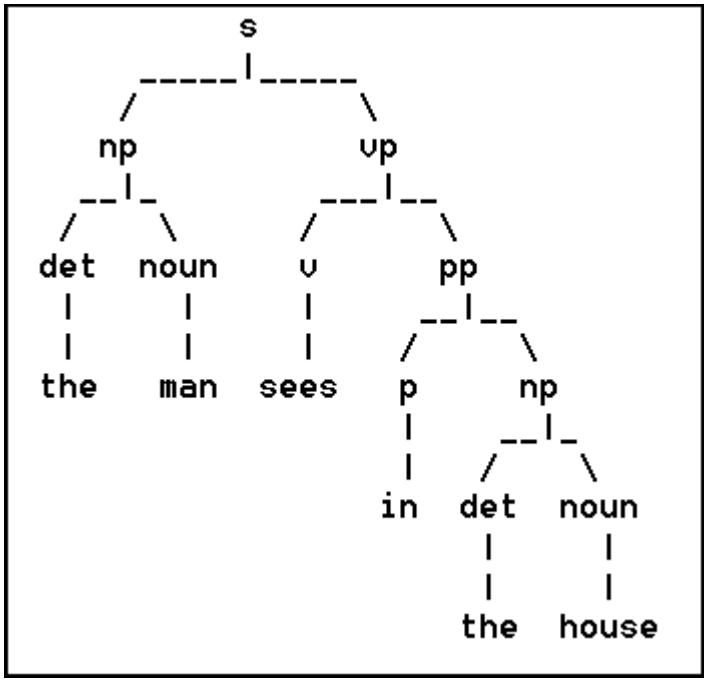
Piccola introduzione al Prolog per docenti di materie umanistiche

v. 67 del 14/01/2017



http://www.nyu.edu/pages/linguistics/workbook/lehner/christop.gif

Indice generale

Come e perché Prolog

Questa breve introduzione, come dice il titolo, è dedicata a insegnanti di materie umanistiche con nessuna o pochissima conoscenza di linguaggi di programmazione. In effetti, non fa parte dei prerequisiti avere esperienza in questo campo, perché le caratteristiche del Prolog sono tali da renderlo piuttosto indipendente e forse unico. Anzi: molti problemi per i neofiti derivano dal tentativo di interpretare dei costrutti del Prolog come se fossero istruzioni di un linguaggio imperativo; il che di solito è un errore. Meglio, quindi, partire da zero.

Perché il Prolog? Perché ha una struttura semplice ed elegante, si può utilizzare da subito anche senza conoscere tutti i dettagli avanzati, e perché ha degli aspetti "magici" che ne rendono l'uso anche divertente. E perché è particolarmente adatto a trattare delle conoscenze di tipo non numerico, quindi si presta a fare "coding" anche nell'ora di Italiano (vedi bibliografia).
Prolog è un linguaggio davvero particolare. Per certi scopi è di un'efficacia spaventosa. Ad esempio, per scrivere un programma che risolva ogni tipo di Sudoku sono sufficienti 16 righe di codice. Non male, no?[1]
Esistono varie versioni di Prolog e varie implementazioni di compilatori, alcune delle quali OpenSource, come Gnuprolog[2] o come SWI-Prolog.[3] SWI-Prolog, in particolare, può essere scaricato e installato su diversi sistemi operativi; tuttavia è possibile iniziare (per esempio per esercitarsi con gli esempi di questo tutorial) usando la versione online all'indirizzo http://swish.swi-prolog.org/. Ancora meglio: è possibile usare l'interprete online *mentre* si studia un tutorial Prolog, come si può vedere qui: http://lpn.swi-prolog.org/lpnpage.php?pagetype=html&pageid=lpn-htmlse1.

Un programma Prolog è un insieme ordinato di fatti e di regole. Per eseguire il codice sorgente degli esempi sul proprio computer occorre:

- scriverlo con un qualsiasi editor testuale e salvarlo come file di testo (ad esempio come esempio.pl),
- lanciare Prolog (es. "swipl" o "gprolog")
- caricare (e compilare) il sorgente nella memoria Prolog scrivendo:
  `consult(esempio).`[4]
- richiedere a Prolog di dimostrare un *goal*

Un goal, come vedremo tra poco, è semplicemente una frase scritta in Prolog, di cui vogliamo sapere se è dimostrabile all'interno del programma che abbiamo caricato. *Compilare* un sorgente significa trasformare un testo scritto in qualcosa di direttamente eseguibile dal computer. Per questo, se vogliamo fare delle modifiche al programma, dobbiamo modificare il testo, salvarlo e ricompilarlo nuovamente.
In una fase di apprendimento iniziale, può essere interessante (oltre che utile per trovare eventuali errori) vedere in diretta come Prolog cerca di trovare la soluzione al goal che abbiamo scritto. Per fare questo occorre attivare la modalità di tracciamento scrivendo:

```
trace.
```

In questa modalità, Prolog mostra un passo alla volta e resta in attesa di una conferma per andare avanti; conferma che gli possiamo dare semplicemente premendo il tasto Invio tante volte quante sono necessarie per arrivare alla fine dell'elaborazione.

---

1 https://www.metalevel.at/sudoku/sudoku
2 http://gprolog.org/
3 http://www.swi-prolog.org/
4 In tutto questo tutorial, le istruzioni in Prolog sono scritte con un carattere diverso, cioè

```
così
```

Per chi proprio non può fare a meno di sapere tutto prima di cominciare: Prolog significa "Programming in Logic" (o meglio "programmation en logique"), ed è il nome dato ad un linguaggio inventato all'Università di Marsiglia all'inizio degli anni 70 da Alain Colmerauer e Philippe Roussel[5] partendo da ricerche precedenti di Robert Kowalski all'università di Edimburgo. Insomma, un linguaggio tutto europeo. Ha avuto un grande successo negli ultimi anni del millennio scorso in connessione con il boom del mito di un'intelligenza artificiale forte che era lì, appena un passo oltre l'orizzonte, e presto sarebbe stata una realtà. Invece quel mito si è rivelato una favola, mentre un altro tipo di intelligenza – quella più opaca del *machine learning* – sta prendendo piede. Qui però non parliamo di intelligenza, ma solo di rappresentazione formale della conoscenza e di ragionamenti deduttivi.

Per "rappresentazione della conoscenza" si intende un processo che parte dalla raccolta dei "fatti" e delle relazioni, li organizza in strutture e li archivia in qualche modo per il riuso automatico. Non è chiarissimo se si parta davvero dalla raccolta dei fatti, oppure dalla definizione delle strutture preparate per contenerli, oppure dai problemi che si vogliono risolvere. Probabilmente un misto dei tre approcci: se il primo obiettivo che ci si pone definisce un dominio che si rivela troppo poco strutturato, allora si cominciano a raccogliere i fatti, e intanto si ipotizzano delle strutture dove raccoglierli, mentre gli obiettivi si precisano e si arricchiscono man mano che si procede.
La mia impressione è che quando si usa un approccio di questo tipo, che si chiama "dichiarativo" – a differenza dall'approccio tradizionale nella programmazione che viene chiamato "procedurale" - è perché non si hanno ancora chiari gli obiettivi: ci si prepara per ogni evenienza futura, per così dire.

5 http://alain.colmerauer.free.fr/alcol/ArchivesPublications/PrologHistory/19november92.pdf

Genealogie

Nei testi italiani di introduzione al Prolog si parte spesso da esempi di famiglie e genealogie (negli Stati Uniti invece l'esempio preferito è quello delle caratteristiche delle automobili…). E' un esempio che funziona bene perché tutti sanno cos'è un albero genealogico, conoscono le relazioni tra i suoi elementi, sia quelle dirette (padre, figlio) che quelle indirette (nonno, zio, cognato). Lo useremo anche noi, espandendolo un po' alla volta.
Ad esempio, immaginiamo di avere una famiglia (per ora tutta maschile):

Mario è il padre di Mattia e di Francesco.
Mario è figlio di Gino; suo padre si chiama Franco.
...

E' abbastanza chiaro per un parlante Italiano (tranne quel "suo" che è un po' ambiguo), ma potrebbe essere un po' complicato da spiegare a qualcuno di una lingua e di cultura molto diversa dall'Italiano. Possiamo cominciare a semplificare, a togliere i riferimenti anaforici, le congiunzioni, i possessivi, fino ad arrivare a riscrivere tutto usando una sola forma:

1. *Mario è il padre di Mattia.*
2. *Mario è il padre di Francesco.*
3. *Gino è il padre di Mario.*
4. *Franco è il padre di Gino.*

La prima di queste frasi potrebbe interpretata così:

Ci sono due individui, Mario e Mattia.
Tra questi due individui c'è una relazione "essere padre di".
La relazione ha un verso: da Mario a Mattia.

Attenzione al fatto che le frasi non contengono nessuna informazione sul fatto che non sia possibile, ad esempio, che Mattia sia il padre di Mario. Questa è una conoscenza del significato della relazione "essere padre di" che abbiamo noi umani, ma che al momento non abbiamo modo di rappresentare. Se avessimo scritto:

Ci sono due individui, Francesco e Mattia.
Tra questi due individui c'è una relazione "essere fratello di".
La relazione ha un verso: da Francesco a Mattia.

la forma resterebbe la stessa, ma a noi verrebbe spontaneo pensare che sia possibile anche scrivere "Mattia è fratello di Francesco". Però, al momento, non sappiamo come distinguere questi due casi utilizzando solo la forma della frase.

L'oscillazione tra sintassi e semantica è una delle difficoltà che si incontrano quando si cerca di rappresentare formalmente la conoscenza su un argomento senza fare affidamento su implicazioni nascoste. E' per questo che è un esercizio utile: perché costringe a trasformare la conoscenza tacita in conoscenza esplicita.
Rappresentare con un linguaggio formale questa conoscenza significa utilizzare una sintassi (e un lessico) ben definiti che escludano il più possibile varianti e ambiguità. E' inutile qui tirare in ballo la stupidità dei computer che sanno fare le cose in maniera bruta senza un briciolo di creatività: per il momento stiamo parlando solo di linguaggi formali di rappresentazione. Perché abbia senso l'operazione di traduzione da una lingua naturale ad una artificiale dobbiamo perdere in ricchezza e guadagnare in rigore.
Ad esempio, potremmo – invento - scrivere qualcosa del tipo:

Mario#Padre#Mattia

usando il segno # per congiungere i termini della relazione.

ma non ci sarebbe modo di sapere, semplicemente leggendo l'espressione, quali sono gli individui e quali le relazioni. Meglio allora sarebbe:

Mario#Padre:Mattia

Tanto varrebbe però scrivere:

Mario = padre (Mattia)

che sarebbe forse più chiaro, almeno per un certo tipo di lettori, perché assomiglia ad una scrittura matematica che gli è familiare:

x = f(y)

E infatti questa è una forma molto usata in altri linguaggi di programmazione. Ma ha il difetto di portare con sé un sottotesto, un'implicazione. Sembra voler dire qualcosa del tipo:

Definisco un oggetto, Mario, come padre di Mattia. D'ora in poi, ogni volta che uso Mario posso intendere "padre di Mattia".

Cioè il centro del discorso è Mario e viene definita una regola di sostituzione.

La forma usata dal linguaggio Prolog, nella versione indicata come "sintassi di Edimburgo" dal nome dell'Università dove è stata inventata (e che è anche quella adottata da SWI Prolog), è invece derivata dalla logica dei predicati del primo ordine[6] ed è questa:

```
padre(mario,mattia).
```

Viene prima la relazione, che si chiama "predicato", poi tra parentesi i due oggetti di cui si predica la relazione (ovvero gli argomenti del predicato). Alla fine di ogni dichiarazione deve esserci un punto, che la separa da quelle seguenti. Queste dichiarazioni in Prolog si chiamano *fatti.*

Volendo dare una traduzione Italiana più vicina letteralmente a quanto abbiamo scritto sopra, possiamo propendere per lo stile matematico:

"*padre è una relazione che esiste tra mario e mattia"*

oppure per quello logico:

"*padre si predica degli argomenti mario e mattia"*

Non stiamo dicendo niente sull'esistenza di Mario o di Mattia, non stiamo definendo in maniera particolare né l'uno né l'altro. L'attenzione, in effetti, non è sugli individui, ma sui predicati.

I fatti possono avere diverse forme. Possono essere, molto semplicemente:

```
mario.
```

Che equivale a dire: mario è un fatto della nostra "base di conoscenze".

Oppure:

```
maschio(mario).
```

Che equivale a dire:

il fatto che si predica maschio di mario ***appartiene*** *alla nostra "base di conoscenze".*

Si potrebbe essere tentati di interpretare in maniera un po' più forte `maschio(mario)`:

il fatto che si predica maschio di mario ***è vero***.

Per certi versi, non c'è una differenza tra questi due modi di intendere un predicato. Vero è solo quello che è presente nella base di conoscenze (o che può essere dimostrato a partire da questa). Di conseguenza, falso è ciò che non è nella base di conoscenza o che non può essere dimostrato a partire da questa. Su questa interpretazione di vero e falso torneremo più avanti.

6 "Logica del primo ordine" si riferisce al fatto che non è possibile per un predicato essere argomento di un altro predicato. Mentre si può rappresentare con questa logica una frase del tipo "Guglielmo conosce il colpevole" come conosce(guglielmo,colpevole), non si può scrivere "Adelmo sa che Guglielmo conosce il colpevole" come sa(adelmo(conosce(guglielmo,colpevole)).

Regole

Torniamo alla genealogia. Per completare la rappresentazione in Prolog della nostra genealogia possiamo aggiungere le relazioni seguenti:

```
padre(mario,francesco).
padre(gino, mario).
padre(franco,gino).
```

Però noi - nella nostra cultura occidentale costruita sull'esperienza greca eccetera - sappiamo fare delle inferenze, cioè sappiamo di poter sapere delle cose ancora prima di saperle. Sappiamo, per esempio, che se Mario è il padre di Mattia, allora Mattia è figlio di Mario. Potrebbe essere utile rappresentare esplicitamente anche questa relazione, non si sa mai. Dobbiamo allora aggiungere, tra i fatti, che mattia è figlio di mario, e così via, cioè dobbiamo scrivere:

```
figlio(mattia,mario).
figlio(mario, gino).
figlio(gino,franco).
```

Si, potremmo farlo. Ma è lungo, e tutto sommato si tratta di relazioni già implicate dalla base di conoscenza che abbiamo creato fino a questo punto. Noi umani sappiamo che figlio è la relazione inversa di padre. Ovvero: per dire se qualcuno è figlio di qualcun altro non abbiamo bisogno di conoscenze ulteriori, oltre quelle che già abbiamo. Ci deve essere un modo per utilizzare questa conoscenza implicita.
E infatti c'è: non serve definire esplicitamente tutti i casi in cui figlio si predica di qualcuno; possiamo utilizzare una forma diversa: una *regola*. Una regola, a differenza di un fatto, non è un'espressione di uno stato di cose, ma di una relazione tra stati di cose.
La forma delle regole, in Prolog, è unica e piuttosto semplice: c'è un predicato (la testa della regola) che è messo in relazione condizionale con uno o più altri predicati o fatti (il corpo). Se lo stato di cose descritto dal corpo si verifica, allora "scatta" anche il predicato descritto nella testa della regola. Su questo movimento meccanico torneremo più avanti parlando del funzionamento del Prolog.

La regola che ci serve dovrebbe più o meno dire:

> *Se qualcuno (A) è padre di qualcun altro (B), allora quel qualcun altro (B) è figlio di quel qualcuno (A).*

Ma questa non sarebbe una forma adatta al Prolog. Invece dobbiamo immaginare qualcosa di questo tipo:

> *figlio si predica di due individui A e B* ***se***
>
> > *padre si predica di quegli stessi due individui.*

Attenzione al fatto che padre, per come l'abbiamo scritto sopra, è un predicato con due argomenti, in cui l'ordine è importante, cioè:

```
padre(mattia,mario)
```

è diverso da:

```
padre(mario,mattia).
```

il che è piuttosto ovvio se ci ricordiamo che in Prolog non esistono omonimi.
Dobbiamo perciò trovare un modo di rappresentare questo ordine nella nuova regola. Cioè vorremmo poter scrivere:

> *figlio si predica di due individui, indicati come Figlio e Padre,* ***se***
>
> > *padre si predica di quegli stessi due individui, indicati come Padre e Figlio.*

In Prolog questa regola si scrive così:

```
figlio(Figlio,Padre) :-
  padre(Padre,Figlio).
```

Il "se" che separa le due parti della regola è indicato dai due segni ":-" e dall'acapo (che non è necessario, ma è comodo).
Sul significato di questo "se" si potrebbe dire qualcosa di più. Intanto: NON sta per "se e solo se". Cioè non stiamo dicendo che:

*figlio si predica di due individui, indicati come Figlio e Padre, **se e solo se***
padre si predica di due individui, indicati come Padre e Figlio.

perché potrebbero esserci delle altre maniere per definire "figlio".

Per esempio questa:

```
figlio(Figlio,Madre) :-
  madre(Madre,Figlio).
```

Un'altra possibile (ma errata) interpretazione del simbolo ":-" che potrebbe venire in mente a chi ha una anche minima esperienza di altri linguaggi di programmazione è questa:

se
padre(Padre,Figlio)
allora
figlio(Figlio,Padre)

cioè: se è vero che padre si predica di due individui, allora è vero che figlio si predica degli stessi individui. Finché restiamo nell'ambito della rappresentazione statica della conoscenza (cioè vogliamo solo descriverla, disegnarla, non vogliamo fare nessuna azione automatica su di essa) è la stessa cosa. Ma vedremo meglio che Prolog funziona al contrario. Non parte dalla condizione, ma dall'obiettivo da raggiungere. Abituarsi a ragionare in questi termini aiuta a progettare una rappresentazione efficace, oltre che corretta.

Continuiamo a rappresentare le relazioni nella famiglia allontanandoci nel tempo. Non ci sono solo padri e figli: ci sono i nonni e i nipoti.
C'è bisogno di scrivere esplicitamente che gino è il nonno di mattia? Forti dell'esperienza appena fatta, rispondiamo di no, perché noi sappiamo che la relazione nonno-nipote non è una relazione diretta, ma il prodotto di due relazioni padre-figlio tra tre individui in cui uno è una volta padre, una volta figlio:

*nonno si predica di due individui, indicati come Nonno e Nipote, **se***
*padre si predica di due individui, indicati come Padre e Nipote, **e***
padre si predica di due individui, indicati come Nonno e Padre.

Ovvero:

```
nonno(Nonno,Nipote) :-
  padre(Padre,Nipote),
  padre(Nonno,Padre).
```

Abbiamo messo una virgola tra la prima e la seconda condizione: è un segno importante, significa che le due condizioni devono essere *entrambe* soddisfatte perché lo sia la regola nel suo complesso.
Si potrebbe scrivere anche la regola per definire nipote:

```
nipote(Nipote,Nonno):-
  figlio(Nipote,Padre),
  figlio(Padre,Nonno).
```

che sarebbe identica alla precedente, salvo il fatto che usa figlio/2 anziché padre/2.[7]

7 La scrittura figlio/2 significa che il predicato padre ha due argomenti. Si usa per distinguere predicati diversi che hanno lo stesso nome ma un numero di argomenti differente (come potrebbe essere figlio/3 che mette in relazione un Figlio con due Genitori).

Come volevasi dimostrare

Se Prolog fosse solo un linguaggio per rappresentare formalmente una conoscenza, sarebbe interessante ma utile fino ad un certo punto. Per fortuna Prolog è anche un meccanismo di deduzione di nuovi fatti a partire da fatti e regole. Deduzione in senso stretto: non inventerà mai nulla, ma troverà tutte le conoscenze che sono ricavabili dai fatti che abbiamo dichiarato, tramite le regole che abbiamo definito.
Prolog è in grado di fare due cose:

1. se gli forniamo un fatto (del tipo "Si predica p di a"), prova a dimostrarlo. Se non ci riesce, concluderà che quel fatto è falso, altrimenti che è vero.[8]
2. se gli forniamo un obbiettivo del tipo "Si predica p di A?", prova a raggiungerlo sostituendo ad A il (o i) termine/i che soddisfano quel predicato.

La differenza non sta nel punto interrogativo alla fine (che difatti non si scrive) ma nelle lettere maiuscole o minuscole. Per Prolog c'è una differenza enorme tra A e a.

- a è il nome di un **termine**, come gino o mario
- A è un **segnaposto** che Prolog proverà a sostituire con il nome di un termine, se riuscirà a soddisfare il predicato p.

Prolog è insomma una specie di oracolo in attesa di domande. A volte sa rispondere, a volte no; ma se non sa rispondere, conclude che non c'è risposta. Questa si chiama "assunzione del mondo chiuso" ed è molto importante per far si che il programma non giri all'infinito.
Per fortuna le sue risposte sono di solito più chiare di quelle degli oracoli classici, purché lo siano anche le domande. In effetti le domande possibili sono di tre tipi:

1a. Mario è il padre di Mattia?
Ovvero:

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

Qui stiamo solo verificando che Prolog si ricordi quello che abbiamo inserito esplicitamente. Bene, ma un po' pochino. Lo fa qualsiasi database. Potevamo anche chiedere:

1b. Gino è il nonno di Mattia?
Ovvero:

```
?nonno(gino,mattia).
```

Qui invece facciamo già qualcosa di più: gli chiediamo se è dimostrabile la relazione di nonno tra gino e mattia, relazione che esplicitamente non è mai stata dichiarata come un fatto.

Il secondo tipo di domanda è quello in cui non ci limitiamo a chiedere se è dimostrabile un'espressione, ma vogliamo anche sapere *con quali valori lo sarebbe*.

2a. Chi è il nipote di Gino?
Per chi ama il linguaggio pseudomatematico: esiste un X tale che gino è nonno di X?

```
?nonno(gino,Nipote).
```

8 In realtà Prolog prova a dimostrare che il fatto sia falso, e se non ci riesce conclude che è vero. Ma questo comportamento bizzarro non ci interessa qui.

Come si vede, mentre gino è scritto con l'iniziale minuscola, Nipote è scritto con l'iniziale maiuscola. Significa che Nipote è un segnaposto e che noi stiamo chiedendo a Prolog di sostituire a quel segnaposto l'individuo reale che soddisfa quella relazione.

Per essere un po' più precisi, noi abbiamo chiesto:

Chi è (o chi sono), se esistono, il (o i) nipoti di Gino?

Siccome nella nostra base di conoscenza c'è un solo figlio di Gino, e quindi un solo nipote di Mario, la riposta è sempre univoca: Mattia. Ma se avessimo aggiunto alle conoscenze dichiarate esplicitamente anche:

```
figlio(francesco,mario).
```

che sarebbe successo?

Niente: Prolog *non sarebbe stato in grado* di trovare il secondo figlio di Mario, e quindi si ostinerebbe a dire che Gino ha un solo nipote. Questo non è un errore ma il comportamento tipico che va capito bene: Prolog cerca la prima soluzione possibile per l'obiettivo che gli si è dato. Tuttavia, se NON riesce a soddisfarlo con il primo fatto che incontra, o con l'applicazione della prima regola, proverà con i fatti e le regole seguenti. Per questo, a differenza di altri linguaggi di programmazione, l'ordine con cui si inseriscono i fatti e le regole *è molto importante*: è il metodo principale con cui si controlla la maniera in cui Prolog procede con i suoi tentativi di deduzione.

Poco più avanti vederemo come intervenire in maniera un po' brusca per costringere Prolog a continuare la sua ricerca e a fornire tutte le soluzioni, non solo la prima.

Abbiamo visto che nonno e nipote sono relazioni inverse. Abbiamo visto come chiedere a Prolog notizie sui nipoti di Gino. E se volessimo sapere qualcosa sui nonni? Per esempio:

2b. Chi è il nonno di Mattia?

Beh, non abbiamo mai detto che si possa usare un segnaposto *solo* per il secondo argomento. Possiamo tranquillamente scrivere:

```
?nonno(Nonno,mattia).
```

E ricevere da Prolog tutti i termini che possono essere sostituiti a Nonno perché sia soddisfatta la relazione con mattia.

C'è una terza forma che è una specie di interrogazione di terzo grado:

3. Dimmi tutto quello che sai sui nonni e i nipoti.

```
?nonno(Nonno,Nipote).
```

Qui il centro è più chiaramente il predicato nonno, non un individuo particolare. Prolog cercherà tutti i termini che possono essere sostituiti a Nonno e Nipote e che soddisfano la relazione fornita.

Apparentemente, fornisce solo la prima:

```
?- nonno(Nonno,Nipote).
Nonno = gino,
Nipote = mattia
```

ma in realtà si ferma ad aspettare la nostra reazione. Se premiamo il tasto <invio>, la pianta lì; se invece premiamo <spazio> continua a cercare tutte le soluzioni possibili. Infatti ce ne sono altre due:

```
Nonno = gino,
Nipote = francesco ;
Nonno = franco,
Nipote = mario ;
```

Però per obbligare Prolog a restituire tutte le soluzioni, e non solo la prima, possiamo modificare la regola in questo modo:

```
nonno(Nonno,Nipote) :-
   padre(Padre,Nipote),
   padre(Nonno,Padre),
   write(Nonno),
```

```
        write(": "),
        write(Nipote),
        nl,
        fail.
```

Cosa abbiamo fatto? Abbiamo inserito al termine della regola il predicato `fail`. E' un predicato strano: intanto ha sempre successo, ma ha come effetto collaterale quello di impedire il soddisfacimento della regola di cui fa parte. Una specie di suicidio. Ma allora a che serve? Semplicemente a fare in modo che Prolog sia costretto a provare una soluzione diversa, che può voler dire il tentativo di assegnare termini diversi agli argomenti della regola, oppure la ricerca di una regola differente.

Con la regola modificata in questo modo, il risultato sarà:

```
?- nonno(Nonno,Nipote).
gino: mattia
gino: francesco
franco: mario
false.
```

Alternative

Ci accorgiamo (solo ora?) che abbiamo esplorato solo la linea maschile.
Possiamo rimediare aggiungendo ai fatti le seguenti righe:

```
madre(giovanna,mattia).
madre(giovanna,francesco).
madre(assunta, mario).
madre(erminia,assunta).
```

E la regola:

```
genitori(Madre, Padre, Figlio):-
  padre(Padre,Figlio),
  madre(Madre,Figlio).
```

Che ci permette di trovare, ad esempio, i genitori di Mattia.
Esiste però un termine generico (genitore) che può significare sia padre che madre. Come si può esprimere in Prolog un'alternativa? Con due regole.
Questa è una differenza grande rispetto ad altri modelli di programmazione, dove in generale non si possono creare due funzioni con lo stesso nome.
Tenendo in mente però come funziona Prolog, è del tutto sensato scrivere:

```
genitore(Genitore,Figlio):-
    padre(Genitore,Figlio).
genitore(Genitore,Figlio):-
    madre(Genitore,Figlio).
```

Queste regole vengono interpretate da Prolog nel modo seguente:
1. Se gli si chiede di dimostrare:

```
genitore(X,francesco)
```

Prolog cerca qualcuno che sia padre di francesco. Ma non esiste, tra i fatti, niente che permetta di asserirlo.
2. Allora prova la seconda regola, e cerca qualcuno che sia madre di francesco. Trova

```
madre(giovanna,francesco).
```

3. A questo punto può rispondere che giovanna è genitore di francesco.

In generale possiamo dire che Prolog prova tutte le regole con lo stesso predicato (e lo stesso numero di argomenti), nell'ordine in cui le trova, finché non ne trova una che sia vera.

Corsi e ricorsi

Ma se volessimo scrivere tutte le regole per rappresentare i rapporti (bisnonno, trisnonno, e poi bisnipote, trisnipote) come potremmo fare? Il problema qui non è più esprimere una relazione come regola, ma il fatto che non sappiamo a priori quanti gradi separano l'avo dal discendente. Quindi non è possibile scrivere esplicitamente tutte le regole.
La soluzione ha un bel nome: ricorsione.
La ricorsione è un'alternativa al ciclo degli altri modelli di programmazione. Consiste semplicemente nella possibilità di richiamare un predicato dall'interno dello stesso predicato. Sembra una cosa scema da fare, ma un senso ce l'ha. Probabilmente il modo più semplice di capirla è come sempre con un esempio.

```
avo(Avo,Discendente):-
   genitore(Avo,Discendente).

avo(Avo,Discendente):-
   genitore(Qualcuno,Discendente),
   avo(Avo,Qualcuno).
```

Ci sono due regole. Una dice:

> *un genitore è un avo del figlio.*

O meglio:

> *se c'è qualcuno (A) che è genitore di qualcun altro (B), allora quel qualcuno (A) è anche un avo di quel qualcun altro (B).*

Insomma, genitore è un tipo di avo. Bella scoperta.
L'altra dice:

> *qualcuno (A) è un avo di qualcun altro (B) se si trova un terzo individuo (C) che è genitore di B, e si può dimostrare che A è avo di C.*

Come si vede, la seconda regola è vera se è vera… la seconda regola. Meglio: la seconda regola è soddisfatta *con alcuni termini* se la seconda regola può essere soddisfatta *con altri termini.*
Nel corso del suo tentativo di dimostrare:

```
? avo(Avo,mattia)
```

il predicato avo viene richiamato ancora, finché non riesce a trovare qualcuno (D) che sia genitore di C: in quel caso si può applicare la prima regola, quella che parla solo di genitori. A quel punto, D sarà anche avo di B.
Per capire meglio cosa succede, si può modificare leggermente aggiungendo un altro argomento:

```
avo(Avo,Discendente,Grado):-
   genitore(Avo,Discendente),
   Grado is 1.

avo(Avo,Discendente,Grado):-
   genitore(Qualcuno,Discendente),
   avo(Avo,Qualcuno,M),
   Grado is M + 1.
```

Ogni volta che viene richiamato il predicato avo, si aumenta il grado di 1.
Incidentalmente, questi predicati calcolano i gradi di separazione tra parenti. Ad esempio, se si chiede:

```
?- avo(franco,mattia,Grado).
```

Prolog risponde con il grado di parentela:

```
Grado = 3 .
```

Però se si chiede semplicemente

```
?- avo(Chi,mattia,Grado).
```

il risultato è solo uno:

```
Chi = mario,
Grado = 3 .
```

che non è quello che volevamo. Per fare in modo che Prolog risalga, dopo aver trovato la prima soluzione, a tutte le altre, bisogna invertire l'ordine dei predicati (e questo è un buon esempio di quanto sia importante l'ordine di scrittura delle regole):

```
avo(Avo,Discendente,Grado):-
   genitore(Qualcuno,Discendente),
   write(Qualcuno),
   nl,
   avo(Avo,Qualcuno,M),
   Grado is M + 1.

avo(Avo,Discendente,Grado):-
   genitore(Avo,Discendente),
   Grado is 1.
```

La risposta in questo caso è più completa:

```
?- avo(Chi,mattia,Grado).
mario
gino
franco
Chi = franco,
Grado = 3 .
```

Liste

Per rappresentare un mondo di cui sappiamo tutto possiamo utilizzare delle forme rigide; per rappresentare un mondo di cui conosciamo solo una parte, o che è in divenire, bisogna inventarsi qualcosa di più flessibile. La lista è candidato ideale. Una lista è un insieme ordinato di elementi (anche vuoto) che può essere modificato aggiungendo e togliendo elementi.
Bisogna però dire che una lista, in Prolog come in altri linguaggi a partire dal Lisp che l'ha introdotta, non corrisponde semplicemente a una sequenza.
Una lista è una sequenza in cui il primo elemento punta al secondo, ma non sa nulla del resto. Il secondo punta al terzo, e non sa nulla del resto. E così via. Per questa sua natura poco strutturata, la lista viene trattate come composte di due parti: la testa e la coda. La coda è a sua volta una lista.
In Prolog le liste si scrivono circondate da parentesi quadre, con gli elementi separati da virgole. Ad esempio:

```
mesi([gennaio,febbraio,marzo,aprile,maggio,giugno,luglio,agosto,settembre,
ottobre,novembre,dicembre]).
```

Questo modo di scrivere assomiglia, ma non è equivalente a:

```
mese(gennaio).
mese(febbraio).
..
mese(dicembre).
```

a meno che non aggiungiamo delle regole apposite che mettono in corrispondenza questi fatti diversi.

La maniera di trattare le liste è piuttosto semplice e comune a molti linguaggi.[9] Come abbiamo detto, una lista è fatta da una testa e una coda. Prolog è in grado di separarle e di associarle a due segnaposti diversi, che si possono riutilizzare. Per esempio, per tornare ai nostri mesi, possiamo scrivere due regole usando la ricorsione:

```
mese(Mese,Listamesi):-
   [Mese|_] = mesi(Listamesi].
```

Ovvero: si predica mese di X se X è il primo elemento della lista dei mesi (come nel caso di gennaio).

```
mese(Mese,Listamesi):-
   [_|Altri]  = mesi(Listamesi),
   mese(Mese,Altri).
```

Ovvero: in alternativa, prova col resto della lista.

Le liste sono comode perché possono contenere altre liste. Per esempio:

```
stagioni([[dicembre,gennaio,febbraio],[marzo,aprile,maggio],
[giugno,luglio,agosto],[settembre,ottobre,novembre]]).
```

In questo caso, alla domanda:

```
? stagioni(Prima,_).
```

Prolog risponde in questo modo:

```
Prima = [dicembre, gennaio, febbraio].
```

9 Molte implementazioni di Prolog offrono diversi predicati comodi per trattare le liste come `append, member, reverse,select,delete.` Una lista completa dei predicati disponibili in SWI Prolog è qui http://www.swi-prolog.org/pldoc/man?section=lists

Perché non dice nulla riguardo al resto delle stagioni? Perché nella domanda abbiamo usato un segnaposto particolare, il trattino basso "_", che significa esattamente questo: non ci interessa nulla del resto.
Una regola per trovare il primo mese di ogni stagione potrebbe essere questa:

```
primo_mese([]).
primo_mese(Mese):-
    stagioni(Prima,Altre),
    [Mese|Altri] = Prima,
    write (Mese),
    nl,
    primo_mese(Altre).
```

Ovvero:

se una stagione è vuota, smetti di cercare: vuol dire che hai finito.
Se invece è composta di più mesi, scrivi il primo e poi continua con le altre stagioni.

Abbiamo introdotto surrettiziamente due predicati particolari che non sono molto dichiarativi e fanno storcere il naso ai logici: `write`, che scrive il suo argomento sullo schermo, e `nl` che scrive un ritorno a capo sullo schermo. Sono predicati sempre soddisfatti che non modificano la logica del programma, ma sono comodi per realizzare applicazioni che interagiscono un po' di più con gli utenti.

Le liste sono utile per trattare delle informazioni piuttosto libere, come le frasi della lingua. D'altra parte questo è proprio il dominio per il quale Colmerauer e Roussel pensavano che Prolog sarebbe stato più utile, e non è un caso che Prolog si trovi a suo agio nel farlo.
Si possono definire delle regole che rappresentano, ad esempio, la struttura di una frase affermativa, come questa:

```
frase_affermativa(Frase,Soggetto,Predicato,Complemento):-
   [Soggetto|Resto] = Frase,
   [Predicato|Complemento] = Resto.
```

Che dice semplicemente: una frase è affermativa se è composta da un soggetto, un predicato e un complemento. Possiamo usare questa regola per verificare se una frase particolare corrisponde alla struttura che abbiamo definito:

```
frase_affermativa([[il,cane,giallo],[[guarda],[il,pallone,rosso]]]).
```

Ma possiamo usare questa struttura anche per costruire frasi corrette. Creiamo una nuova base di conoscenza, composta da un vocabolario e una grammatica, come nell'esempio che segue:

```
/* Vocabolario */
articolo(il).
nome(cane).
nome(gatto).
nome(pallone).
verbo(guarda).
verbo(morde).

/* Grammatica */
soggetto(Soggetto,Articolo,Nome):-
   articolo(Articolo),
   nome(Nome),
   Soggetto = [Articolo,Nome].
predicato(Predicato,Verbo):-
   verbo(Verbo),
   Predicato = [Verbo].
```

```
complemento(Complemento,Articolo,Nome):-
   articolo(Articolo),
   nome(Nome),
   Complemento = [Articolo,Nome].
frase(Soggetto,Predicato,Complemento):-
   soggetto(Soggetto,Articolo,Nome1),
   predicato(Predicato,_),
   complemento(Complemento,Articolo,Nome2),
   Nome1 \= Nome2,
   append(Soggetto,Predicato,SP),
   append(SP,Complemento,SPC),
   atomic_list_concat(SPC," ",Frase),
   write(Frase),nl,
   fail.
```

Leggendo le regole può venire in mente che siano una specie di filastrocca che ripete la stessa cosa

Per esempio, che significa:

```
soggetto(Soggetto,Articolo,Nome):-
   articolo(Articolo),
   nome(Nome),
   Soggetto = [Articolo,Nome].
```

? Non è una ripetizione vuota dello stesso concetto per tre volte? No. La regola dice:

si può predicare "soggetto" di un'espressione composta da due parole se
la prima parola è un articolo,
la seconda è un nome.
Se tutto va bene, si può sostituire al segnaposto "Soggetto" la lista Articolo,Nome.

E' un controllo su Articolo e Nome (gli ultimi due argomenti del predicato "soggetto") che ha per effetto quello di legare al primo argomento (Soggetto) il risultato. Le altre regole sono costruite allo stesso modo, compresa l'ultima che dice che una frase deve essere composta da un soggetto, da un predicato e da un complemento. Qui abbiamo introdotto un vincolo: non vogliamo le frasi in cui lo stesso nome sia usato tanto nel soggetto che nel complemento.

```
Nome1 \= Nome2,
```

E abbiamo introdotto altri due predicati comodi: `append`, che crea una lista a partire da due altre, e `atomic_list_concat` che crea una stringa da una lista.

Ora possiamo chiedere a Prolog di mostrare tutte le frasi affermative che è possibile scrivere con il nostro vocabolario e la nostra grammatica:

```
 ? frase(Soggetto,Predicato,Complemento).
```

Che ci restituisce 12 frasi diverse, non tutte di comprensione immediata:

```
il cane guarda il gatto
il cane guarda il pallone
il cane morde il gatto
il cane morde il pallone
il gatto guarda il cane
il gatto guarda il pallone
il gatto morde il cane
il gatto morde il pallone
il pallone guarda il cane
il pallone guarda il gatto
il pallone morde il cane
il pallone morde il gatto
```

Certo manca un po' di varietà. Ma si può aggiungere facilmente. Cominciamo con quattro nuovi predicati:

```
aggettivo(minaccioso).
aggettivo(perplesso).
avverbio(insistentemente).
avverbio("con nonchalance").
```

Per poterli utilizzare, dobbiamo aggiungere un nuovo tipo di soggetto/4, che prevede anche la possibilità di un aggettivo:

```
soggetto(Soggetto,Articolo,Nome,Aggettivo):-
  articolo(Articolo),
  nome(Nome),
  aggettivo(Aggettivo),
  Soggetto = [Articolo,Nome,Aggettivo].
```

Lo stesso per complemento/4:

```
complemento(Complemento,Articolo,Nome,Aggettivo):-
  articolo(Articolo),
  nome(Nome),
  aggettivo(Aggettivo),
  Complemento = [Articolo,Nome,Aggettivo].
```

E infine aggiungiamo predicato/2, che permette l'uso di avverbi:

```
predicato(Predicato,Verbo,Avverbio):-
  verbo(Verbo),
  avverbio(Avverbio),
  Predicato = [Verbo,Avverbio].
```

Cosa manca? Una nuova forma di frase, più complessa della precedente, che prevede l'utilizzo di aggettivi e di avverbi. La scriviamo per ultima:

```
frase(Soggetto,Predicato,Complemento):-
  soggetto(Soggetto,Articolo,Nome1,Aggettivo1),
  predicato(Predicato,_,_),
  complemento(Complemento,Articolo,Nome2,Aggettivo2),
  Nome1 \= Nome2,
  Aggettivo1 \= Aggettivo2,
  append(Soggetto,Predicato,SP),
  append(SP,Complemento,SPC),
  atomic_list_concat(SPC," ",Frase),
  write(Frase),nl,
  fail.
```

Volendo rendere il codice un po' più ordinato, si potrebbe separare la parte di costruzione della frase da quella di interazione con l'utente. Ovvero:

```
frase(Soggetto,Predicato,Complemento):-
  soggetto(Soggetto,Articolo,Nome1,Aggettivo1),
  predicato(Predicato,_,_),
  complemento(Complemento,Articolo,Nome2,Aggettivo2),
  Nome1 \= Nome2.

scrivi_frase:-
  frase(Soggetto,Predicato,Complemento),
```

```
append(Soggetto,Predicato,SP),
append(SP,Complemento,SPC),
atomic_list_concat(SPC," ",Frase),
write(Frase),nl,
fail.
```

A questo punto abbiamo un generatore di frasi corrette sufficientemente vario e, per lo meno per i miei gusti, abbastanza divertente. E' in grado di produrre piccole istantanee come questa:

```
il gatto perplesso guarda insistentemente il pallone minaccioso
```

o questa:

```
il cane minaccioso morde con nonchalance il gatto perplesso
```

Come si diceva nei manuali scientifici di una volta, si lascia al lettore la prova e, se ne ha voglia, lo sviluppo ulteriore.[10]

Ci sono altre maniera per rappresentare e costruire frasi. Probabilmente la più semplice è quella che fa uso di un simbolismo che si chiama Grammatica a Clausole Definite (o Definite Clause Grammar).[11]

```
frase --> s_oggetto, predicato.
s_oggetto --> articolo, nome.
predicato --> verbo, s_oggetto.
articolo --> [il].
articolo --> [un].
nome --> [gatto].
nome --> [topo].
verbo --> [mangia].
```

Questo modo di scrivere è alternativo a quello che abbiamo usato finora, più semplice da capire, ma ugualmente funzionante in Prolog. Si può quindi chiedere:

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

per ottenere tutte le frasi generabili con questa grammatica, oppure verificare che

```
frase([il,gatto,mangia,il,topo],[]).
```

sia una frase corretta.

10 Qui trovate una trattazione più completa del tema: http://www.ce.unipr.it/research/HYPERPROLOG/natlan1.html#AnalisiSintProl

11 https://en.wikipedia.org/wiki/Definite_clause_grammar

Apprendimento

Per apprendimento intendiamo qui semplicemente il processo per cui un programma Prolog può aggiungere (o togliere) dei fatti o delle regole durante il suo utilizzo. Ancora una volta, nessun'intelligenza magica.

Intanto possiamo dare una buona notizia: a differenza del calcolo dei predicati di primo ordine (il formalismo logico da cui Prolog è ispirato), in Prolog si possono usare predicati che hanno per argomento altri predicati.

Non è una funzionalità banale se ci si pensa un attimo. Nel mondo fisico siamo abituati a distinguere tra strumenti e cose (per esempio, il martello e il chiodo) o tra azioni e oggetti dell'azione (per esempio, il cane guarda il pallone). La grammatica della lingua Italiana contiene cablata in sé questa distinzione con la differenza tra verbi e sostantivi; nei linguaggi di programmazione più semplici, questa distinzione è affidata alla differenza tra strutture di controllo e istruzioni semplici.

La possibilità di applicare uno strumento ad un altro strumento, o un'azione ad un'altra azione, è fondamentale per attività complesse come la manutenzione, il giudizio o, appunto, l'apprendimento. In maniera del tutto simile, avere anche in un linguaggio artificiale la possibilità di creare nuove funzioni o predicati che si possano applicare ad altre funzioni o predicati apre ad attività più complesse.

In Prolog, ad esempio, è presente un metapredicato (cioè un predicato che sia applica ad altri predicati) che si chiama `call/1` e si limita a definire come obiettivo il suo argomento. Potremmo scrivere, ad esempio,

```
call(scrivi_frase).
```

e otterremmo lo stesso risultato di:

```
scrivi_frase.
```

Allora a che serve?

Forse è più chiara l'utilità di un predicato come `findall/3`, che permette di trovare tutte le soluzioni ad un obiettivo e di raccoglierle in una lista:

```
findall(X,genitore(X,Y),Lista_genitori).
```

Oppure di due varianti, `bagof/3` e `setof/3`:

```
bagof(X,Y^genitore(X,Y),Lista_genitori).
```

```
setof(X,Y^genitore(X,Y),Lista_genitori).
```

In entrambi i casi, abbiamo specificato con il simbolo ^ messo davanti all'obiettivo da dimostrare che non ci interessa che venga restituito il suo valore (ci interessano solo i genitori, non i figli).

`setof/3` ha in più la caratteristica di buttare via le eventuali ripetizioni dalla lista dei risultati, il che può essere comodo.

Arriviamo alla questione dell'apprendimento. Il predicato:

```
assertz(predicato(Argomento)).
```

aggiunge predicato(argomento) alla fine della lista di ciò che è noto. Allo stesso modo,

```
retract(predicato(Argomento)).
```

elimina quel predicato. Il risultato di un'operazione di modifica della lista di predicati noti lo si può vedere con il predicato (indovinate quale?)

```
listing.
```

Ad esempio, se non avessimo inserito fin dall'inizio i fatti relativi alla relazione figlio-genitore, avremmo potuto farlo (per così dire *in diretta)* così:

```
assertz(figlio(mattia,mario)).
assertz(figlio(francesco,mario)).
```

Ci si domanderà quale può essere mai il vantaggio di questo metodo sul precedente, che sembra (ed è) più semplice e veloce. La differenza, come nel caso di call/1, è che questi fatti possono essere aggiunti solo a certe condizioni, sulla base di un'interazione con l'utente o col mondo esterno. Per esempio, partendo da una lista delle coppie figlio-padre (caricata da un file esterno, o richiesta all'utente)

```
[[francesco,mario],[mattia,mario],[mario,gino],[gino,franco]]
```

si può scrivere una regola che aggiunge un fatto per ogni coppia:

```
trova_figli(Figli):-
 forall(member([Figlio,Genitore|_],Figli),
          assertz(figlio(Figlio,Genitore))).
```

Che va interpretato così:

> *per ogni elemento della lista Figli (che è a sua volta una lista), aggiungi un predicato figlio con argomenti gli elementi della lista.*

Altre piste

Non ci sono molti testi introduttivi in Italiano.
Questo è uno, dell'Università di Parma
http://www.ce.unipr.it/research/HYPERPROLOG/prolog.html

E questo dell'Università di Milano
http://homes.di.unimi.it/~logica/logimat/Prolog/Furlan_Lanzarone_PROLOG.pdf

Ma sono entrambi piuttosto tecnici e dedicati a studenti di Informatica. Invece questo:

http://www.cs.unibo.it/~riccucci/Teaching%20Material/Scienze%20della%20Formazione/Guida%20SWI-Prolog1.pdf

descrive come si installa e si comincia a usare SWI-Prolog ed è rivolto agli studenti di Scienze della Formazione dell'Università di Bologna

Un tutorial molto completo (con esercitazioni), sempre a cura dell'Università di Parma, è questo:

http://www.ce.unipr.it/research/HYPERPROLOG/indprolog.html

Uno libro che ho trovato molto utile, anche se un po' datato dal punto di vista strettamente informatico e poco attraente da quello didattico, è

G.Casadei, P.Cuppini, A. Palareti, Informatica per le discipline umanistiche. Applicazioni didattiche del Prolog, Zanichelli, 1989.

Si, avete letto bene la data. Si parla di DOS e di WordStar, per chi sa cosa significa. Però è pieno di esempi per i docenti di Italiano, Geografia etc. Solo che è quasi impossibile da trovare.
Come quest'altro:

V.Midoro, Il filo di Arianna - Introduzione al Prolog, Torino, SEI, 1987.

Una serie di spunti interessanti, applicabili alla didattica, sono contenuti in questa pagina

http://www.funzioniobiettivo.it/glossadid/Caviglia/didattica.htm

a cura di Francesco Caviglia.