaggio utilizzato nella pratica della programmazione, qualsiasi implementazione del LISP offre le caratteristiche minimali del Micro-LISP. Innanzitutto, occorre ricordare che un programma LISP è costituito da un insieme di definizioni di funzione, e che l'esecuzione di un programma corrisponde al calcolo del valore di una espressione. Tralasciamo per il momento il meccanismo che consente di definire nuove funzioni, e consideriamo la sintassi delle espressioni

ed il meccanismo di valutazione per esse. Nella trattazione viene esemplificato il comportamento di un interprete LISP con le seguenti convenzioni. Il simbolo "`>`" denota il segnale di pronto (prompt) dell'interprete, e le linee che iniziano con tale carattere contengono le richieste di calcolo del valore di una espressione formulate dall'utente. Le linee seguenti fino al successivo carattere di pronto contengono la risposta del sistema. In Micro-LISP consideriamo cinque tipi di espressioni:

Costanti
Le costanti del LISP corrispondono agli elementi del dominio dei tipi di dati ammessi. In particolare, se restringiamo la nostra attenzione all'insieme di atomi descritto in precedenza ed alle liste, abbiamo che le costanti sono numeri, T, NIL, () e le espressioni precedute dal carattere QUOTE. Il valore di una costante corrisponde semplicemente all'oggetto da essa denotato. Ad esempio `3 'a '(a b c)` sono costanti i cui valori sono l'atomo `3`, l'atomo `a` e la lista `(a b c)`, rispettivamente. Un sistema LISP interpreta la richiesta `3` come il calcolo del valore dell'espressione `3` e la risposta è semplicemente 3.

```
>3
3
>
```

Analogamente nel caso dell'atomo

```
>'a
a
>
```

Si noti che l'uso del carattere `QUOTE` consente anche di esprimere costanti del tipo lista

```
>' (a b c)
(a b c)
>
```

Variabili
In LISP qualsiasi identificatore viene considerato una variabile se non è preceduto dal carattere QUOTE. Ad esempio, la risposta alla richiesta del valore dell'espressione a è un messaggio di errore che segnala che il valore della variabile a non è definito.

```
 a
''unbound variable''
>
```

Il valore di una variabile corrisponde al valore che le è stato attribuito nell'ambiente corrente. L'ambiente è una associazione tra variabili e valori inizialmente definita dall'utente, attraverso l'uso dell'espressione speciale `SETQ`. In particolare abbiamo che

(`SETQ` *variabile espressione*)

restituisce il valore dell'espressione che compare come secondo argomento, ma il suo effetto principale è quello di associare alla variabile denotata come primo argomento il valore stesso dell'espressione. Ad esempio

```
>(SETQ a (CONS 3 NIL))
(3)
>
```

comporta l'associazione del valore `(3)` alla variabile `a`. L'effetto della definizione del valore di `a` nell'ambiente si può semplicemente verificare chiedendo il valore di `a`; abbiamo quindi

```
>a
(3)
>
```

Pertanto, la valutazione di una variabile presuppone che sia noto l'ambiente al quale si fa riferimento, cioè l'ambiente nel quale si deve ricercare il valore per la variabile. Con `SETQ` si definisce l'ambiente globale, che viene usato nella valutazione delle variabili al livello più esterno di colloquio con il sistema (cioè quello sin qui considerato). Ovviamente l'effetto di `SETQ` su una variabile il cui valore è stato definito in precedenza comporta la sostituzione del vecchio valore con il nuovo. Osserviamo, quindi, che `SETQ` ha un ruolo analogo a quello di una istruzione di assegnazione nei linguaggi imperativi; in questo caso `SETQ` viene usata soltanto per definire i valori iniziali delle variabili, che rimangono poi invariati durante il calcolo del valore di una espressione, cioè durante l'esecuzione di un programma. In generale, il LISP consente un'uso più esteso dell'espressione `SETQ`, che ricalca quello dell'istruzione di assegnazione, e non viene qui approfondito, poiché si intendono illustrare soltanto gli aspetti applicativi del linguaggio.

Applicazioni di Funzioni Predefinite

Un'applicazione di funzione predefinita è un'espressione del tipo

(*nomefun argomenti*)

dove *nomefun* è il nome di una funzione predefinita nel linguaggio, come ad esempio `CONS`, `FIRST`, `REST`, `ATOM`, e `NULL` viste per il tipo di dato lista, e *argomenti* è costituito da un numero di espressioni pari al numero degli argomenti della funzione denotata da *nomefun*. Ad esempio

```
(CONS 'a ())
```

è una applicazione di funzione predefinita, in cui `CONS` corrisponde a *nomefun* e `'a` e `()` sono gli argomenti. Il valore di un'applicazione di funzione primitiva si ottiene calcolando il valore degli argomenti ed applicando ad essi la definizione della funzione. Ad esempio

```
>(CONS 'a ())
(a)
>
```

è il valore ottenuto applicando la definizione di `CONS` all'atomo `a` ed alla lista vuota, cioè al risultato della valutazione degli argomenti. Tutti gli esempi forniti per illustrare le definizioni delle funzioni del tipo di dato lista sono esempi di valutazione di funzioni primitive, in cui il risultato è riportato a destra del simbolo di uguale. Si noti che gli argomenti di una applicazione primitiva possono essere a loro volta applicazioni di funzioni primitive. In questo modo si possono costruire espressioni più complesse come

```
> (FIRST (CONS 4 (CONS 3 ())))
4
>
```

Il calcolo del valore di questa espressione richiede l'applicazione della definizione di `FIRST` al risultato della valutazione del suo argomento, cioè l'espressione `(CONS 4 (CONS 3 ()))`, il quale a sua volta richiede di applicare la definizione di `CONS` al valore di `4` e `(CONS 3 ())`. Il valore di `4` è 4, mentre il valore di `(CONS 3 ())` è la lista `(3)`, ottenuta applicando la definizione di `CONS` all'atomo `3` ed alla lista vuota. Quindi, il meccanismo di valutazione comporta che il calcolo dell'applicazione più esterna rimanga in sospeso fino a quando non siano stati ottenuti i valori degli argomenti

Condizionali

Un condizionale è un'espressione del tipo

`(COND (`*condizione1 espressione1*`)...(`*condizionen espressionen*`))`

in cui *condizione1*,..., *condizionen* sono espressioni il cui valore determina la selezione dell'espressione da valutare per ottenere il valore del condizionale. In particolare, il valore di un condizionale si ottiene valutando inizialmente la *condizione1*. Se il risultato è diverso da `NIL` allora il risultato del condizionale è dato dal valore dell'*espressione1*. Altrimenti si valuta la *condizione2*; se questa risulta diversa da `NIL` il risultato del condizionale è dato da *espressione2*; altrimenti si prosegue analogamente finché non si incontra una condizione il cui valore è diverso da `NIL`. Se nessuna delle condizioni produce un valore diverso da `NIL` il risultato del condizionale è `NIL`. Ad esempio, supponendo che alla variabile a sia stato assegnato il valore `(3)` tramite l'espressione `SETQ` abbiamo

```
>(COND ((ATOM a) a) ((LISTP a) (FIRST a))))
3
>
```

L'esecuzione della prima condizione in questo caso fallisce mentre la seconda ha successo ed il risultato del condizionale è dato dalla valutazione dell'espressione `(FIRST a)`. Si noti che se il valore di a non è nè un atomo nè una lista il risultato del condizionale è semplicemente `NIL`. Esiste tuttavia un modo molto semplice per specificare l'espressione da valutare nel caso in cui tutte le condizioni restituiscano

valore falso, come nel caso della clausola `default` nell'istruzione `switch` di Java. Ad esempio, per specificare un condizionale tale che se la variabile `a` non è nè un atomo nè una lista, viene restituito l'atomo `error` ad indicare una condizione di errore si può scrivere

```
(COND ((ATOM a) a) ((LISTP a) (FIRST a)) (T 'error)))
```

L'ultima condizione in questo caso è costituita dall'atomo `T`, che denota il valore di verità vero, ed è quindi sempre verificata, analogamente alla clausola `default` di un'istruzione `switch`. Il risultato di questo condizionale è l'atomo `error` se nessuna delle due condizioni precedenti è verificata. Osserviamo, infine, che qualunque espressione è una condizione ammissibile in LISP. Nei casi in cui una condizione restituisce un valore diverso da `NIL`, la condizione ha successo, mentre il valore `NIL` corrisponde al fallimento della condizione. In alternativa, il condizionale si può utilizzare nella forma IF-THEN-ELSE

`(IF` *condizione espressione-then espressione-else*`)`

che ha lo stesso significato di

`(COND (` *condizione espressione-then* `) (T` *espressione-else*`))`

Applicazioni di funzioni definite da utente

L'ultimo tipo di espressioni che consideriamo nel micro-LISP sono le funzioni definite da utente, attraverso le quali avviene la programmazione in LISP. Infatti, scrivere un programma in LISP equivale a definire un insieme di funzioni che vengono in seguito attivate dalle richieste di applicazione delle funzioni stesse. Introduciamo nel seguito il meccanismo per definire nuove funzioni da parte dell'utente e, successivamente, le modalità della loro valutazione.

Definizione

La definizione delle funzioni avviene attraverso una espressione speciale chiamata `DEFUN` la cui sintassi è:

`(DEFUN` *nomefunzione* `(`*parametri*`)` *corpofunzione*`)`

in cui *nomefunzione* è un identificatore che rappresenta il nome della funzione che si intende definire, *parametri* sono una lista di variabili che rappresentano i parametri formali della funzione, e *corpofunzione* è un'espressione che viene valutata al momento dell'esecuzione di una chiamata della funzione. Se una funzione è già definita la nuova definizione rimpiazza la precedente, anche nel caso delle funzioni predefinite. Ad esempio la definizione di fattoriale vista nel paragrafo precedente in LISP diventa

```
>(DEFUN fatt (X) (COND ((LE X 1) 1)(T (* X (fatt (- X 1))))))
fatt
>
```

dove `LE`, `*` e `-` sono degli operatori definiti sul tipo di dato dei numeri interi, per il minore uguale, la moltiplicazione e la sottrazione, rispettivamente. Il sistema risponde ad una definizione di funzione con il nome della funzione appena definita, fatt in questo caso. Si noti che l'associazione tra nomi di funzioni e la loro definizione prende il nome di ambiente funzionale

Valutazione
Il meccanismo di valutazione delle funzioni definite da utente è molto simile a quello delle funzioni primitive, ma occore tenere conto del legame dei parametri e della definizione della funzione specificata nel *corpofunzione*. In particolare, la sintassi di una applicazione di funzione definita da utente è del tutto analoga a quella delle funzioni primitive, e la distinzione avviene semplicemente in base al nome della funzione.

(*nomefun argomenti*)

Il numero delle espressioni corrispondenti agli argomenti deve essere uguale a quello dei parametri della definizione della funzione. Si noti tuttavia che in LISP vi sono anche funzioni che ammettono un numero variabile di argomenti, che però non verranno qui prese in considerazione. Per poter valutare un'applicazione di funzione definita da utente occorre eseguire i seguenti tre passi:

a) reperire i nomi dei parametri e la definizione di funzione nell'ambiente funzionale;
b) definire un ambiente locale contenente tutte le variabili che si trovano nella lista dei parametri, cui viene associato il valore ottenuto valutando l'espressione dell'argomento corrispondente;
c) valutare il corpofunzione usando l'ambiente locale appena definito.
Ad esempio, definiamo la funzione che seleziona il secondo elemento di una lista, e restituisce l'atomo errore se la lista ha meno di due elementi.

```
>(DEFUN secondo (l)
   (COND ((NULL l) 'errore)
         (NULL (REST l)) 'errore)
         (T (FIRST (REST l))))))
secondo
>
```

Per selezionare il secondo elemento di una lista è sufficiente prendere il resto della lista iniziale, e selezionare il primo elemento della lista risultante. Vediamo ora una valutazione della applicazione della funzione secondo

```
>(secondo '(a b c))
b
>
```

I tre passi che vengono eseguiti nella valutazione di secondo sono i seguenti:

a) vengono reperite nell'ambiente funzionale la lista dei parametri della funzione `(l)` ed il suo corpo

```
(COND ((NULL l) 'errore)
      (NULL (REST l)) 'errore)
      (T (FIRST (REST l)))))
```

b) viene creato l'ambiente locale in cui alla variabile `l` viene associato il valore risultante dalla valutazione dell'argomento della chiamata e cioè la lista `(a b c)`.
c) viene valutato il *corpofunzione*:

```
(COND ((NULL l) 'errore)
      (NULL (REST l)) 'errore)
      (T (FIRST (REST l)))))
```

usando l'ambiente locale in cui alla variabile l è associata la lista `(a b c)`. Si lascia al lettore la verifica del risultato.

Nel caso della valutazione di una funzione ricorsiva come `fatt` vista in precedenza, il meccanismo di creazione dgli ambienti locali consente senza alcuna difficoltà la valutazione di chiamate ricorsive. Consideriamo ad esempio

```
> (fatt 3)
6
>
```

In questo caso la valutazione avviene nel modo seguente

a) vengono reperiti nell'ambiente funzionale la lista dei parametri della funzione `(X)` ed il suo corpo

```
(COND ((LE X 1) 1) (T (* X (fatt (- X 1)))))
```

b) viene creato l'ambiente locale in cui alla variabile `X` viene associato il valore risultante dalla valutazione dell'argomento della chiamata e cioè 3.
c) viene valutato il corpo della funzione:

```
(COND ((LE X 1) 1) (T (* X (fatt (- X 1)))))
```

usando l'ambiente locale in cui la variabile `X` ha valore `3`. La valutazione della condizione `(LE X 1)` produce `NIL`, quindi si procede con la valutazione di `(* X (fatt (- X 1)))`. Nell'ambiente locale `X` vale `3`, mentre il valore di `(fatt (- X 1)))` si ottiene riapplicando il meccanismo di valutazione della applicazione di funzione definita da utente.

> *ca)* vengono reperiti nell'ambiente funzionale la lista dei parametri della funzione (X) ed il suo corpo
>
> `(COND ((LE X 1) 1) (T (* X (fatt (- X 1)))))`

cb) viene creato un nuovo ambiente locale in cui alla variabile `X` viene associato il valore risultante dalla valutazione dell'argomento della chiamata e cioè `(- X 1)`, il cui valore è `2`.
cc) viene valutato il corpo della funzione,

```
(COND ((LE X 1) 1) (T (* X (fatt (- X 1)))))
```

usando l'ambiente locale in cui alla variabile `X` ha valore `2`. La valutazione della condizione `(LE X 1)` produce `NIL`, quindi si procede con la valutazione di `(* X (fatt (- X 1)))`. Nell'ambiente locale `X` vale `2`, mentre il valore di `(fatt (- X 1))` si ottiene riapplicando il meccanismo di valutazione della applicazione di funzione definita da utente.

> *cca)* vengono reperiti nell'ambiente funzionale la lista dei parametri della funzione `(X)` ed il suo corpo `(COND ((LE X 1) 1) (T (* X (fatt (- X 1)))))`
> *ccb)* viene creato un nuovo ambiente locale in cui alla variabile X viene associato il valore risultante dalla valutazione dell'argomento della chiamata e cioè `(- X 1)`, il cui valore è `1`
> *ccc)* viene valutato il corpo della funzione, `(COND ((LE X 1) 1) (T (* X (fatt (- X 1)))))` usando l'ambiente locale in cui alla variabile `X` ha valore `1`. La valutazione della condizione `(LE X 1)` produce `T`, quindi si procede con la valutazione di `1`, che dà `1`. In questo modo si completa la valutazione degli argomenti della applicazione `TIMES` al passo cc), il cui risultato `(2*1)` è `2`. Questo a sua volta completa la valutazione degli argomenti della applicazione `TIMES` al passo c), il cui risultato `(3*2)` è `6` che viene restituito come valore dell'espressione `(fatt 3)`.

1.4 Programmazione tramite funzioni ricorsive

In questo paragrafo vengono forniti alcuni esempi di programmi in Micro-LISP. Iniziamo col definire un predicato memb che determina se un atomo appartiene ad una lista.

```
>(DEFUN memb (a l)
    (COND ((NULL l) NIL)
          ((EQ a (FIRST l)) T)
          (T (memb a (REST l)))))
memb
>(memb 4 '(3 4 5))
T
```

```
>(memb 6 '(3 4 5))
NIL
>
```

La definizione data ha due casi elementari corrispondenti alla lista vuota e ad una lista il cui primo elemento è quello cercato, ed un caso ricorsivo, che prosegue la verifica nel resto della lista. Questa definizione di `memb` è corretta per liste lineari, vediamo come occorre modificarla per liste di liste.

```
>(DEFUN membertree (a l)
    (COND ((NULL l) NIL)
          ((ATOM (FIRST l))
             (COND ((EQ a (FIRST l))T) (T (membertree a (REST l)))))
          ((membertree a (FIRST l)) T)
          (T (membertree a (REST l)))))

>(membertree '4 '(3 (4 6) 5))
T
>(membertree '6 '(3 (4 7) 5))
NIL
>
```

In questo caso la definizione prevede due casi ricorsivi; il primo effettua la ricerca nella lista eventualmente presente in prima posizione; il secondo, analogo a quello di `memb`, effettua la ricerca nel resto della lista. Il lettore può verificare i risultati degli esempi applicando la definizione di `membertree`. Vediamo ora come costruire una funzione che conta il numero di occorrenze di un atomo all'interno di una lista di liste.

```
>(DEFUN conta (a l)
    (COND ((NULL l) 0)
          ((ATOM (FIRST L))
              (PLUS (COND((EQ a (FIRST l)) 1)(T 0))
                    (conta (a (REST l)))))
          (T (PLUS (conta a (FIRST l)) (conta a (REST l))))))
conta
>(conta 4 '(3 (4 6) (5 (2 4 7))))
2
>
```

dove `PLUS` è la funzione che calcola la somma di due interi. La struttura della funzione conta prevede tre casi: la lista è vuota ed il risultato è 0, la lista ha un atomo come primo elemento e a seconda che questo sia o meno uguale all'atomo cercato occorre aggiungere 1 o 0 al conto delle occorrenze nel resto della lista, altrimenti basta sommare le occorrenze nella lista che si trova in prima posizione, a quelle del resto della lista. Si noti che nella seconda clausola del condizionale il primo argomento della funzione `PLUS` è l'espressione condizionale

```
(COND ((EQ a (FIRST l)) 1) (T 0)),
```

che restituisce 1 o 0 in base al confronto tra l'atomo cercato ed il primo elemento della lista. Passiamo ora ad una funzione che, data una lista di liste ne, restituisce una uguale in cui le occorrenze dell'atomo x, vengono sostituite dall'atomo y.

```
>(DEFUN sost (x y l)
   (COND ((NULL l) NIL)
         ((ATOM (FIRST l))
            (CONS (COND ((EQ x (FIRST l)) y)
                        (T (FIRST l))))
                  (sost x y (REST l)))
         (T (CONS (sost x y (FIRST l)) (sost x y (REST l))))))
sost
>(sost 4 8 '(3 (4 6) (5 (2 4 7))))
(3 (8 6) (5 (2 8 7))
>
```

Si noti che la funzione `sost` pur essendo del tutto analoga come struttura alla funzione `conta` restituisce un risultato di natura diversa, cioé una lista strutturalmente analoga a quella in input, con le opportune sostituzioni di atomi. Il risultato viene costruito attraverso la funzione `CONS` mettendo insieme le liste ottenute dalle chiamate ricorsive, senza modificare la lista di partenza. Questo stile di programmazione, detto applicativo, non richiede di modificare i valori delle variabili nella memoria e si basa soltanto sul calcolo del risultato dell'applicazione di funzioni ad argomenti.

1.5 Altre caratteristiche del LISP

Il LISP, in numerose versioni e dialetti proliferati grazie alla possibilità di estendere e modificare con facilità il linguaggio stesso, è senza dubbio il linguaggio funzionale più usato. È stato definito uno standard per il LISP, chiamato COMMON LISP, che ha avuto una notevole diffusione. Anche per il sottosistema del LISP riguardante la programmazione orientata ad oggetti è stato definito uno standard noto come CLOS (COMMON LISP OBJECT SYSTEM). Il LISP offre una notevole varietà di strutture dati e consente l'uso di diversi paradigmi di programmazione oltre a quello applicativo qui discusso. Innanzitutto, sono disponibili in LISP le istruzioni tipiche dei linguaggi imperativi, quali il ciclo o l'assegnazione che, se da una parte portano ad una migliore efficienza, dall'altra rischiano di compromettere, in molti casi, la chiarezza e la leggibilità del programma. Infine va ricordato che molti ambienti di programmazione per la costruzione di sistemi basati sulla conoscenza sono inseriti in ambiente LISP. Questo implica che per molti sistemi LISP è possibile utilizzare prodotti software per la rappresentazione della conoscenza sottoforma di regole di produzione o di frames.

I linguaggi funzionali si sono diffusi inizialmente negli ambienti in cui maggiormente era sentita la necessità di uno strumento adatto alla manipolazione di simboli

ed in particolare nei laboratori di Intelligenza Artificiale. La flessibilità offerta dal linguaggio ne ha inoltre consentito una rapida evoluzione. Infatti, per mezzo delle liste possono essere facilmente rappresentati i programmi scritti nel linguaggio stesso, e, quindi, scrivere agevolmente programmi che a loro volta operano su programmi come, ad esempio, interpreti e compilatori. In questo modo sono stati quindi costruiti veri e propri *ambienti di programmazione* che facilitano lo sviluppo di programmi di notevoli dimensioni e complessità.

Infatti, in un ambiente di programmazione LISP, oltre ai tradizionali interprete e compilatore per il linguaggio, sono spesso disponibili altri strumenti in grado di permettere ad esempio l'esecuzione di programmi parzialmente definiti, di sospendere l'esecuzione di un programma e riprenderla dopo che questo è stato modificato ecc.. In questo modo, l'attività di programmazione diventa altamente interattiva e si eliminano parte delle inefficienze legate al ciclo redazione del programma-compilazione-esecuzione. La presenza di un insieme di strumenti in grado di assistere il programmatore in tutte le fasi della messa a punto del programma, al pari delle caratteristiche proprie del linguaggio ha contribuito ad incentivare uno stile di programmazione noto col nome di *raffinamento incrementale*. Questo si basa, oltre che sulla scomposizione del progetto in molte funzioni, sulla possibilità di mantenere un elevato livello di astrazione, ritardando le decisioni relative alla rappresentazione dei dati e alla realizzazione delle operazioni. In tal senso la flessibilità offerta dalle liste, sfruttata in modo opportuno, consente di rendere i programmi largamente indipendenti dalla rappresentazione usata per le strutture dati. I linguaggi funzionali hanno avuto una notevole influenza sull'evoluzione dei linguaggi di programmazione. Ad esempio linguaggi imperativi spesso incorporano costrutti che consentono uno stile di programmazione funzionale (vedi ad esempio il costrutto function del PASCAL). Si pensi inoltre al concetto di ambiente di programmazione, che, sviluppatosi grazie alle caratteristiche di estendibilità del LISP, è stato poi adottato generalizzato a qualsiasi tipo di linguaggio. Un ultimo apporto è, infine, dato dalla definizione formale della semantica dei linguaggi di programmazione con il metodo detto *denotazionale*. Infatti i linguaggi funzionali, essendo basati sul concetto di funzione, sono facili da caratterizzare formalmente in termini di funzioni matematiche; su questo si basano anche tecniche per la dimostrazione della correttezza dei programmi.

Capitolo 2

PROLOG e programmazione logica

La logica è uno strumento di espressione che trae origine dall'esigenza di formalizzare il ragionamento umano. Fin dalla comparsa dei primi elaboratori è stato affrontato il problema della automatizzazione dei processi deduttivi propri della logica matematica (in particolare del calcolo dei predicati del primo ordine). Questo ha portato alla individuazione di sistemi logici caratterizzati da meccanismi deduttivi realizzabili su un elaboratore. Inizialmente i sistemi basati sulla logica erano rivolti alla dimostrazione di teoremi; successivamente questi studi hanno aperto la strada alla possibilità di usare la logica come linguaggio di programmazione. I linguaggi di programmazione basati sulla logica sono anche detti *linguaggi dichiarativi*.

Scrivere un programma in un linguaggio basato sulla logica richiede al programmatore di comunicare al sistema automatico (formalizzandole) le proprie conoscenze relativamente ad un certo problema. Ciò avviene mediante l'asserzione di *fatti* e *regole* che ne descrivano gli aspetti di interesse. L'esecuzione di un programma si ottiene ponendo un quesito al sistema. La risposta, consiste nella verifica della verità o della falsità dell'affermazione contenuta nel quesito, sulla base delle conoscenze specificate.

Un sistema di programmazione basato sulla logica è in genere costituito da un interprete incaricato di memorizzare, in una base delle conoscenze, fatti e regole asseriti dal programmatore, e dedurre, mediante l'applicazione di un principio di deduzione, altre verità a partire da quelle specificate.

La caratteristica fondamentale di un linguaggio di programmazione basato sulla logica, sta nel richiedere, per la soluzione di un problema, la specifica di cosa deve essere calcolato, piuttosto che il modo in cui il risultato deve essere ottenuto. Ciò viene solitamente indicato tramite la distinzione tra linguaggi dichiarativi (o di specifica), quali quelli basati sulla logica, e linguaggi procedurali, quali quelli imperativi e funzionali.

Per evidenziare le caratteristiche dei linguaggi basati sulla logica R. Kowalski ha definito l'equazione: Algorithm = Logic + Control. Nel caso dei linguaggi logici, il compito del programmatore è limitato alla specifica della parte logica, mentre il controllo della computazione è completamente demandato all'elaboratore. Al contrario,

nel caso dei linguaggi procedurali, anche il flusso del controllo viene specificato dal programmatore.

Linguaggi di programmazione dichiarativi sono molto utili specie in fase di progetto di programmi molto complessi, poiché permettono di procedere senza sforzo per raffinamenti successivi (passaggio da una conoscenza intuitiva ad una precisa del problema da risolvere). Raffinamenti successivi sono spesso necessari per migliorare l'efficienza del programma: al crescere del livello di dettaglio della specifica è possibile ridurre l'insieme delle possibilità che devono essere vagliate dal sistema per trovare una soluzione.

È infine importante osservare che l'uso della logica come linguaggio di programmazione rende più agevole lo sviluppo di tecniche per la dimostrazione di correttezza dei programmi.

Il PROLOG è l'esempio principale di linguaggio di programmazione basato sulla logica, correntemente in uso anche al di fuori degli ambienti accademici e di ricerca. Esso si basa sull'uso della forma a clausole che riduce il potere espressivo del linguaggio. La descrizione dei linguaggi logici fatta nel seguito fa riferimento al linguaggio PROLOG, introdotto da A. Colmerauer e R. Kowalski; esistono diverse versioni del linguaggio nel seguito si farà riferimento alla versione più frequentemente usata proposta dai ricercatori dell'Università di Edimburgo.

L'utilizzo di formalismi logici più espressivi per la programmazione logica si scontra con l'efficienza della implementazione. A questo proposito va sottolineata l'importanza di combinare un metodo di specifica ad alto livello quale quello offerto dai linguaggi logici, con metodi efficienti per la memorizzazione delle informazioni.

I linguaggi basati sulla logica comprendono anche i formalismi per la rappresentazione della conoscenza forniti dai numerosi ambienti e strumenti per la progettazione di sistemi esperti attualmente disponibili sul mercato.

2.1 Principi di programmazione logica

Il PROLOG utilizza un sottoinsieme di quella parte della logica nota come il *calcolo dei predicati del primo ordine.* Prima di presentare le caratteristiche principali del linguaggio illustriamo i principi di ragionamento. La trattazione successiva farà riferimento al modello operazionale del PROLOG, mentre in questa sezione si fornisce un'intuizione dei principi di programmazione logica dei sistemi logico-deduttivi. Per approfondimenti sulla relazione tra il PROLOG ed i sistemi logico-deduttivi si rimanda [**?**, 4].

Il meccanismo che permette di effettuare deduzioni di nuova conoscenza a partire da un insieme di regole e fatti noti è noto in logica con il nome di *modus ponens.* Il modus ponens stabilisce che dal fatto B *è vero* e dalla regola *(la verità di) B implica (la verità di) A* possiamo dedurre il nuovo fatto A *è vero.* Un modo più compatto di scrivere è quello che utilizza il simbolo $\supset$" come simbolo di implicazione. In questo modo la regola del modus ponens diviene: dalla verità di B e di e di $B \supset A$ si deduce la verità di A. Ad esempio l'applicazione della regola del modus ponens ci

permette di dedurre che *la terra si bagna* dalle due seguenti affermazioni *se piove la terra si bagna* e *piove*.

Il meccanismo di deduzione di nuova conoscenza del linguaggio PROLOG è più complesso. Infatti in PROLOG i fatti e le regole possono avere variabili e costanti. In questo caso le variabili devono essere considerate quantificate in modo universale; in particolare, se X è una variabile la regola *B(X) implica A(X)* equivale ad affermare *(per ogni valore di X la verità di) B(X) implica (la verità di) A(X)*; analogamente se c è una costante *B(c)* è un fatto che afferma la verità di B relativamente alla costante c , cioè *la costante c soddisfa B*. Se si assume la verità della regola *B(X) implica A(X)* e del fatto *B(c)* è possibile dedurre la verità di *A(c)*. La deduzione è stata effettuata osservando che

1. *B(c) implica A(c)* è un'istanza (cioè un caso particolare) della regola *B(X) implica A(X)*;

2. se *B(c) implica A(c)* e *B(c)* sono veri, allora il modus ponens permette di dedurre *A(c)*.

In generale il formato di una regola in PROLOG è del tipo *(la verità di) B, C,..., Z implica (la verità di) A*. In questo caso dalla verità dei fatti $B', C', ..., Z'$ possiamo dedurre la verità di A' se i fatti $A', B', C', ..., Z'$ sono un caso particolare dei fatti $A, B, C, ..., Z$.

Questo principio di deduzione era noto agli antichi greci. Una sua applicazione famosa è la seguente: dalla regola *Tutti gli uomini sono mortali* e dal fatto *Socrate è un uomo* si deduce il nuovo fatto *Socrate è mortale*. Infatti la regola *Tutti gli uomini sono mortali* equivale ad affermare che *la verità di 'X è un uomo' implica la verità di 'X è mortale'*; pertanto da questa regola e dalla verità di *Socrate è un uomo* possiamo dedurre *Socrate è mortale*.

L'esempio precedente in PROLOG può essere scritto come:

```
mortale(X) :- uomo(X).
uomo(socrate).
```

La deduzione che Socrate è mortale si ottiene ponendo al sistema il quesito

```
? mortale(socrate).
```

Scrivere un programma in un linguaggio di programmazione basato sulla logica richiede quindi i seguenti passi:

1. definizione del problema da risolvere mediante l'asserzione di fatti e regole ad esso relativi, in una *base delle conoscenze*;

2. interrogazione del sistema automatico che è in grado di eseguire deduzioni sulla base dei fatti e delle regole note (prova di teoremi).

Questi due aspetti verranno affrontati nei prossimi paragrafi.

2.1.1 La base delle conoscenze

Un programma PROLOG è costituito da un insieme di *clausole*, ossia di *asserzioni condizionate o incondizionate*. Una asserzione incondizionata, detta altrimenti *fatto*, esprime una relazione tra oggetti: ad esempio, la relazione *il padre di Mario è Enzo* può essere espressa dalla clausola

```
padre(enzo,mario).
```

in cui `padre` è un simbolo di predicato, mentre `enzo` e `mario` sono i suoi argomenti che rappresentano due individui della classe esseri umani.

Nel seguito i nomi dei predicati e delle costanti iniziano con minuscole. Come già osservato questa e le successive convenzioni fanno riferimento al PROLOG di Edimburgo. Analogamente avremo che *la madre di Enzo è Livia* si scrive `madre(livia,enzo)` e *Rita è bella* si scrive `bella(rita)`.

Definiamo ora un insieme di fatti che utilizzeremo negli esempi seguenti.

```
padre(enzo,mario).
padre(enzo,claudia).
padre(andrea,sara).
padre(andrea,enzo).
madre(sara,rita).
madre(livia,enzo).
madre(maria,mario).
madre(maria,claudia).
bella(rita).
bella(claudia).
```

Se si introducono i simboli di variabile, un fatto può esprimere anche relazioni più complesse. Nel seguito i nomi delle variabili iniziano con lettere maiuscole. In PROLOG per esprimere il fatto che *Enzo ama ogni essere umano* si può scrivere

```
ama(enzo,X).
```

dove `X` è un simbolo di variabile che rappresenta uno qualunque degli oggetti definiti nel programma.

Una asserzione condizionata, detta altrimenti *regola*, assume la forma

```
A :- B,C,...,D.
```

che si legge *A è vero se lo sono B, C,... , D*, dove *A, B, C,* e... *D* sono formule costituite da un solo predicato, come quelle mostrate in precedenza. *A* è detta conclusione, mentre *B, C,... , D* costituiscono complessivamente l'ipotesi della asserzione considerata. Ad esempio la relazione nonno è definita da due regole. La prima regola asserisce che *un individuo X è il nonno di un individuo Z se: X è padre di un individuo Y e Y è a sua volta padre di Z*, e si scrive nel modo seguente

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

la seconda regola afferma che *un individuo X è il nonno di un individuo Z se: X è padre di un individuo Y e Y è a sua volta madre di Z*, che si scrive

```
nonno(X,Z) :- padre(X,Y), madre(Y,Z).
```

2.1.2 Interrogazione del sistema

La forma assunta da una interrogazione del sistema è detta *goal* ed è del tipo seguente:

```
? A,B,C,...,D.
```

dove *A,B,C* ecc. sono formule contenenti un solo predicato. Il significato del goal è: dimostra, in base alle tue conoscenze, che tutte le formule *A,B,C,...* sono vere, e ritorna come risultato i valori assunti dalle variabili al termine della dimostrazione. Nel caso in cui la prova non abbia un esito positivo, si assume che il goal non sia soddisfatto ed il risultato della computazione è un opportuno messaggio al programmatore.

Un primo esempio di interrogazione utilizza solo le informazioni presenti nella base delle conoscenze. Ad esempio, alla domanda

```
? padre(mario,rita).
```

il sistema risponde *SI* perchè nella base delle conoscenze è memorizzato un tale fatto. Invece nel caso della domanda

```
? bella(maria).
```

Il sistema risponde *NO*, perchè nella sua base delle conoscenze non è memorizzato tale fatto. Si noti che in questo caso si assume che tutto quello che non è memorizzato nella base dei fatti (e che non sia deducibile utilizzando le regole) non sia vero.

Nel caso che l'interrogazione siano presenti variabili allora il sistema fornisce anche i valori delle costanti che rendono vero il goal. Ad esempio, nel quesito *sai il nome di una bella ragazza?*, che si scrive

```
? bella(X).
```

`X` è una variabile, e la risposta che si ottiene è *SI rita*. Infatti la base delle conoscenze contiene il fatto `bella(rita)`, e pertanto la costante rita soddisfa il goal. Se siamo interessati a conoscere se esistono altre soluzioni alla domanda posta è sufficiente fornire in ingresso il simbolo `';'` ed il sistema ricerca una nuova soluzione diversa da quella precedentemente fornita (in questo caso *SI claudia*).

Negli esempi precedenti le risposte alle domande poste erano basate unicamente sulle informazioni rappresentate dai fatti. Questo tipo di interrogazione è molto simile alle interrogazioni di un data base relazionale.

I casi più interessanti di interrogazioni sono quelli che coinvolgono regole perchè in questo caso si possono dedurre nuove conoscenze da quelle memorizzate nella base delle conoscenze.

Supponiamo ora di voler individuare quella persona che è nipote di Andrea ed è bella; ciò si traduce nella seguente richiesta:

```
? nonno(andrea,X), bella(X).
```

In questo caso entrambe le condizioni devono essere verificate per lo stesso valore della variabile `X`; la risposta è ancora positiva, poiché il sistema, utilizzando i fatti della base delle conoscenze e una delle regole che definiscono la relazione di nonno, trova che Rita soddisfa la richiesta.

Come ulteriore esempio consideriamo il goal: *dimmi il nome di una persona che è nonno di una bella ragazza*; questo goal si scrive nel seguente modo

```
? nonno(X,Z), bella(Z).
```

In questa domanda sono presenti due variabili (`X` e `Z`); è facile verificare che una risposta affermativa è ottenuta ponendo `X = andrea` e `Y = rita` poiché la base delle conoscenze contiene i fatti

```
padre(andrea,sara).
madre(sara,rita).
```

che permettono di dedurre che `nonno(andrea,rita)`; inoltre il fatto `bella(rita)` permette di rispondere affermativamente al quesito posto.

Fornendo in ingresso un `';'` possiamo ottenere una nuova soluzione, con `X = andrea` e `Z = claudia`.

2.1.3 Regole ricorsive

In un programma PROLOG è possibile definire anche regole di tipo ricorsivo. Consideriamo, ad esempio il predicato `discendente(X,Y)` che è vero se `X` è discendente di `Y`. Esso è specificato dal seguente programma:

```
discendente(X,Y):-figlio(X,Y). % 1
discendente(X,Y):-figlio(Z,Y),discendente(X,Z). % 2
figlio(X,Y):-padre(Y,X). % 3
figlio(X,Y):-madre(Y,X). % 4
```

La prima regola [1] stabilisce che `X` è discendente di `Y` se `X` è figlio di `Y`; la seconda regola ricorsiva afferma che `X` è discendente di `Y` se esiste un individuo `Z` tale che `Z` è figlio di `Y` e `X` è discendente di `Z`.

Poniamo ora la domanda:

[1] % 1 è un commento del programma, usato nel testo per distinguere le regole

```
? discendente(rita,andrea).
```

In questo caso il sistema verifica che effettivamente Rita è discendente di Andrea. Si ha infatti che Andrea è padre di Sara, e Sara è madre di Rita, e la relazione `discendente(rita,andrea)` viene provata utilizzando la regola 2 una volta. Si osservi che, in generale, la regola 2 può essere un applicata un numero arbitrario di volte nella ricerca della soluzione, dipendente dalla base delle conoscenze e dal quesito posto.

Supponiamo ora di voler conoscere le coppie di persone che appartengono alla stessa generazione. In questo caso abbiamo le seguenti due regole:

```
stessa_generazione (X,X).
stessa_generazione (X,Y):-
   genitore(Z,X),stessa_generazione(Z,W),genitore(W,Y).
```

La prima regola afferma che un individuo appartiene alla sua generazione; la seconda regola ricorsiva afferma che se due individui `Z` e `W` (non necessariamente distinti fra loro) sono della stessa generazione e `Z` è un genitore di `X` e `W` è un genitore di `Y`, allora anche `X` e `Y` sono della stessa generazione.

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

Ad esempio, alla domanda

```
? stessa_generazione(rita,Y).
```

il sistema risponde *SI rita* perchè quando la variabile Y è posta pari a Rita la prima regola è verificata (infatti Rita appartiene alla sua stessa generazione). Altre risposte all'interrogazione sono date da fornendo il nome di una persona che è della stessa generazione di Rita (se utilizziamo la nostra base di conoscenze Mario o Claudia).

2.1.4 L'esecuzione dei programmi PROLOG

Le risposte fornite agli esempi visti finora hanno permesso di definire in modo intuitivo le modalità di ragionamento del linguaggio utilizzato. Nel seguito si analizza questo aspetto in maggior dettaglio.

Una caratteristica importante nella scrittura di programmi PROLOG riguarda l'ordine con cui si scrivono le regole e i predicati nel corpo di una regola.

Infatti l'esecuzione di un programma PROLOG consiste nel cercare di soddisfare il goal esaminando le regole *nell'ordine con cui esse sono scritte a partire dalla prima regola* e i predicati all'interno di ciascuna regola da *sinistra a destra*. Questa modalità di scelta fa sì che modificando l'ordine delle regole e dei predicati all'interno delle regole di un dato programma possiamo ottenere un nuovo programma formato dallo stesso insieme di fatti e regole, che ha diverso tempo di esecuzione o che, addirittura, non termina. Illustriamo questo concetto riprendendo l'esempio in cui si determinano i discendenti di una data persona. Supponiamo di avere il seguente programma ottenuto scambiando fra loro le regole 3 e 4.

```
discendente(X,Y):- figlio(X,Y).
discendente(X,Y) :- figlio(Z,Y),discendente(X,Z).
figlio(X,Y) :- madre(Y,X).
figlio(X,Y):- padre(Y,X).
```

Poniamo ora la domanda:

```
? discendente(andrea,rita).
```

Il programma scritto cerca di soddisfare il goal `?discendente(andrea,rita)` esaminando preliminarmente la regola 1. Essa stabilisce che Andrea è discendente di rita se è vero `figlio(andrea, rita)`; a questo scopo si possono utilizzare le regole 3 e 4. Poichè il sistema esamina le regole nell'ordine con cui queste sono scritte, viene considerata prima la regola 3 e ricercato nella base delle conoscenze il fatto `padre(rita,andrea)`. Solo dopo che è stato verificato che questo fatto non esiste, si prova a soddisfare il predicato `figlio(andrea,rita)` utilizzando la regola 4.

È evidente che l'ordine di scrittura delle regole influenza l'efficenza del programma. Infatti se avessimo scambiato fra loro le due regole 3 e 4 non sarebbe stato necessario esaminare la base delle conoscenze alla ricerca del fatto `padre(rita,andrea)` ottenendo un programma più efficiente (cioè un programma che esamina un minor numero di fatti e applica un numero minore di regole).

L'esempio illustra anche come l'efficienza di un programma dipende dal goal che si deve soddisfare.

L'esempio seguente mostra invece come l'ordine con cui si scrivono i predicati all'interno di una regola può influenzare la terminazione di un programma. Consideriamo ancora il programma che stabilisce la relazione di discendente e supponiamo di riscrivere il programma nel seguente modo:

```
discendente(X,Y):- discendente(X,Z),figlio(Z,Y). %1'
discendente(X,Y):- figlio(X,Y).
figlio(X,Y):- padre(Y,X).
figlio(X,Y):- madre(Y,X).
```

Se ora cerchiamo di soddisfare il goal `?discendente(andrea,luigi)` si applica la regola 1' che richiede di ricercare un individuo Z che verifica i seguenti due fatti

```
discendente(andrea,Z).
figlio(Z,luigi).
```

Per come è stata scritta la regola 1' il sistema cerca prima di soddisfare il fatto `discendente(andrea,Z)` e solo successivamente il fatto `figlio(Z,luigi)`; per soddisfare il fatto `discendente(andrea,Z)` si utilizza ancora la regola 1' che afferma Andrea è discendente di `Z` se esiste un individuo `Z'` che soddisfa i seguenti predicati

```
discendente(andrea,Z'). % 3.
figlio(Z',Z). %4.
```

L'esecuzione del programma prosegue in modo analogo: per verificare la verità di `discendente(andrea,Z')` il sistema ricerca un individuo `Z''` che soddisfa opportune proprietà e poi un individuo `Z'''` e così via. è chiaro che in questo modo il programma non termina e genera una sequenza infinita di predicati da soddisfare.

2.1.5 Modello operazionale

La caratterizzazione precisa dell'esecuzione di un programma PROLOG viene fornita attraverso il *modello operazionale* del linguaggio. Questa caratterizzazione viene indicata normalmente come *semantica procedurale*. Per una caratterizzazione della esecuzione dei programmi PROLOG in termini di un processo logico deduttivo, i.e *semantica dichiarativa* si rimanda a [5, 4].

Il modello operazionale del PROLOG è costituito da un *interprete astratto* che dato un programma ed una interrogazione, restituisce la risposta `SI/NO`. Quando la risposta è positiva vengono restituiti anche i valori delle variabili che rendono vera l'asserzione corrispondente alla interrogazione. Un ruolo fondamentale nell'interprete viene svolto dall'algoritmo di *unificazione*, che restituisce la sostituzione più generale che rende uguali due espressioni. Infine, il procedimento di calcolo della risposta viene illustrato come il processo di costruzione dell'*albero di ricerca delle soluzioni.*

L'interprete astratto

L'interprete astratto descrive il procedimento utilizzato per trovare la risposta ad un goal. Più precisamente, dato un programma ed un goal l'interprete astratto restituisce un'istanza del goal che è conseguenza logica del programma, altrimenti `NO`.

Input: un goal G ed un programma P
Output: un'istanza di G, conseguenza logica di P se esiste,
altrimenti NO
begin
  $R := G$; R risolvente
  finito := false;
  while not $R = \emptyset$ and not finito **do**
    begin
      scegli un goal A dal risolvente
      scegli una clausola `A' :- B1,...,Bn` (ridenominata)
        tale che $\theta = unifica(A, A')$
      if scelte esaurite
        then finito:=true;
        else begin
          sostituisci A con $B1, ..., Bn$ in R
          applica θ ad R e G;
        end
    end

if $R = \emptyset$
then ritorna G
else NO
end

L'inteprete astratto costruisce la risposta al programma utilizzando una struttura, denominata *risolvente*, che rappresenta la porzione del goal che deve ancora essere dimostrata (congiunzione di goal). Ad ogni passo viene selezionata una regola applicabile per dimostrate una porzione del goal. Se la regola applicata è un fatto, la porzione del goal è dimostrata direttamente, altrimenti la sua dimostrazione viene ricondotta alla dimostrazioine della parte destra della regola, che viene quindi inserita nel risolvente.

La selezione della regola da applicare si basa sul processo di unificazione che restituisce, se esiste, una sostituzione che consente di rendere la testa della clausola identica alla porzione di goal da dimostrare. Tale sostituzione rappresenta un vincolo sul valore delle variabili che compaiono nel goal e nella regola e deve, pertanto, essere applicata ad entrambi. L'algoritmo di unificazione viene descritto in dettaglio nel prossimo paragrafo.

Si noti che prima dell'unificazione le variabili nelle regole vengono ridenominate in modo da renderle diverse da quelle che compaiono nel goal. Questa operazione è necessaria in quanto l'unificazione considera uguali le variabili con lo stesso nome, mentre non vi è alcuna relazione tra variabili che compaiono nel goal e variabili usate nelle clausole che abbiano lo stesso nome.

L'algoritmo è formulato in modo non deterministico e termina con fallimento solo se nessuna delle scelte possibili per il programma consente di terminare il procedimento con il risolvente vuoto. In pratica il PROLOG utilizza un pila per la gestione del risolvente ed esamina le clausole nell'ordine in cui sono scritte nel programma. Questa scelta può in alcuni casi portare alla non terminazione del programma, come abbiamo visto per il calcolo dei discendenti.

Unificazione (senza simboli di funzione)

Per completare il modello di computazione occorre risolvere il problema di verificare quali regole sono applicabili per la dimostrazione di un goal. Il metodo che ci permette di compiere questa operazione si chiama *unificazione*, in quanto realizza il confronto tra il goal che si vuole dimostrare e la testa di una clausola e determina la sostituzione delle variabili che permette di rendere uguali le due espressioni. L'operazione di unificazione viene usata in molte applicazioni. Per una trattazine approfondita si rimanda ai testi specializzati. Per precisare meglio le modalità di questa operazione introduciamo alcune definizioni.

Una *sostituzione* è una funzione dall'insieme delle variabili VAR all'insieme dei termini TERM, (per il momento costituito solo da variabili e costanti) cioè $\sigma : Var \mapsto Term$.

Dato un termine t, $t\sigma$ è definito ricorsivamente come segue:

- se c è un simbolo di costante, $c\sigma = c$;

- se x è un simbolo di variabile, $x\sigma = \sigma(x)$;

La sostituzione σ di un termine t al posto di un simbolo di variabile x è indicata da $x = t$ (oppure x/t).

Le definizioni seguenti fanno genericamente riferimento ad espressioni che, nel nostro caso, si intendono costituite soltanto da simboli di costante e di variabile, ma sono valide anche nel caso del linguaggio con i simboli di funzione, che viene considerato successivamente.

Un'espressione s si dice *più generale* di un'espressione t se t è istanza di s, ma non viceversa.

Esempio: $p(a, X)$ è più generale $p(a, b)$.

Si dice *unificatore* di due espressioni la sostituzione che applicata alle due espressioni di partenza restituisce due espressioni uguali.

Esempio: $\{X = b\}$ è un unificatore di $p(a, X)$e $p(a, b)$, infatti $p(a, X)\{X = b\} = p(a, b)$.

Si dice *unificatore più generale* di due espressioni l'unificatore che applicato ad esse produce l'istanza più generale delle due espressioni. Tale unificatore è unico a meno di ridenominazione delle variabili e viene detto *mgu* (most general unifier).

Esempio: $\{X = b, Y = b, Z = a\}$ e $\{X = Y, Z = a\}$ sono entrambi unificatori di $p(a, X)$e $p(Z, Y)$, ma $\{X = Y, Z = a\}$ è più generale di $\{X = b, Y = b, Z = a\}$.

Vediamo ora l'algoritmo di unificazione semplificato, cioè relativo al caso in cui il linguaggio non contiene simboli di funzione, quindi gli argomenti dei predicati possono essere soltanto costanti e variabili. Il confronto tra due espressioni $p(t_1, \ldots, t_n)$ e $p(s_1, \ldots, s_n)$ richiede di analizzare gli argomenti delle due espressioni nelle posizioni corrispondenti. L'algoritmo ha quindi in ingresso una lista di coppie $(< t_1, s_1 > \ldots < t_n, s_n >)$ e produce in uscita l'unificatore delle due espressioni, se esiste. L'algoritmo analizza in sequenza le coppie della lista e prevede 4 casi:

1. $t_i = s_i$: in questo caso il confronto ha successo e si prosegue passando alla coppia successiva.

2. t_i è una variabile: in questo caso $t_i = s_i$ viene inserita nell'unificatore e tutte le occorrenze di t_i nelle coppie ancora da analizzare vengono sostituite con s_i.

3. s_i è una variabile: simmetrico del precedente.

4. $t_i \neq s_i$ con t_i, s_i entrambe costanti: in questo caso le due espressioni non sono unificabili.

L'algoritmo è il seguente:

Input: C un insieme di coppie $< t_i, s_i >$ con t_i, s_i costanti o variabili
Output: unificatore più generale θ, se esiste, altrimenti false
begin
  $\theta := \{\}$;
  successo := true;
  while not empty(C) and successo **do**
  begin
    scegli $< t_i, s_i >$ in C;
    if $t_i = s_i$ **then** C:=C/$\{< t_i, s_i >\}$
      else if var(t_i)
        then begin
          $\theta :=$ subst$(\theta, t_i, s_i) \cup \{t_i = s_i\}$;
          C:=subst(rest(C)$, t_i, s_i$)
          end
        else if var(s_i)
          then begin
            $\theta :=$ subst$(\theta, s_i, t_i) \cup \{s_i = t_i\}$;
            C:=subst(rest(C)$, s_i, t_i$)
            end
          else successo := false
  end;
  if not successo **then**
    output false
  else
    output true,θ
end

subst(l, v, e) ha in ingresso una sostituzione, una variabile e una espressione e restituisce un insieme di coppie in cui le occorrenze della variabile v sono sostituite dall'espressione e.

L'albero di ricerca

Un ulteriore modo di rappresentare il procedimento descritto dall'interprete astratto si basa sulla costruzione di un albero di ricerca.

Il procedimento di dimostrazione di un goal G per un programma P si può descrivere come la costruzione di un albero ottenuto, a partire dal goal iniziale, generando un ramo per ogni applicazione di una regola. Ogni nodo contiene un risolvente.

La costruzione dell'albero si effettua nel modo seguente:

- la radice è il goal iniziale;

- ogni nodo ha tanti successori quante sono le clausole la cui testa unifica con uno dei goal presenti nel nodo. Ogni successore ha un risolvente ottenuto da quello del padre sostituendo al goal selezionato il corpo della clausola corrispondente ed applicando ad esso l'unificatore.

Se il risolvente è vuoto il nodo è un *nodo di successo*. Un nodo privo di successori, che non sia di successo, è un *nodo di fallimento*.

Ogni nodo di successo rappresenta una soluzione. Se l'albero non può essere espanso e non contiene nodi di successo il goal fallisce.

L'analisi del metodo di ricerca della soluzione ad una interrogazione mette in evidenza le due forme di non determinismo presenti nell'interprete astratto:

- nella scelta del goal da valutare;
- nella scelta della clausola da utilizzare.

Nell'interprete PROLOG la scelta del goal da dimostrare viene realizzata esaminando i goal da sinistra a destra e le clausole vengono usate nell'ordine in cui sono scritte. La scelta del goal da dimostrare determina la struttura dell'albero di ricerca. L'ordine in cui vengono esaminate le clausole determina l'ordine dei successori di un nodo. Più precisamente, il risolvente viene gestito tramite una pila nella quale i goal vengono inseriti nell'ordine inverso a quello in cui sono scritti, in modo che l'ultimo inserito sia il primo incontrato nel corpo di una clausola. La politica di gestione del risolvente comporta che la costruzione dell'albero avvenga con una strategia in profondità. La creazione dei successori avviene da sinistra a destra, con il successore sinistro corrispondente alla prima clausola del programma. Queste scelte rendono effciente, ma incompleto il meccanismo di computazione del PROLOG.

2.1.6 Esercizi

Genealogia Definire la relazione `fratello` e quindi la relazione `cugino`.

Analisi di un programma Dato il seguente frammento di codice

```
p(X,X).
p(X,Y):- figlio(X,Z),p(Z,W),figlio(Y,W).
```

verificare cosa calcola il predicato `p(X,Y)`

Grafi Definire un programma per la rappresentazione di un grafo tramite la relazione che descrive gli archi. Definire una relazione `cammino` che ha come argomenti 2 nodi del grafo e verifica se esiste un cammino tra i due nodi.

Albero di ricerca Data la seguente definizione di `discendente2`

```
discendente2(X,Y):-figlio(Z,Y),discendente2(X,Z). % 2
discendente2(X,Y):-figlio(X,Y). % 1
```

Costruire l'albero di ricerca per `?- discendente(enzo,andrea).` e verificare le differenze con `?- discendente2(enzo,andrea).`

2.2 Termini e strutture dati

In questa sezione vengono introdotti nel linguaggio i termini attraverso i quali si possono costruire strutture di dati.

2.2.1 Termini

Innanzitutto occorre estendere insieme dei *termini* TERM (in precedenza costituito solo da variabili e costanti). TERM è l'insieme induttivo definito come segue:

1. Ogni simbolo di costante è un termine;
2. Ogni simbolo di variabile è un termine;
3. Se $t_1 \dots t_n$ sono termini e f è un simbolo di funzione n-aria, $f(t_1, \dots, t_n)$ è un termine (detto *termine funzionale*).

Seguendo la definizione di termine si ha che x, c, $f(x, y + c)$ sono termini.
I termini possono essere usati per definire delle strutture dati in PROLOG, come illustrato nel segiente programma per gestire l'orario:

```
corso(ia,orario(ma,10,12),docente(nardi,d),aula(spv,1)).
corso(ia,orario(me,10,12),docente(nardi,d),aula(spv,1)).
corso(ia,orario(ve,10,12),docente(nardi,d),aula(spv,1)).
...
insegna(Ins,Corso) :- corso(Corso,Orario,Ins,Aula).
durata(Corso,Lung) :-
    corso(Corso,orario(Giorno,Inizio,Fine),Ins,Aula),
    plus(Inizio,Lung,Fine).
haLezione(Ins,Giorno) :-
    corso(Cors,orario(Giorno,Inizio,Fine),Ins,Aula).
occupata(Aula,Giorno,Ora) :-
    corso(Corso,orario(Giorno,Inizio,Fine),Ins,Aula),
    Inizio =< Ora, Ora =< Fine.
```

2.2.2 Liste

In PROLOG si possono definire liste di elementi usando il linguaggio dei termini. La definizione di liste utilizza con una diversa sintassi le stesse operazioni fondamentali del LISP. In particolare, una lista viene rappresentata fra parentesi quadre e il simbolo `|` viene usato per separare il primo elemento di una lista dal resto. Pertanto

- `[a,b,c,d]` rappresenta una lista di quattro elementi;
- `[a | X]` rappresenta una lista il cui primo carattere è 'a' e il resto della lista è denotato dalla variabile `X`;
- `[Y | X]` rappresenta una lista il cui primo carattere è denotato dalla variabile `Y` e il resto della lista è denotato dalla variabile `X`;

- Si noti che la notazione `[X | Xs]` corrisponde al termine `cons(X,Xs)`,

Per esempio, il predicato `appartiene(Elemento,Lista)` che afferma che Elemento appartiene a Lista può essere definito nel seguente modo

```
appartiene(X,[X|Xs]).
appartiene(X,[Y|Ys]) :- appartiene(X,Ys).
```

La prima regola stabilisce che `X` appartiene alla lista `[X,Xs]` cioè la lista il cui primo elemento è proprio `X`; la seconda stabilisce che X appartiene alla lista `[Y|Ys]` se quando `X` è diverso da `Y`, risulta che `X` appartiene alla lista ottenuta da `[Y|Ys]` togliendo il primo elemento.

2.2.3 Modello computazionale per il linguaggio con i termini

Il modello computazionale per il linguaggio con i termini si ottiene semplicemente definendo l'implicazione per il linguaggio esteso.

Sostituzioni nel linguaggio con i termini

Occorre innanzitutto estendere la definizione di sostituzione. Una *sostituzione* è una funzione dall'insieme delle variabili VAR all'insieme dei termini TERM, cioè $\sigma : Var \mapsto Term$.

Dato un termine t, $t\sigma$ è definito ricorsivamente come segue:

- se c è un simbolo di costante, $c\sigma = c$;
- se x è un simbolo di variabile, $x\sigma = \sigma(x)$;
- se f è un simbolo di funzione di arità n, allora $f(t_1, \ldots, t_n)\sigma = f(t_1\sigma, \ldots, t_n\sigma)$.

Se t è un termine e σ è una sostituzione, allora $t\sigma$ è un termine.

Esempio: $p(f(X,Y), a, g(b,W))$ unifica con $p(Z, X, g(b,Y))$.

Esempio: $p(f(X,Y), a, g(b,W))$ non unifica con $p(Z, f(a), g(b,Y))$.

Esempio: $p(f(X,Y), a, g(b,W))$ non unifica con $p(X, a, g(b,Y))$.
Di nuovo interessa trovare l'unificatore più generale.

Unificazione (rivista)

A questo punto è possibile definire l'unificazione di espressioni costituite da termini con simboli di funzione. L'elenco dei casi da trattare è il seguente:

1. $t_i = s_i$ due variabili o due costanti uguali: in questo caso il confronto ha successo e si prosegue passando alla coppia successiva.

2. t_i è una variabile: se t_i occorre in s_i l'unificazione fallisce, altrimenti $t_i = s_i$ viene inserita nell'unificatore e tutte le occorrenze di t_i nelle coppie ancora da analizzare vengono sostituite con s_i.

3. s_i è una variabile: simmetrico del precedente.

4. sia t_i $f(tt_1, \ldots, tt_n)$ ed s_i $g(ss_1, \ldots, ss_m)$ se $\neg(f = g) \vee \neg(n = m)$ allora l'unificazione fallisce, altrimenti le coppie $< tt_1, ss_1 >, \ldots < tt_n, ss_n >$ devono essere unificate.

L'algoritmo risulta quindi:

Input: C un insieme di coppie $< t_1, s_2 >$ con t_i, s_i termini
Output: unificatore più generale θ, se esiste, altrimenti false

```
begin
  θ := {};
  successo := true;
  while not empty(C) and successo do
  begin
    scegli < t_i, s_i > in C;
    if t_i = s_i then C:=C/{< t_i, s_i >}
      else if var(t_i)
        then if occorre(t_i, s_i)
            then successo:=false;
            else begin
              θ := subst(θ, t_i, s_i) ∪ {t_i = s_i};
              C:=subst(rest(C), t_i, s_i)
              end
        else if var(s_i)
          then if occorre(s_i, t_i)
            then successo:=false;
            else begin
                θ := subst(θ, s_i, t_i) ∪ {s_i = t_i};
                C:=subst(rest(C), s_i, t_i)
                end
          else if t_i = f(tt_1, ..., tt_n) and
                s_i = g(ss_1, ..., ss_m) and
                f = g ∧ n = m
                then C:= C ∪{< tt_1, ss_1 >, ... < tt_n, ss_n >}
                else successo := false
  end;
  if not successo then output false
    else output true, θ
end
```

Esercizi

Unificazione Verificare se le seguenti coppie di espressioni unificano. In caso positivo specificare l'unificatore altrimenti indicare altrimenti indicare una modifica del secondo termine che lo renda unificabile con il primo. X, Y, Z sono variabili e a, b sono costanti.

(1u) $f(cons(car(X), cdr(Y)), Z, X)$ e $f(Z, Z, cons(car(X), cdr(a)))$

(2u) $f(g(x, a), g(b, a))$ e $f(y, y)$

(3u) $P(g(x, a), f(b, a))$ e $P(g(f(b, y), y), f(z, y))$

(4u) `P([X|[a,b]])` e `P([a|[a|[Xs]]])`

Albero di ricerca per appartiene Costruire l'albero di ricerca per `appartiene(e,[a,b,c])`, definito nel paragrafo 2.2.2.

Albero di ricerca per reverse Costruire l'albero di ricerca per `reverse(X,[a,b,c])`, definito nel paragrafo 2.3.3.

2.3 Programmazione in PROLOG

2.3.1 Definizione dei numeri naturali

In PROLOG i numeri naturali si possono definire usando il linguaggio dei termini e la definizione induttiva di numero naturale basata sulla funzione *successore*. `natural_number(X)` è vero se X è ottenibile da 0 applicando un certo numero di volte l'operatore di successione `s()`.

```
natural_number(0).
natural_number(s(X)) :- natural_number(X).
```

Esempi

plus1 Definizione del predicato `plus1(X,Y,Z)`, vero quando Z è la somma di X ed Y.

```
plus1(0,X,X) :- natural_number(X).
plus1(s(X),Y,s(Z)):- plus1(X,Y,Z).
```

lesseq1 Definizione del predicato lesseq1(X,Y), vero quando X è minore di Y.

```
lesseq1(0,X) :- natural_number(X).
lesseq1(s(X),s(Y)) :- lesseq1(X,Y).
```

times1 Predicato `times1(X,Y,Z)`, vero quando Z rappresenta il prodotto di X ed Y.

```
times1(0,_X,0).
times1(s(X),Y,Z) :- times1(X,Y,XY), plus1(XY,Y,Z).
```

Esercizi

exp1 Definire il predicato `exp(X,Y,Z)`, vero quando Z rappresenta X^Y.

fatt Definire il predicato `fatt(X,Y)`, vero quando Y è il fattoriale di X.

minimo e massimo Definire i predicati `minimo(X,Y,Z)` e `massimo(X,Y,Z)`, veri quando Z è rispettivamente il minimo ed il massimo tra X ed Y.

2.3.2 Operatori per dati di tipo numerico: il predicato `is`

La definizione dei numeri tramite la funzione successore risulta evidentemente inefficiente per le esigenze di un linguaggio di programmazione. Il PROLOG mette a disposizione alcuni predicati di sistema.

`A is B` è un predicato di sistema, vero quando la variabile A è uguale alla *valutazione* dell'espressione B. La valutazione di B viene fatta utilizzando le funzioni di sistema. Per mezzo del predicato `is` si possono utilizzare gli operatori standard dell'aritmetica: +,*,-,/. I predicati definiti mediante `is` non sono invertibili. Ad esempio, ponendo all'interprete la richiesta
`?5 is X+Y.`
non si ottengono le possibili assegnazioni ad X e Y che rendono vero il predicato, mentre
`?X is 3+4` ritorna `X=7`.

Esempi

Fattoriale

```
factorial(0,1).
factorial(X,Y):-
        X1 is X-1,
        factorial (X1, Y1),
        Y is X*Y1.
```

Valutazione di espressioni numeriche `evalExpression(X), Z` è vero se Z è il risultato della valutazione dell'espressione X. Una espressione può è definita induttivamente come segue:

- un valore numerico è un'espressione

- `plus(X,Y)`, `minus(X,Y)`, `mult(X,Y)`, `frac(X,Y)`, con sono espressioni che denotano rispettivamente somma, sottrazione, prodotto e divisione sono espressioni se lo sono X ed Y.

```
evalExpression(plus(A,B), N):- evalExpression(A, N1),
        evalExpression(B, N2), N is N1+N2.
evalExpression(minus(A,B), N):- evalExpression(A, N1),
        evalExpression(B, N2), N is N1-N2.
evalExpression(mult(A,B), N):- evalExpression(A, N1),
        evalExpression(B, N2), N is N1*N2.
evalExpression(frac(A,B), N):-  evalExpression(A, N1),
        evalExpression(B, N2), N is N1 // N2.
evalExpression(X, X).
```

Esercizi

Valutazione di un polinomio Scrivere un predicato `evalPolynomial(Y,X,L)`, Y è il risultato della valutazione in X del polinomio i cui coefficienti sono contenuti nella lista L. Ad esempio `evalPolynomial(Y,2,[1,-1,3])` deve rispondere $y = 1 \cdot 2^0 - 1 \cdot 2^1 + 3 \cdot 2^2 = 3$.

sumElements Scrivere un predicato `sumElements(N,L)`, vero se l'intero N è pari alla somma degli elementi della lista di interi L.

2.3.3 Primitive per la manipolazione di liste

lunghezza `lunghezza(X,N)` è vero quando X è una lista di lunghezza N.

```
lunghezza([],0).
lunghezza([_X|Xs],s(N)) :- lunghezza(Xs,N).
```

member `member1(X,Y)`: l'elemento X è contenuto nella lista Y [2]. [3]

```
member1(X,[X|_Xs]).
member1(X,[Y|Ys]):- X\==Y, member1(X,Ys).
```

append `append1(X,Y,Z)` è vero quando Z è la lista ottenuta dalla concatenazione di X e Y.

```
append1([],Ys,Ys).
append1([X|Xs],Ys,[X|Zs]) :- append1(Xs,Ys,Zs).
```

[2] In questa e nelle definizioni successive viene usato un nome con 1 alla fine quando il predicato è predefinito in PROLOG.

[3] Il predicato `X \==Y` è vero quando X è diverso da Y.

prefisso `prefisso(X,Y)` è vero quando X è il prefisso di Y.

```
prefisso([],_Ys).
prefisso([X|Xs],[X|Ys]) :- prefisso(Xs,Ys).
```

delete `delete1(X,Y,Z)` è vero quando Z è la lista ottenuta da X eliminando tutte le occorrenze dell'elemento X.

```
delete1([X|Xs],X,Ys) :- delete1(Xs,X,Ys).
delete1([X|Xs],Z,[X|Ys]) :- X \== Z, delete1(Xs,Z,Ys).
delete1([],_X,[]).
```

Costruzione di programmi PROLOG Per facilitare la scrittura di programmi PROLOG si può fare riferimento alla cosiddetta lettura *procedurale*, in base alla quale la regola

$$A \leftarrow B_1, B_2, \cdots B_n$$

A è una procedura che comporta l'attivazione in sequenza delle procedure B_i.

- Il problema va espresso in modo ricorsivo.
- La selezione dei casi viene affidata all'unificazione.

reverse Definire un predicato `reverse1(X,Y)` è vero quando la lista X è ottenuta dall'inversione di Y.

Soluzione La soluzione di questo esercizio viene data assieme ad un complemento sulla costruzione ricorsiva e logica di programmi PROLOG.

Ad esempio si può formalizzare la funzione `reverse` ricorsivamente, come segue:

$$reverse(L) = \begin{cases} [] & iff\ L = [] \\ [reverse(Xs)|front(L)] & iff\ L = [X|Xs] \end{cases}$$

Il corrispondente programma PROLOG sarà:

```
reverse([], []).
reverse([X|Xs, Y):-
        reverse(Xs, Rs),
        append(Rs, [X], Y).
```

Ordinamento

```
sort1(Xs,Ys) :- permutation(Xs,Ys), ordered(Ys).

permutation(Xs,[Z|Zs]) :- select(Z,Xs,Ys),
                          permutation(Ys,Zs).
```

```
permutation([],[]).

ordered([]).
ordered([_X]).
ordered([X,Y|Ys]) :- X =< Y, ordered([Y|Ys]).

select(X,[X|Xs],Xs).
select(X,[Y|Ys],[Y,Zs]):-select(X,Ys,Zs).
```

Ordinamento per selezione Una lista è ordinata se:

- è vuota.
- ha almeno un elemento e il risultato si ottiene inserendo il primo elemento *in ordine* in una lista ordinata.

```
sort([],[]).
sort([X|Xs], Ys):-sort(Xs, Zs), insert(X, Zs, Ys).
insert(X, [], [X]).
insert(X, [Y|Ys], [Y|Zs]):- X>Y, insert(X, Ys, Zs).
insert(X, [Y|Ys], [X,Y|Zs]):- X=<Y.
```

Ordinamento per scambio

```
sort1(Xs,Xs) :- ordered(Xs).
sort1(Xs,Ys) :- append1(As,[X,Y|Bs],Xs), X >  Y,
                append1(As,[Y,X|Bs],Xs1),
                sort1(Xs1,Ys).
```

Liste di liste `memberlist(X,Y)` X è un elemento della lista di liste L.
`memberlist(X,Y)` è vero se è verificata una delle seguenti condizioni:
- X è il primo elemento di L
- X non è il primo elemento di L , il primo elemento di L è una lista ed X è membro del primo elemento di L.
- X è membro del resto di L.

```
memberlist(X,[X|_Xs]).
memberlist(X,[Y|_Ys]) :- memberlist(X,Y).
memberlist(X,[_Y|Ys]) :- memberlist(X,Ys).
```

Esercizi

suffisso `suffisso(X,Y)`: la lista X è suffisso della lista Y.

sottoinsieme `sottoinsieme(X,Y)` vero quando tutti gli elementi della lista X appartengono alla lista Y.

intersezione `intersezione(X,Y,Z)`: Z è la lista che contiene tutti gli elementi comuni ad X ed Y.

notMember Si definisca il predicato `notMember(X,L)`, vero se `X` non è membro della lista `L`.

countListElems `countListElems(X,Z)`, vero se Z è il numero di elementi contenuti nella lista di liste X

Percorso Si consideri un grafo, rappresentato mediante una serie di predicati `connected(X,Y).`, veri se il nodo `X` ed `Y` sono connessi. Si realizzi un predicato `verificaPercorso(X,Y,L)` vero se `L` è la lista che contiene un percorso nel grafo che connette `X` ad `Y`.

2.3.4 Alberi binari

`binary_tree(X)` è vero se X è un albero, ovvero se una delle seguenti condizioni vale:
- X è l'albero vuoto.
- X è una struttura composta da una informazione e due sotto-strutture, anch'esse alberi binari.

```
binary_tree(void).
binary_tree(tree(_Element,Left,Right)) :-
        binary_tree(Left), binary_tree(Right).
```

Isomorfismo `isotree(X,Y)` è vero se X ed Y sono isomorfi, ovvero se è vera una delle seguenti:
- X ed Y sono entrambi l'albero vuoto
- X ed Y hanno lo stesso elemento come radice ed i sottoalberi sinistri e destri sono isomorfi

```
isotree(void,void).
isotree(tree(X,Left1,Right1),tree(X,Left2,Right2)) :-
        isotree(Left1,Left2), isotree(Right1,Right2).
isotree(tree(X,Left1,Right1),tree(X,Left2,Right2)) :-
        isotree(Left1,Right2), isotree(Right1,Left2).
```

Visita in preordine `preorder (X,Y)`: Y è la lista che contiene le etichette dei nodi dell'albero X, secondo la sua visita in preordine.

```
preorder (void, [void]).
preorder (tree(X, Left, Right), CompleteList):-
        preorder(Left, LeftList),
        preorder(Right, RightList),
        append([X|LeftList], RightList, CompleteList).
```

Appartenenza `hasElement(X,T)`: vero quando la costante X appartiene all'albero binario T.

```
hasElement(X, tree(X, _Left, _Right)).
hasElement(X, tree(_Element,Left,_Right)):-
        hasElement(X, Left).
hasElement(X, tree(_Element, _Left, Right)) :-
        hasElement(X, Right).
```

Visita in ampiezza di un albero Definire il predicato `breadth_first(T,L)`, vero quando la lista L è la visita in ampiezza dell'albero T.

Formalizzazione ricorsiva:

$$bf(T) = \begin{cases} [void] & iff\ T = [void] \\ [E|bf([S,D])] & iff\ T = [tree(E,S,D)] \\ [void|bf(R)] & iff\ T = [void|R] \\ [E|bf([R|S,D])] & iff\ T = [tree(E,S,D)|R] \end{cases}$$

Versione PROLOG:

```
breadth_first([void], [void]).
breadth_first([tree(I, Dx, Sx)], [I|Ls]):-
        breadth_first([Dx, Sx], Ls).
breadth_first([void| Rest], [void |Ls]):-
        breadth_first(Rest, Ls).
breadth_first([tree(I, Dx, Sx)| Rest], [I|Ls]):-
        append(Rest, [Dx, Sx], Nodes),
        breadth_first(Nodes, Ls).
```

Nota: si è scelto di proporre la versione che stampa anche i nodi `void`, La modifica per ottenere un programma che stampa solamente i nodi non vuoti, risulta immediata.

Esercizi

countElems Definire un predicato `countElems(T,,N)` vero se, dato N è il numero di elementi dell'albero T.

countDeepElems Definire un predicato `countDeepElems(T,L,N)` vero se, dato N è il numero di elementi che si trovano a profondità L dell'albero T.

breadthFirst Modificare il predicato breadthFirst, in modo che non stampi i nodi void.

2.4 Negazione e taglio

2.4.1 Modificare l'ordine di ricerca: il taglio

Ci sono situazioni in cui il meccanismo di risoluzione del PROLOG risulta inefficiente: in molte circostanze, il verificarsi di una condizione esclude a priori una serie di altre possibili soluzioni. Il *taglio*, denotato con `!`, è uno strumento per limitare l'albero di ricerca. `!` è considerato come un predicato senza argomenti. `!` è un elemento extra logico.

Una generica clausola con taglio ha la seguente forma:

$$A \leftarrow B_1, \ldots, B_k, !, B_{k+2}, B_n.$$

Eseguendo un programma in cui è presente una clausola con taglio, se il goal corrente G unifica con A e $B_1, \ldots, B_k$ vengono raggiunti con successo: (i) ogni altra clausola che potrebbe unificare con G viene eliminata dall'albero di ricerca, (ii)ogni altra possibile dimostrazione per $B_1, \ldots, B_k$ viene eliminata dall'albero di ricerca.

Esempio

```
countListElems([], 0).

countListElems([X|Xs], N):-
        countListElems(X, N1),
        countListElems(Xs, N2),!,
        N is N1+N2.

countListElems(_X, 1).
```

Eliminare soluzioni valide

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

Ferma la ricerca al primo antenato di X: Z. Poichè non è detto che Y sia genitore di Z, la ricerca fallisce.

Tagli verdi e rossi Un taglio è *verde* se il comportamento del programma non cambia a seguito di una rimozione del taglio, è *rosso* altrimenti. In altre parole, data la regola:

$$A \leftarrow B_1, \ldots, B_k, !, B_{k+2}, B_n$$

un taglio è verde se, a fronte della verità di $B_1, \ldots, B_k$ tutte le soluzioni sono individuate da B_{k+2}, B_n.

Taglio verde:

```
if (B1,B2,..Bk) return f(y);     minimum(X,Y,X):-X=<Y,!.
if (B1',B2',..,Bj')return f'(y); minimum(X,Y,Y):-X>Y,!.
```

Taglio rosso:

```
if (B1,B2,..Bk) return f(y);    minimum(X,Y,X):-X=<Y,!.
return f'(y);                   minimum(X,Y,Y).
                                minimum(2,5,5)/YES????????
```

Regola di programmazione: Scrivere sempre il programma PROLOG in versione senza tagli, ed aggiungere solo tagli verdi.

Esempio di taglio

Taglio verde:

```
sort2(Xs,Ys) :- append1(As,[X,Y|Bs],Xs), X > Y, !,
                        append1(As,[Y,X|Bs],Xs1), sort2(Xs1,Ys).
sort2(Xs,Xs) :- ordered(Xs), !.
```

Taglio rosso:

```
sort3(Xs,Ys) :- append1(As,[X,Y|Bs],Xs), X > Y, !,
                append1(As,[Y,X|Bs],Xs1),
                sort3(Xs1,Ys).
sort3(Xs,Xs).
/*
 * verificare che sort4([1,3,2],[1,3,2]). \'e si
 * mentre sort3([1,3,2],[1,3,2]). \'e no
 */
```

Esercizi

primo Si realizzi un predicato *primo*(N, X, L) vero solamente se N è la posizione della prima occorrenza dell'atomo X nella lista L.

member Si applichi il taglio al programma

```
member1(X,[X|_Xs]).
member1(X,[_Y|Ys]) :- member1(X,Ys).
```

e si verifichi il comportamento nella ricerca di soluzioni multiple.

insert Si applichi il taglio al programma `insert` della sezione 2.3.3 e si disegni l'albero di ricerca per l'interrogazione `insert(4, [3,5,7], X).`

merge Si realizzi un predicato `merge(X,Y,Z)`, vero se la lista `Z` é ottenuta dal merge delle liste ordinate `X` ed `Y`.

merge Si realizzi un predicato `merge(X,Y,Z)`, vero se la lista `Z` è ottenuta dal merge delle liste ordinate `X` ed `Y`, e successivamente si applichi il taglio.

Ordinamento Si realizzi un predicato `ordinam(X,Y,Z)`, che ordina le liste `X` ed `Y` in `Z`, mediante l'algoritmo merge-sort. Mettere i tagli ai programmi `merge` e `ordinam` in modo da ottenere una sola soluzione.

2.4.2 Negazione in PROLOG

In PROLOG esiste il concetto di negazione come fallimento della ricerca. Di esso verrà data nel seguito una trattazione molto breve. Per maggiori dettagli e per una discussione delle differenze con la negazione in logica si rimanda a [4, 5].

Se `X` è una lista di atomi, `not(X)` è vero se l'interprete non riesce a trovare una dimostrazione per `X`.

Esempio

```
student(bill).
student(joe).
married(joe).

unmarriedStudent(X) :- not married(X), student(X).
? unmarriedStudent(joe).
```

La risposta `no` viene ottenuta verificando che il goal `married(joe)` ha successo.

Mentre la risposta alla interrogazione `unmarriedStudent(bill)` è `si` perché il goal `married(bill)` fallisce.

Il trattamento della negazione del PROLOG non è né corretto, né completo, a causa delle caratteristiche del modello operazionale.

Per quanto riguarda l'incompletezza si consideri il seguente esempio.

Esempio

```
p(s(X)):-p(X).
q(a).
```

L'interrogazione `not (p(X), q(X)).` non termina. Terminerebbe `not (q(X), p(X))`.

Utilizzare la negazione con atomi completamente istanziati, non genera comportamenti scorretti, nel caso la computazione dei corrispondenti predicati termini. Se si utilizza la negazione con variabili occorre tenere però in considerazione il meccanismo di esecuzione del PROLOG.

Esempio

```
unmarriedStudent(X) :- not married(X), student(X).
student(bill).
student(joe).
married(joe).
```

Il goal `unmarriedStudent(X)` fallisce anche se `X=bill` è una soluzione coerente con quanto specificato nel programma. Il problema è che il goal `not married(X)` fallisce perché la sostituzione `X=joe` porta al successo di `married(X)`. Basterebbe tuttavia scambiare l'ordine deglia atomi nella clausola per ottenere il comportamento desiserato.

```
unmarriedStudent(X) :- student(X), not married(X).
student(bill).
student(joe).
married(joe).
```

Occorre quindi utilizzare con attenzione la negazione, tenendo conto delle anomalie illustrate, soprattutto nel caso di goal non istanziati.

Esercizi

notMember Si definisca il predicato `notMember(X,L)`, vero se `X` non è membro della lista `L`, mediante la negazione.

figliounico Definire in PROLOG la relazione `figliounico(X)` sfruttando le relazioni di parentela definite nella sez. 2.1.

unione Definire la definizione di unione usando la negazione di `member`. Costruire l'albero di ricerca per l'interrogazione `? unione([a,b],[b,c],Z).`.

2.5 Esercizi Riepilogativi

Occorrenze Dato un albero binario A nella consueta rappresentazione `tree(Elemento, AlberoDestro, AlberSinistro)`, ed una lista di elementi L:

- definire un predicato `occorrenzeAlbero(E,A,N)`, vero quando l'elemento E occorre N volte nell'albero A.
- definire un predicato `occorrenzeLista(E,L,N)`, vero quando l'elemento E occorre N volte nella lista L, e tracciare l'albero di ricerca per la richiesta `occorrenzeLista(b,[a,b,b],N).`
- definire un predicato `occorrenzeAlberoLista(A,L)`, vero quando le occorrenze di un elemento nell'albero A sono uguali alle occorrenze dell'elemento nella lista L.

Sostituzione

- Si definisca un predicato `sostListaAtomi(A,B,L1,L2)`, che dati due atomi A e B, sia vero se la lista $L2$ è ottenuta dalla lista $L1$ sostituendo tutte le occorrenze di A con B.

- Si definisca un predicato `sostLista(A,B,L1,L2)`, che si comporti come quello del punto precedente, ma che operi su liste di liste.

- Si definisca un predicato `sostituzione(S,L1,L2)`, che data una lista $S = [[A_1, B_1][A_2, B_2] \ldots [A_n, B_n]]$ formata da coppie di atomi e rappresentante una sostituzione in cui gli A_i sono le variabili, i B_i sono i termini da sostituire, e due liste L_1 ed L_2, sia vero se L_2 è ottenuta da L_1 applicando la sostituzione rappresentata da S.

Capitolo 3

Applicazioni AI in PROLOG

3.1 Problemi di ricerca in PROLOG

- Astrazione;
- Individuazione di una rappresentazione per lo stato;
- Specifica di stato iniziale e stato obiettivo;
- Scelta e descrizione degli operatori;
- Definizione dei vincoli:
  - esplicitamente;
  - come precondizioni degli operatori;
  - utilizzando una rappresentazione che li soddisfi sempre.
- Ricerca di una soluzione.

L'attraversamento del fiume: LPC

Un uomo possiede un lupo, una pecora ed un cavolo e si trova su una riva di un fiume con una barca che può trasportare in un singolo viaggio - oltre lui stesso - solo uno dei suoi tre beni. L'uomo vorrebbe raggiungere l'altra sponda del fiume portando con sé integri i suoi tre beni, ma sa che solo la sua presenza può evitare che il lupo mangi la pecora, e che la pecora mangi il cavolo. Come si deve comportare?

LPC: lo spazio degli stati S Sia $S = D \times D \times D \times D$ dove $D = \{a, b\}$ con a e b che rappresentano le due sponde e $< U, L, P, C > \in D$ che rappresenta rispettivamente la posizione dell'uomo, del lupo, della pecora e del cavolo. Si definisca inoltre l'operazione di complementazione in D tale che $\overline{a} = b$ e $\overline{b} = a$.

Stato iniziale $< a, a, a, a >$
stato obiettivo $< b, b, b, b >$

LPC: gli operatori

Operatore	Condizione	Da stato	A stato
portaNiente	$L \neq P, P \neq C$	$< U, L, P, C >$	$< \overline{U}, L, P, C >$
portaLupo	$U = L, P \neq C$	$< U, L, P, C >$	$< \overline{U}, \overline{L}, P, C >$
portaPecora	$U = P$	$< U, L, P, C >$	$< \overline{U}, L, \overline{P}, C >$
portaCavolo	$U = C, L \neq P$	$< U, L, P, C >$	$< \overline{U}, L, P, \overline{C} >$

Soluzione

$< a, a, a, a > -portaPecora\rightarrow < b, a, b, a > -portaNiente\rightarrow$
$< a, a, b, a > -portaLupo\rightarrow < b, b, b, a > -portaPecora\rightarrow$
$< a, b, a, a > -portaCavolo\rightarrow < b, b, a, b > -portaNiente\rightarrow$
$< a, b, a, b > -portaPecora\rightarrow < b, b, b, b >$

Ricerca delle soluzioni in PROLOG

- Modularizzazione del procedimento di ricerca;
- Definizione ed uso dei predicati *applicabile(Azione,Stato)* e *applica(Azione,Stato,NuovoStato)*;
- Evitare gli stati ripetuti; identità tra stati; predicato *uguale(S1,S2)*;
- I predicati *statoIniziale(S)* e *statoFinale(S)*;
- Ricerca in profondità sfruttando il backtracking dell'interprete PROLOG.

Schema PROLOG

```
  soluzione(ListaAzioni):-iniziale(I),
    soluzione(I,[],ListaAzioni).
soluzione(Stato,StatiVisitati,[]) :- finale(Stato).
soluzione(Stato,StatiVisitati, [Azione|Resto]) :-
   applicabile(Azione,Stato),
   applica(Azione,Stato,NuovoStato),
   not giaVisitato(NuovoStato,StatiVisitati),
   soluzione(NuovoStato,[Stato|StatiVisitati],Resto).

giaVisitato(Stato,[StatoVisitato|AltriStatiVisitati]) :-
   uguale(Stato,StatoVisitato).
giaVisitato(Stato,[StatoVisitato|AltriStatiVisitati]) :-
   giaVisitato(Stato,AltriStatiVisitati).

uguale(S1,S2):-... (dipende dalla rappresentazione dello stato)
```

L'attraversamento del fiume

```
  iniziale(posizioni(0,0,0,0)).
finale(posizioni(1,1,1,1)).
applicabile(portaNiente,posizioni(U,L,P,C)):-L =/= P,P =/= C.
applicabile(portaLupo,posizioni(UeL,UeL,P,C)):-P =/= C.
applicabile(portaPecora,posizioni(UeP,L,UeP,C)).
applicabile(portaCavolo,posizioni(UeC,L,P,UeC)):-L =/= P.
applica(portaNiente,posizioni(U,L,P,C),posizioni(NewU,L,P,C)):-
   NewU is 1-U.
applica(portaLupo,posizioni(U,L,P,C),posizioni(NewU,NewL,P,C)):-
   NewU is 1-U, NewL is 1-L.
applica(portaPecora,posizioni(U,L,P,C),posizioni(NewU,L,NewP,C)):-
   NewU is 1-U, NewP is 1-P.
applica(portaCavolo,posizioni(U,L,P,C),posizioni(NewU,L,P,NewC)):-
   NewU is 1-U, NewC is 1-C.
```

Il problema dei due boccali

Due boccali A (capacità 4 litri) e B (3 litri) sono inizialmente vuoti e non hanno scale graduate che consentano di individuare con esattezza riempimenti intermedi. Si hanno a disposizione una fontana ed un lavandino e si vogliono mettere esattamente 2 litri di acqua nel boccale A.

Spazio degli stati

$S = \mathcal{N} \times \mathcal{N}$, dove la coppia $< Q_a, Q_b > \in S$ indica la quantità di liquido presente rispettivamente nel primo e nel secondo boccale;

Stato iniziale $< 0, 0 >$

Stato obiettivo $< 2,_>$;

Operatori

Operatore	Condizione	Da stato	A stato
vuotaA	$Q_a > 0$	$< Q_a, Q_b >$	$< 0, Q_b >$
vuotaB	$Q_b > 0$	$< Q_a, Q_b >$	$< Q_a, 0 >$
riempiA	$Q_a < 4$	$< Q_a, Q_b >$	$< 4, Q_b >$
riempiB	$Q_b < 3$	$< Q_a, Q_b >$	$< Q_a, 3 >$
vuotaAinB	$Q_a + Q_b \leq 3, Q_a > 0$	$< Q_a, Q_b >$	$< 0, Q_a + Q_b >$
vuotaBinA	$Q_a + Q_b \leq 4, Q_b > 0$	$< Q_a, Q_b >$	$< Q_a + Q_b, 0 >$
riempiAconB	$Q_b \geq 4 - Q_a, Q_a < 4$	$< Q_a, Q_b >$	$< 4, Q_b - (4 - Q_a) >$
riempiBconA	$Q_a \geq 3 - Q_b, Q_b < 3$	$< Q_a, Q_b >$	$< Q_a - (3 - Q_b), 3 >$

Soluzione

$< 0, 0 > -riempiB\rightarrow < 0, 3 > -vuotaBinA\rightarrow$
$< 3, 0 > -riempiB\rightarrow < 3, 3 > -riempiAconB\rightarrow$
$< 4, 2 > -vuotaA\rightarrow < 0, 2 > -vuotaBinA\rightarrow$
$< 2, 0 >$

Missionari e cannibali

Tre missionari e tre cannibali si trovano su una stessa sponda di un fiume e vogliono attraversarlo. Hanno a disposizione una barca che può trasportare una o due persone. E' bene però che i cannibali non si trovino mai a sopravanzare in numero i missionari.

Un primo approccio

MC: Spazio degli stati

$S = \mathcal{N}^6$, con $< M_a, C_a, B_a, M_b, C_b, B_b > \in S$ che rappresenta rispettivamente il numero di missionari, cannibali e barche sulla sponda di partenza (sponda a) e su quella di arrivo (sponda b); Stato iniziale $< 3, 3, 1, 0, 0, 0 >$ stato obiettivo $< 0, 0, 0, 3, 3, 1 >$

Operatore 1

- nome: $porta(M', C')$
- significato: porta M' missionari e C' cannibali dalla sponda iniziale all'altra;
- precondizioni per l'applicazione nello stato $< M_a, C_a, B_a, M_b, C_b, B_b >$:
  - assicurare posizione della barca: $B_a = 1$
  - rispettare capienza della barca: $1 \leq M' + C' \leq 2$
  - evitare quantità negative: $M' \leq M_a$ e $C' \leq C_a$
  - evitare di essere mangiati: ($M_a - M' = 0$ oppure $C_a - C' \leq M_a - M'$) e ($M_b + M' = 0$ oppure $C_b + C' \leq M_b + M'$)
- nuovo stato: $< M_a - M', C_a - C', 0, M_b + M', C_b + C', 1 >$

Operatore 2

- nome: $riporta(M', C')$
- significato: riporta M' missionari e C' cannibali sulla sponda iniziale;
- precondizioni per l'applicazione nello stato $< M_a, C_a, B_a, M_b, C_b, B_b >$:
  - assicurare posizione della barca: $B_b = 1$
  - rispettare capienza della barca: $1 \leq M' + C' \leq 2$
  - evitare quantità negative: $M' \leq M_b$ e $C' \leq C_b$
  - evitare di essere mangiati: ($M_a + M' = 0$ oppure $C_a + C' \leq M_a + M'$) e ($M_b - M' = 0$ oppure $C_b - C' \leq M_b - M'$)
- nuovo stato: $< M_a + M', C_a + C', 1, M_b - M', C_b - C', 0 >$

Soluzione
$< 3, 3, 1, 0, 0, 0 > -porta(1, 1) \rightarrow < 2, 2, 0, 1, 1, 1 >$
$-riporta(1, 0) \rightarrow < 3, 2, 1, 0, 1, 0 >$
$-porta(0, 2) \rightarrow < 3, 0, 0, 0, 3, 1 >$
$-riporta(0, 1) \rightarrow < 3, 1, 1, 0, 2, 0 >$
$-porta(2, 0) \rightarrow < 1, 1, 0, 2, 2, 1 >$
$-riporta(1, 1) \rightarrow < 2, 2, 1, 1, 1, 0 >$
$-porta(2, 0) \rightarrow < 0, 2, 0, 2, 0, 1 >$
$-riporta(0, 1) \rightarrow < 0, 3, 1, 3, 0, 0 >$
$-porta(0, 2) \rightarrow < 0, 1, 0, 3, 2, 1 >$
$-riporta(0, 1) \rightarrow < 0, 2, 1, 3, 1, 0 >$
$-porta(0, 2) \rightarrow < 0, 0, 0, 3, 3, 1 >$

Missionari e cannibali: formulazione alternativa

Si potrebbe codificare in maniera implicita le presenze numeriche sulla sponda b come complemento ai valori relativi alla sponda a.

$S = \mathcal{N} \times \mathcal{N} \times \mathcal{N}$ con $< M, C, B > \in S$ che rappresenta il numero di missionari M, di cannibali C e di barche B sulla sponda iniziale.

Nessuno si perde, nessuno scappa, la barca non affonda, ecc. si può calcolare che nello stato $< M, C, B >$ ci sono $3 - M$ missionari, $3 - C$ cannibali e $1 - B$ barche sulla sponda b.

Operatore 1

- nome: $porta(M', C')$
- significato: porta M' missionari e C' cannibali dalla sponda iniziale all'altra;
- precondizioni per l'applicazione nello stato $< C, M, B >$:
  - assicurare posizione della barca: $B = 1$
  - rispettare capienza della barca: $1 \leq M' + C' \leq 2$
  - evitare quantità negative: $M' \leq M$ e $C' \leq C$
  - evitare di essere mangiati: ($M - M' = 0$ oppure $C - C' \leq M - M'$) e ($M - M' = 3$ oppure $C - C' \geq M - M'$)
- nuovo stato: $< M - M', C - C', 0 >$

Operatore 2

- nome: $riporta(M', C')$
- significato: riporta M' missionari e C' cannibali sulla sponda iniziale;

- precondizioni per l'applicazione nello stato $< C, M, B >$:

  - assicurare posizione della barca: $B = 0$
  - rispettare capienza della barca: $1 \leq M' + C' \leq 2$
  - evitare quantità negative: $M' \leq 3 - M$ e $C' \leq 3 - C$
  - evitare di essere mangiati: $(M + M' = 0$ oppure $C + C' \leq M + M')$ e $(M + M' = 3$ oppure $C + C' \geq M + M')$

- nuovo stato: $< M + M', C + C', 1 >$

Soluzione

$< 3, 3, 1 > -porta(1, 1) \rightarrow < 2, 2, 0 >$
$-riporta(1, 0)) \rightarrow < 3, 2, 1 >$
$-porta(0, 2) \rightarrow < 3, 0, 0 >$
$-riporta(0, 1) \rightarrow < 3, 1, 1 >$
$-porta(2, 0) \rightarrow < 1, 1, 0 >$
$-riporta(1, 1)) \rightarrow < 2, 2, 1 >$
$-porta(2, 0) \rightarrow < 0, 2, 0 >$
$-riporta(0, 1) \rightarrow < 0, 3, 1 >$
$-porta(0, 2) \rightarrow < 0, 1, 0 >$
$-riporta(0, 1) \rightarrow < 0, 2, 1 >$
$-porta(0, 2) \rightarrow < 0, 0, 0 >$

Missionari e cannibali: ancora una rappresentazione

Rappresentazione che soddisfa implicitamente i vincoli sul numero di missionari e cannibali per sponda.

$S = \mathcal{N}^6$ dove $< DM_a, C_a, B_a, DM_b, C_b, B_b > \in S$ dove DM_a rappresenta la differenza (intero positivo o nullo) tra il numero di missionari e quello di cannibali sulla sponda a, e analogamente DM_b per la sponda b.

Poichè DM_a e DM_b sono interi positivi o nulli per definizione, questa rappresentazione esclude le situazioni in cui i vincoli sulla numerosità relativa di missionari e cannibali non sono rispettati.

Questo è ovviamente un vantaggio in termini di ampiezza dello spazio di ricerca.

3.1.1 Esercizi

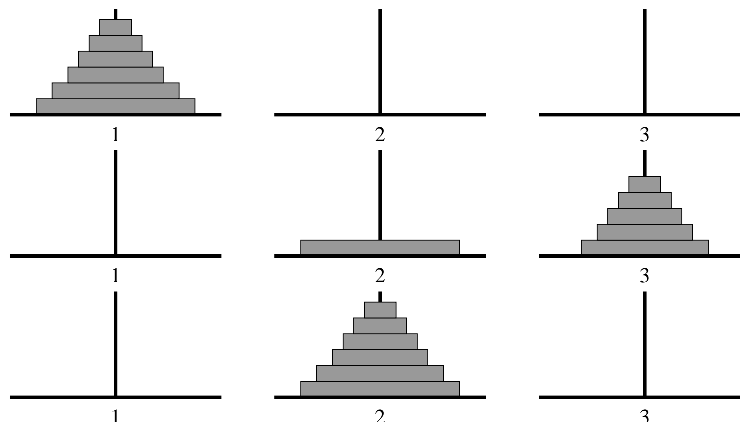
Ascensore In un grattacielo ci sono 3 coppie formate da marito e moglie. Il cancello delle scale viene chiuso e l'unico modo per scendere è con l'ascensore che può portare 3 persone alla volta e deve sempre portare almeno un passeggero. I mariti non vogliono che le rispettive mogli si ritrovino in presenza di altri mariti se non sono presenti essi stessi. Come devono scendere le 3 coppie senza crisi di gelosia?

Formalizzare la soluzione come un problema di ricerca.

Torri di Hanoi Il gioco delle Torri di Hanoi ha origine da un'antica leggenda Vietnamita, secondo la quale un gruppo di monaci sta spostando una torre di 64 dischi (secondo la leggenda, quando i monaci avranno finito, verrà la fine del mondo). Lo spostamento della torre di dischi avviene secondo le seguenti regole:

- inizialmente, la torre di dischi di dimensione decrescente è posizionata su un perno 1;
- l'obiettivo è quello di spostarla su un perno 2, usando un perno 3 di appoggio;
- le condizioni per effettuare gli spostamenti sono:
  - tutti i dischi, tranne quello spostato, devono stare su una delle torri
  - è possibile spostare un solo disco alla volta, dalla cima di una torre alla cima di un'altra torre;
  - un disco non può mai stare su un disco più piccolo.

Lo stato iniziale, uno stato intermedio, e lo stato finale per un insieme di 6 dischi sono mostrati nelle seguenti figure:



Formalizzare la soluzione del gioco delle torri di Hanoi come problema di ricerca.

3.2 Classici AI

Nella ricerca di soluzioni è spesso utile ricorrere al non determinismo del PROLOG. Il fatto che tale linguaggio sia in grado di restituire risultati multipli può essere convenientemente usato nei problemi di ricerca in cui si disponga due predicati:

- $test(X)$, vero se X è una soluzione valida
- $generate(X)$, vero se X è una possibile istanza di soluzione.

In tale caso, grazie al non determinismo del PROLOG, chiamate successive a *generate* restituiranno istanze di soluzione diverse. La verità di ognuna di tali istanze è verificabile mediante *test*, secondo il seguente schema:

```
find(X) :- generate(X), test(X).
```

Esempio: Ordinamento e parsing

```
%ordinamento
sort(X,Y):- permutation(X,Y), ordered(Y).
%parsing
parse(X):- append(Y,Z,X), soggetto(Y), verbo(Z).
```

Per motivi di efficienza risulta spesso utile *anticipare il test*, accorpandolo alla fase di generazione in modo da scartare in anticipo soluzioni non valide. Nei due casi (ordinamento e parsing) in esame, due ottimizzazioni possibil sono le seguenti:

- L'ordinamento anzichè generare una permutazione casuale mette in prima posizione il minore.
- Il parsing non fa tutte le decomposizioni possibili ma solo quelle tali per cui la prima parte della stringa viene riconosciuta.

Esempio: Intersezione non vuota

```
nonEmptyIntersection(Xs,Ys):-member(X,Xs),member(X,Ys).

member(X,Xs):- append(A,[X|B],Xs).

permutation([],[]).
permutation(Xs,[Z|Zs]) :- select(Z,Xs,Ys), permutation(Ys,Zs).

select(X,[X|Xs],Xs).
select(X,[Y|Ys],[Y|Zs]) :- select(X,Ys,Zs).
```

Esercizio Si vuole implementare un programma PROLOG per trovare il codice di una cassaforte, rappresentato come una sequenza di numeri di lunghezza sconosciuta.

Il programma puó eseguire il test di apertura della cassaforte mediante il predicato `apre(X)`, vero se la combinazione X apre la cassaforte.

Risolvere il problema con il metodo *generate and test*, si consiglia di articolare il programma in tre predicati: `generaCombinazione(L,X)`, che genera una combinazione di lunghezza L, `generaCombinazioni(X,...)`, che genera tutte le combinazioni, in ordine lessicografico, ed infine `trovaCombinazione(X)`, che genera tutte le combinazioni e controlla se la cassaforte è aperta.

Problema delle N-regine

Data una scacchiera avente lato di dimensione N si vogliono posizionare su questa N regine in modo che nessuna possa essere minacciata dalle altre[1].

[1]Nel gioco degli scacchi una regina in posizione x, y minaccia tutti i pezzi che si trovano sulla stessa riga, sulla stessa colonna o che giacciono su una delle due diagonali passanti per la regina.

```
queens(N,Qs) :- range(1,N,Ns), permutation(Ns,Qs), safe(Qs).
safe([Q|Qs]) :- safe(Qs), not attack(Q,Qs).
safe([]).
attack(X,Xs) :- attack(X,1,Xs).
attack(X,N,[Y|Ys]) :- X is Y+N ; X is Y-N.
attack(X,N,[Y|Ys]) :- N1 is N+1, attack(X,N1,Ys).
range(M,N,[M|Ns]) :- M < N, M1 is M+1, range(M1,N,Ns).
range(N,N,[N]).
```

Le prestazioni migliorano anticipando il test:

```
queens1(N,Qs) :- range(1,N,Ns), queens1(Ns,[],Qs).

queens1(UnplacedQs,SafeQs,Qs) :-
         select(Q,UnplacedQs,UnplacedQs1),
         not attack(Q,SafeQs),
         queens1(UnplacedQs1,[Q|SafeQs],Qs).
queens1([],Qs,Qs).
```

Colorazione Mappe

Data una mappa , si vuole trovare una assegnazione di colori per le regioni in modo che non esistano due regioni adiacenti a cui è associato lo stesso colore. I colori sono in numero finito.

La mappa si puó rappresentare come un grafo, i cui nodi rappresentano le regioni ed in cui un arco tra due nodi a e b rappresenta il fatto che le regioni corrispondenti ad a e b sono adiacenti.

La mappa è una lista di regioni. Ogni regione è caratterizzata da un nome, un colore ed una lista di vicini (Neighbors). Ai fini della soluzione del problema i vicini sono caratterizzati solamente dal colore.

```
[region(a,A,[B,C,D]),     region(b,B,[A,C,E]),
region(c,C,[A,B,D,E,F]), region(d,D,[A,C,F]),
region(e,E,[B,C,F]),     region(f,F,[C,D,E])]
```

Nota: questa struttura si presta al side effect.

Scheletro di soluzione:

- scegli il colore per una regione (in modo *opportuno*).
- verifica la scelta per le altre regioni.

```
color_map([Region|Regions],Colors) :-
        color_region(Region,Colors),
        color_map(Regions,Colors).
color_map([],_Colors).
```

Scelta del colore per la regione Scelgo un colore a caso, tra quelli dei vicini, escludendo quelli già assegnati.

```
color_region(region(_Name,Color,Neighbors),Colors) :-
        /*seleziona in Color un colore, in Colors1 ci sono
        tutti i colori tranne quello scelto*/
        select(Color,Colors,Colors1),
        /*verifica che il colore dei vicini del nodo
        appartengano tutti ai colori non scelti*/
        members(Neighbors,Colors1).

members([X|Xs],Ys) :- member(X,Ys), members(Xs,Ys).
members([],_Ys).
```

Analogy

Date tre figure, a, b, c, ed un insieme di risposte R, si chiede di trovare una relazione tra a e b, ed applicare tale relazione a c. L'applicazione della relazione a c determina una nuova figura candidata: d. Si vuole trovare la figura $f \in R$, che meglio approssima d.

Algoritmo di base:

$r = trovaRelazione(a, b)$
$f = applicaRelazione(a, b)$
if $f \in R$ return f
return fail

```
/*Definizione dell'operatore infisso is_to, con priorita' 100
A is_to B equivale a is_to(A,B).
*/
:- op(100,xfx,[is_to]).

analogy(A is_to B,C is_to X,
        match(A,B,Match),
        match(C,X,Match),
        member(X,Answers).
```

Il predicato `match(A,B,relazione)` definisce come viene formulata una relazione.

```
match(_X,_X,same).
match(inside(Figure1,Figure2),inside(Figure2,Figure1),invert).
match(above(Figure1,Figure2),above(Figure2,Figure1),invert).
match(top(Figure1,Figure2),top(Figure2,Figure1),invert).
match(onBorder(Figure1,Figure2),onBorder(Figure2,Figure1),invert).
```

Bibliografia

[1] A. Marchetti Spaccamela, D. Nardi, *Principi di programmazione*, Calderini, 2003.

[2] D. Friedman, M. Fellesein, *The Little Lisper*, MIT Press, 1987.

[3] G. Steele, *COMMON LISP 2nd ed.*, Thinking Machines, 1990.

[4] L. Console, E. Lamma, P. Mello, *Programmazione logica*, Utet, 1991.

[5] L. Sterling, E. Shapiro, *The art of PROLOG*, MIT Press, 1994.

[6] M. Ginzberg, Artificial Intelligence Techniques in PROLOG, Morgan Kaufman, 1987.

[7] S. Russell, P. Norvig, *Intelligenza Artificiale, un approccio moderno*, UTET-Libreria, 1998; si veda anche: S. Russell, P. Norvig, *Artificial Intelligence, A Modern Approach 2nd ed.*, Prentice Hall, 2003.

[8] N.J. Nilson, *A*rtificial intelligence: A new Synthesis, Morgan Kaufmann Pub., 1998.