Linguaggi di Programmazione

Marco Natali

INDICE

1 INTRODUZIONE AI PARADIGMI DI PROGRAMMAZIONE

Nel corso di Linguaggi di Programmazione ci occupiamo di 3 importanti paradigmi di linguaggi quali i linguaggi logici, linguaggi funzionali e linguaggi predicativi, la cui conoscenza è importante per decidere quale paradigma usare per risolvere un particolare problema e permette di migliorare la propria competenza nell'ambito della programmazione per scrivere programmi il più possibile efficienti.

Durante il corso, ma soprattutto durante la vita futura lavorativa, si dovrebbero rispettare le regole standard di rappresentazione e formattazione dei programmi quali ad esempio lo standard Google, regole GNU, per cui nei programmi sviluppati in questo corso devono rispettare le seguenti convenzioni:

- "Usare Emacs come editor in quanto indenta lui al meglio" cit Antoniotti
- le linee di codice non devono essere più lunghe di 80 colonne per riga
- inserire spazi tra gli operatori e uno spazio dopo virgola e punto e virgola ossia **2 + 3** e **(4, 5)**.
- non inserire uno spazio tra il nome di una funzione/predicato e le parentesi ossia **foo(4)**.
- inserire uno spazio tra un istruzione di controllo/operatori logici e le parentesi ossia **if (4 || 5)**
- non usare i commenti all'interno di un box perchè difficilmente mantenibile

I linguaggi di programmazione vengono classificati in base a quale paradigma implementano per la computazione, ossia il modello di riferimento usato per strutturare un programma, al fine di facilitare l'apprendimento di linguaggi similari:

- **Imperative Languages**: basati sull'architettura di Von Neumann, utilizzata da tutti i calcolatori moderni, in cui il processore ha il compito di leggere e scrivere le celle di memoria durante l'esecuzione della computazione.
  Questo paradigma utilizza uno stile prescrittivo, ossia i programmi iterativi specificano una sequenza di istruzioni da eseguire per modificare lo stato del sistema e questo flusso può essere modificato soltanto con le strutture di controllo.
  Ne fanno parte il C/C++, i linguaggi Assembler, Pascal, Python, ...
- **Logic Languages**: paradigma in cui un programma è una deduzione logica, ossia ogni istruzione è una formula del linguaggio e la computazione consiste nell'interrogare il sistema per sapere se l'interrogazione fa parte della conoscenza rappresentata dal programma.

  Questo paradigma viene utilizzato principalmente per la dimostrazione della correttezza di un programma, per rappresentare i database ed infine sta avendo un notevole utilizzo ultimamente nel campo dell'AI(Artificial Intelligence).
  Fa parte di questo paradigma principalmente solo il Prolog e i suoi derivati, infatti noi useremo il SWI-Prolog.
- **Functional Languages**: paradigma in cui il concetto di funzione è l'unica cosa importante infatti i programmi consistono nell'applicazione e nella definizione di una serie di funzioni.
  Ne fanno parte i linguaggi Lisp, Common Lisp, Scheme, Javascript, ML, Ocaml, Haskell, R e pochi altri.

Ognuno dei seguenti paradigmi può essere ad oggetti, come ad esempio Java, C++, Common Lisp, Python, infatti il paradigma ad oggetti è ortogonale alla definizione data da noi delle diverse tipologie dei linguaggi, anche se presume alcune caratteristiche tipiche dei linguaggi imperativi.

1.1 PARADIGMA IMPERATIVO

Il paradigma imperativo, come già visto, si occupa di eseguire una sequenza di istruzioni infatti fu inventato per la computazione "numerica", a differenza degli altri paradigmi che si concentrano sulla computazione simbolica.

Un programma imperativo è composto da due componenti, uno per rappresentare le strutture dati e l'altro per gli algoritmi:

- dichiarazione in cui vengono dichiarate le variabili, i tipi e le funzioni necessarie per la computazione, ossia vengono stabilite le strutture dati necessarie per l'implementazione del programma.
- definizione in cui si implementatno gli algoritmi, per eseguire correttamente la computazione, utilizzando le istruzioni del linguaggio.

Il paradigma imperativo ha spopolato fino agli anni '70/80 quando venne spodestato dal paradigma ad Oggetti e noi lo abbiamo già affrontato durante il corso Programmazione per cui in sto corso vedremo soltanto alcune peculiarità del linguaggio C, come ad esempio la gestione manuale della memoria.

La necessità di usare le applicazione ad un maggiore livello di astrazione e di scrivere programmi il più concisi possibile, ha spinto alla creazione di nuovi paradigmi, come quello logico che analizzeremo ora.

1.2 PARADIGMA LOGICO

Il paradigma logico non è più basato sull'architettura di Von Neumann ma sulla logica matematica, anche se quando viene implementato sui calcolatori alcune caratteristiche sono derivate dal paradigma imperativo, ed utilizza uno stile di programmazione descrittivo, ossia viene definito cosa fa parte del problema da rappresentare.

Inoltre nella programmazione logica non è presente alcuna separazione netta tra gli algoritmi e le strutture dati e, come già visto nella definizione dei vari paradigmi, un programma consiste nel descrivere un problema come una serie di sentenze del linguaggio ed interrogare il sistema, il quale effettua una deduzione sulla base della conoscenza rappresentata.

Un esempio di programma logico è il seguente:

```
%%%% -* mode:Prolog -*-

colleague(X, Y):-
    worksFor(X, Z),
    worksFor(Y, Z),
    X \= Y. %Utilizzato per dire che X != Y

worksFor(ugo, ibm).
worksFor(gino, ibm).
worksFor(enrica, samsung).
worksFor(salvo, olivetti).
worksFor(ciro, samsung).
```

```
worksForAnEvilCompany(X, Y) :- worksFor(X, trenord).

:- colleague(X, Y).
%%%% End of file employees.pl
```

Questo programma implementa la regolazione di lavorare per una compagnia e se due persone sono colleghi, anche se una trattazione completa della semantica verrà fornita nei successivi paragrafi.

1.3 PARADIGMA FUNZIONALE

Il paradigma funzionale, come già notato nel paradigma logico, viene usato per una programmazione simbolica, in cui si adotta solitamente uno stile descrittivo ed infine non vi è una completa separazione tra strutture dati ed algoritmi.

Il concetto fondamentale di questo paradigma è la funzione, relazione definita su due insiemi che associa ad un elemento del dominio uno e un solo elemento del codominio.

Dopo che sono state definite le funzioni necessarie per la computazione, possiamo applicarle sugli elementi del dominio per ottenere un valutazione delle funzioni e ciò è cosa si definisce per computazione nel paradigma funzionale.
In un linguaggio funzionale puro la valutazione viene determinata soltanto dalla funzione e non da i valori in memoria e ciò permette di definire una variabile come una costante matematica, in cui il valore non è mutabile, cosa differente rispetto alle variabili nei linguaggi imperativi che sono una astrazione di una locazione in memoria.

Esempio di programma funzionale:

1.4 RICHIAMI SUGLI AMBIENTI RUNTIME

Per eseguire un programma in un qualsiasi linguaggio il sistema operativo deve mettere a disposizione un ambiente *run time*, anche una macchina virtuale, il quale fornisce almeno due funzionalità:

- mantenimento dello stato della computazione attraverso program counter, limiti di memoria ed etc...
- gestione della memoria disponibile, fisica e virtuale, il quale avviene usando due aree distinte con funzioni diverse:
  - lo **Stack** serve per la gestione delle chiamate, soprattutto ricorsive, a procedure attraverso i record di attivazione.
  - lo **Heap** serve per la gestione di strutture dati dinamiche, quali liste, alberi, dizionari ed etc ...

I linguaggi logici e funzionali (ma anche Java) utilizzano pesantemente lo Heap dato che forniscono come strutture dati built-in liste e spesso vettori di dimensione variabile.

La gestione dei record di attivazione è stata affrontata nei corsi di Programmazione ed Architettura, a cui si può consultare libri e gli appunti per rinfrescarne la memoria.

La gestione della memoria, in particolare quella dinamica, può avvenire in maniera automatica, attraverso il *Garbage Collector* usato da Python, Java, Lisp, Prolog ed altri, oppure in maniera manuale, come in C/C++ attraverso i comandi malloc(new) e free(delete).

2 | LOGICA

Effettuiamo ora un ripasso della logica proposizionale e predicativa, affrontata nel corso Fondamenti dell'Informatica, al fine di rivedere e perfezionare i concetti necessari per la comprensione e la scrittura di programmi logici.

Partiamo con l'esempio di una semplice dimostrazione geometrica:

Th. *Dato un triangolo isoscele (con $\overline{AB} = \overline{BC}$) si ha che $\angle A$ e $\angle C$, ovvero l'angolo in C, sono uguali.*

Dimostrazione. Si comincia la dimostrazione con l'elenco delle conoscenze pregresse:

1. se due triangoli sono uguali essi hanno lati e angoli uguali
2. se due triangoli hanno due lati e l'angolo sotteso uguale allora i due triangoli sono uguali
3. se viene definita la bisettrice di $\angle B$, $\overline{BH}$, si ha che $\angle ABH = \angle HBC$

Procediamo ora coi passi della dimostrazione:

1. $\overline{AB} = \overline{BC}$ per ipotesi
2. $\angle ABH = \angle HBC$ per la terza conoscenza pregressa
3. $HBC = ABH$ per la seconda conoscenza pregressa in quanto due lati sono uguali per ipotesi e dal passo precedente l'angolo sotteso ai due lati uguali è uguale in ambedue i triangoli.
4. $\angle A = \angle C$ per la prima conoscenza pregressa dato che $HBC = ABH$

Siamo così giunti alla fine della dimostrazione ma adesso vogliamo rappresentarla attraverso gli strumenti della logica al fine di renderla totalmente formale per cui si effettua i seguenti passaggi:

- si è trasformata la seconda conoscenza pregressa in:
  se $\overline{AB} = \overline{BC}$ **e** $\overline{BH} = \overline{BH}$ **e** $\angle ABH = \angle HBC$ **allora** $ABH = HBC$
- si è trasformata la prima conoscenza pregressa in:
  se $ABH = HBC$ **allora** $\overline{AB} = \overline{BC}$ **e** $\overline{BH} = \overline{BH}$ **e** $\overline{AH} = \overline{HC}$ **e** $\angle ABH = \angle HBC$ **e** $\angle AHB = \angle CBH$ **e** $\angle A = \angle C$

Possiamo ora procedere col processo di formalizzazione, ossia il processo che ci permette di affermare $\overline{AB} = \overline{BC} \vdash \angle A = \angle C$ con $\vdash$ simbolo di derivazione logica.

Assumendo $P = \{\overline{AB} = \overline{BC}, \angle ABH = \angle HBC, \overline{BH} = \overline{BH}\}$ ed avendo le seguenti conoscenze pregresse:

Figura 1: Triangolo Isoscele

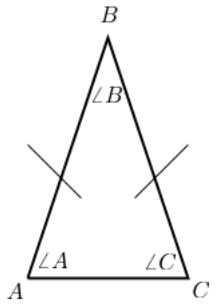
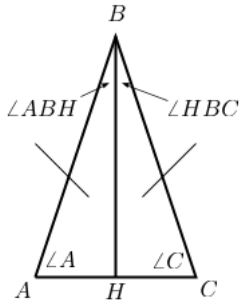


Figura 2: Triangolo isoscele scomposto in due triangoli rettangoli



1. $\overline{AB} = \overline{BC} \wedge \overline{BH} = \overline{BH} \wedge \angle ABH = \angle HBC \rightarrow ABH = HBC$
2. $ABH = HBC \rightarrow \overline{AB} = \overline{BC} \wedge \overline{BH} = \overline{BH} \wedge \overline{AH} = \overline{HC} \wedge$
   $\angle ABH = \angle HBC \wedge \angle AHB = \angle CBH \wedge \angle A = \angle C.$

per ottenere $\overline{AB} = \overline{BC} \vdash \angle A = \angle C$ bisogna effettuare la seguente catena:

P1 $\overline{AB} = \overline{BC}$ da **P**

P2 $\angle ABH = \angle HBC$ da **P**

P3 $\overline{BH} = \overline{BH}$ da **P**

P4 $\overline{AB} = \overline{BC} \wedge \overline{BH} = \overline{BH} \wedge \angle ABH = \angle HBC$ da **P1, P2, P3** attraverso l' **introduzione della congiunzione**.

P5 $ABH = HBC$ da **P4**, dalla **regola 2** attraverso l'applicazione del **modus ponens**.

P6 $\overline{AB} = \overline{BC} \wedge \overline{BH} = \overline{BH} \wedge \overline{AH} = \overline{HC} \wedge$
$\angle ABH = \angle HBC \wedge \angle AHB = \angle CBH \wedge \angle A = \angle C$ da **P5**, dalla **regola 1** attraverso Modus Ponens.

P7 $\angle A = \angle C$ da **P6** attraverso **eliminazione della congiunzione**

□

Defi. *Una dimostrazione, chiamata **dim**, indicata con $S \vdash F$, è un sequenza:*

$$dim =< P_1, P_2, ..., P_n >$$

con:

- $P_n = F$
- $P_i \in S$ *o con* P_i *ottenibile dalle* $P_1, ..., P_{i-1}$ *applicando una regola di inferenza*

Un insieme di regole di inferenza costituisce la base di un calcolo logico, il quale ha lo scopo di manipolare le formule in modo unicamente sintattico al fine di stabilire una connessione tra un insieme di formule di partenza, dette assiomi e un insieme di conclusioni.

2.1 LOGICA PROPORZIONALE

La logica proposizionale si occupa delle conclusioni che si possono trarre da un insieme di proposizioni, ma purtroppo è un linguaggio limitato in quanto non si può generalizzare le proposizioni e/o definire delle proprietà.

La sintassi di un linguaggio è composta da una serie di formule ben formate(FBF) definite induttivamente nel seguente modo:

A	B	$A \wedge B$	$A \vee B$	$\neg A$	$A \rightarrow B$
F	F	F	F	T	T
F	T	F	T	T	T
T	F	F	T	F	F
T	T	T	T	F	T

1. Le costanti e le variabili proposizionali sono FBF.
2. Se A e B sono FBF allora $(A \wedge B)$,$(A \vee B)$,$(\neg A)$,$(A \rightarrow B)$, TA e FA sono delle formule ben formate.
3. nient'altro è una formula

Esempio:
$(P \wedge Q) \in Fbf$ è una formula ben formata
$(PQ \wedge R) \notin Fbf$ in quanto non si rispetta la sintassi del linguaggio definita.

La semantica di una logica consente di dare un significato e un interpretazione alle formule del Linguaggio.

Defi. *Sia data una formula proposizionale P e sia $P_1, \ldots, P_n$, l'insieme degli atomi che compaiono nella formula A.*
Si definisce come interpretazione *una funzione $v : \{P_1, \ldots, P_n\} \mapsto \{T, F\}$ che attribuisce un valore di verità a ciascun atomo della formula A.*

I connettivi della Logica Proposizionale hanno i seguenti valori di verità: La tavola di verità costituisce la semantica di un insieme di proposizioni mentre un calcolo logico dice come generare nuove formule logiche, ovvero espressioni sintattiche, a partire dagli assiomi e questo processo di generazione si chiama dimostrazione.

Per ottenere nuove formule dagli assiomi si usa il calcolo proposizionale, che si base sulle regole di inferenza, ossia regole attraverso cui si può derivare una nuova formula ben formata.
Le regole di inferenza analizzate sono le seguenti:

- **Modus Ponens:**
$$\frac{a \rightarrow b,\ a}{b}$$
- **Modus Tollens:**
$$\frac{a \rightarrow b, \neg b}{\neg a}$$
- **Eliminazione $\wedge$:**
$$\frac{P_1 \wedge P_2 \wedge ... \wedge P_n}{P_i}$$
- **Introduzione di $\wedge$:**
$$\frac{P_1, P_2, ..., P_n}{P_1 \wedge P_2 \wedge ... \wedge P_n}$$
- **Introduzione di $\vee$:**
$$\frac{a}{a \vee b}$$
- **Terzo Escluso:**
$$\frac{a \vee \neg a}{vero}$$
- **Eliminazione di $\neg$:**
$$\frac{\neg\neg a}{a}$$

- **Eliminazione di** $\wedge$:
$$\frac{a \wedge vero}{a}$$
- **Contraddizione:**
$$\frac{a \wedge \neg a}{b}$$
ovvero da una contraddizione posso trarre qualsiasi conseguenza

Queste regole di inferenza fanno parte del calcolo naturale, detto anche di Gentzen, similare al calcolo tramite Tableaux visto nel corso di Fondamenti dell'informatica. Questo tipo di calcolo consiste nel formalizzare i modi di derivare conclusioni a partire dalle premesse, ovvero di derivare direttamente un FBF mediante una sequenza di passi ben codificati.
La regola del modus ponens insieme al principio del terzo escluso, posso essere usati anche procedendo per assurdo alla dimostrazione di una data formula e ciò viene detto *principio di risoluzione*, affrontata poi quando analizziamo il linguaggio Prolog.

Una formula nella logica proposizionale può essere di 4 diversi tipi:

TAUTOLOGICA la formula è soddisfatta da qualsiasi valutazione della formula.

SODDISFACIBILE NON TAUTOLOGICA la formula è soddisfatta da qualche valutazione.

FALSIBICABILE la formula non è soddisfatta da qualche valutazione della formula.

CONTRADDIZIONE la formula non viene mai soddisfatta.

La logica proposizionale è decidibile, ossia posso sempre verificare il significato di una formula, infatti esiste una procedura effettiva che stabilisce la validità o no di una formula, o se questa ad esempio è una tautologia.
In particolare il verificare se una proposizione è tautologica o meno è l'operazione di decibilità principale che si svolge nel calcolo proposizonale.

Una dimostrazione di una formula di una logica può venire tramite:

- **Metodo diretto**: Data un'ipotesi, attraverso una serie di passi si riesce a dimostrare la correttezza della Tesi
- **Metodo per assurdo**(non sempre accettato in tutte le logiche): Si nega la tesi ed attraverso una serie di passi si riesce a dimostrare la negazione delle ipotesi.

La logica proposizionale è completa e corretta per cui data una formula ben formata, sia utilizzando i sistemi deduttivi sia analizzando la semantica, si è in grado di stabilire se una formula è tautologica, soddisfacibile o falsificabile.

2.2 LOGICA DEL PRIMO ORDINE

La logica del primo ordine, chiamata anche logica predicativa, permette di quantificare i vari fatti ed introduce il concetto di funzione e predicato per poter esprimere delle proprietà su una serie di individui.

Un linguaggio predicativo L è composto dai seguenti insiemi di simboli:

1. insieme di variabili individuali(infiniti) $x, y, z, \ldots$
2. connettivi logici: $\wedge \vee \neg \rightarrow \Longleftrightarrow$
3. quantificatori: $\forall \exists$
4. simboli: (,)
5. Costanti proposizionali: T, F

6. simbolo di uguaglianza $=$, eventualmente assente

Questa è la parte del linguaggio tipica di ogni linguaggio del primo ordine poi ogni linguaggio definisce la propria segnatura:

1. insiemi di simboli di costante $a, b, c, \ldots$
2. simboli di funzione con arieta $f, g, h, \ldots$
3. simboli di predicato $P, Q, Z, \ldots$ con arietà

Esempio:Linguaggio della teoria degli insiemi
Costante:$\emptyset$
Predicati:$\in (x, y)$, $= (x, y)$
Esempio:Linguaggio della teoria dei Numeri
Costante:0
Predicati:$< (x, y)$,$= (x, y)$
Funzioni:$succ(x)$,$+(x, y)$,$*(x, y)$

Per definire le formule ben formate della logica predicativa bisogna prima definire l'insieme di termini e le formule atomiche.

Defi. *L'insieme TERM dei termini è definito induttivamente come segue*

1. *Ogni variabile e costante è un termine*
2. *Se $t_1 \ldots t_n$ sono dei termini e f è un simbolo di funzione di arietà n allora $f(t_1, \ldots, t_n)$ è un termine*

Defi. *L'insieme ATOM delle formule atomiche è definito come:*

1. *T e F sono degli atomi*
2. *Se t_1 e t_2 sono dei termini, allora $t_1 = t_2$ è un atomo*
3. *Se $t_1, \ldots, t_n$ sono dei termini e P è un predicato a n argomenti, allora $P(t_1, \ldots, t_n)$ è un atomo.*

Defi. *L'insieme delle formule ben formate(FBF) di L è definito induttivamente come*

1. *Ogni atomo è una formula*
2. *Se $A, B \in FBF$, allora $\neg A$, $A \wedge B$,$A \vee B$,$A \to B$ e $A \iff B$ appartengono alle formule ben formate*
3. *Se $A \in FBF$ e x è una variabile, allora $\forall x A$ e $\exists x A$ appartengono alle formule ben formate*
4. *Nient'altro è una formula*

Defi. *L'insieme $var(t)$ delle variabili di un termine t è definito come segue:*

- $var(t) = \{t\}$, *se t è una variabile*
- $var(t) = \emptyset$ *se t è una costante*
- $var(f(t_1, \ldots, t_n)) = \bigcup_{i=1}^{n} var(t_i)$
- $var(R(t_1, \ldots, t_n)) = \bigcup_{i=1}^{n} var(t_i)$

Si definisce come *aperto* un termine che non contiene variabili altrimenti il termine è *chiuso*.
Le variabili nei termini e nelle formule atomiche possono essere soltanto libere in quanto gli unici operatori che "legano" le variabili sono i quantificatori.

Il campo di azione dei quantificatori si riferisce soltanto alla parte in cui si applica il quantificatore per cui una variabile si dice *libera* se non ricade nel campo di azione di un quantificatore altrimenti è *vincolata*.

Si aggiunge una nuova regola d'inferenza per la logica dei predicati, l'eliminazione del quantificatore universale $\forall$:

$$\frac{\forall x, T(..., x, ...), c \in C}{T(..., c, ...)}$$

Abbiamo altre regole di inferenza per il quantificatore esistenziale:

- **Introduzione del quantificatore esistenziale** $\exists$:

$$\frac{T(..., c, ...), c \in C}{\exists x, T(..., x, ...)}$$

- si hanno le seguente identità:

$$\exists x, \neg T(..., x, ...) \equiv \neg \forall x, T)..., x, ...)$$
$$\forall x, \neg T(..., x, ...) \equiv \neg \exists x, T(..., x, ...)$$

Per una trattazione migliore e per approfondimenti sulla logica si può consultare gli appunti del corso di Fondamenti oppure i vari libri di logica, come ad esempio "How to prove it".

3 PROGRAMMING IN LOGIC

Dopo aver effettuato un ripasso della logica, incominciamo a considerare il Prolog e la programmazione logica: le basi sono state poste da Robert Kowalski e Marten Van Emdem, mentre la progettazione e implementazione, avvenne nel 1972 a Marsiglia grazie ad Alain Colmerauer e Philippe Roussel.

Il Prolog è un linguaggio di programmazione logica basato sulle clausole di Horn, la cui definizione sarà data in seguito, e la procedura utilizzata dal Prolog per la computazione è il principio di risoluzione e di unificazione, anch'esso trattato in seguito.

Un programma logico è formato da un insieme di istruzioni, rappresentanti un sottoinsieme di frasi ben formate della logica del primo ordine e l'ambiente Prolog determina se una data assunzione è verificata o meno nel programma e sotto quali eventuali vincoli.

I componenti basilari di un programma Prolog, rappresentanti tutti una clausola di Horn, sono:

FATTI : indica una relazione esistente tra due oggetti, necessaria per stabilire la base di conoscenza del linguaggio.

```
worksFor(paolo, coop).
```

QUERY : chiede al sistema l'esistenza di una relazione tra gli oggetti e quindi inizia una deduzione per stabilire se la query è una conseguenza diretta del programma, dopo l'applicazione del principio di risoluzione per un numero finito di volte.

```
:- worksFor(paolo, trenord).
```

REGOLE : definisce una nuova relazione esistente tra gli oggetti, ossia permette di derivare una nuova conclusione dalla base di conoscenza.

```
parent(X, Y) :- father(X, Y).
```

La testa della regola A, viene detta conseguenza mentre il corpo $B_1, B_2, \ldots, B_n$ sono l'antecedente e il simbolo :- indica il simbolo logico di implicazione.
Una relazione può essere definita ricorsiva, per cui necessita di almeno due regole, una per il caso base e una per il caso passo, come il seguente esempio:

```
natural_number(0).
natural_number(s(X)) :- natural_number(X).%s(X) indica il successore di X
```

I fatti e le regole sono quantificate universalmente mentre una query si intende sempre quantificata esistenzialmente, per cui una query risponde true se esiste un instanza α che la verifica altrimenti risponde false.

Le query e le regole le abbiamo definite nella forma generale, ossia possiamo definire regole e query su congiunzioni di termini, infatti il simbolo **","** rappresenta l'operatore logico and, perciò in caso più termini hanno lo stesso simbolo di variabile l'instanza α deve essere la stessa per stabilire se una query è una conseguenza diretta del programma.

Per vedere al meglio il concetto di implicazione vediamo la definizione del concetto di nonno nel linguaggio Prolog:

```
grandfather(X, Y) :- father(X, Z), parent(Z, Y).
```

Questa relazione stabilisce che X è il nonno di Y se risulta che esiste un Z tale che X è padre di Z e Z è genitore di Y.

Ogni espressione prolog viene chiamata *termine*, che può essere della seguente tipologia:

ATOMI : elemento base del Prolog rappresentato da un numero, qualsiasi sequenza racchiusa tra ' ', una sequenza di caratteri alfanumerici, con la lettera minuscola iniziale, e nel SWI-Prolog anche una stringa indica un atomo.

VARIABILI : sequenza di caratteri alfanumerici, iniziata con la lettera maiuscola oppure con _ e vengono inizializzate quando il sistema prova a dimostrare una query.
Una variabile composta soltanto dal simbolo _ viene detta *anonima*.

TERMINI COMPOSTI : indicato da un funtore, simbolo usato per il nome di una funzione/predicato, e da una sequenza di termini all'interno di parentesi e separati da una virgola, usati per rappresentare gli argomenti del funtore.

Un programma caricato nel sistema prolog rappresenta una base di conoscenza, mostrabile nei seguenti modi:

- tutte le informazioni vengono rappresentate come argomenti in un'unica relazione e ciò può essere comodo solo per piccole relazioni dato che è difficile da mantenere e coprendere una relazione così definita.
- le informazioni vengono rappresentate con più relazione annidate tra di loro e ciò aumenta la leggibilità e la mantenibilità del programma prolog.
- tutte le relazioni possono essere rappresentate tramite lo schema XML

Per riuscire a stabilire se un goals è una conseguenza diretta del programma il sistema prolog utilizza il principio di risoluzione, utilizzato per effettuare la dimostrazione del programma , e il principio di unificazione, necessario come si evince dal nome per unificare le variabili presenti in una formula del programma.

Nei prossimi due paragrafi considereremo questi due principi al fine di comprendere al meglio come avviene la computazione in un linguaggio logico.

3.1 PRINCIPIO DI RISOLUZIONE

Il principio di risoluzione è una regola di inferenza generalizzata semplice e facilmente implementabile in un calcolatore ed opera su formule ben formate nella forma normale congiuntiva, in cui i letterali si chiamano clausole.
Questo principio viene utilizzato per la dimostrazione di formule ben formate attraverso la refutazione per assurdo, metodologia usata dall'interprete Prolog per computare una data query.

La regola di inferenza ha la seguente forma:

$$\frac{p \vee r,\ s \vee \neg r}{p \vee s} \qquad \frac{\neg r,\ r}{\bot}$$

dove:

- $p \vee s$ è la *clausola risolvente*
- $\bot$ è la *clausola vuota*, che corrisponde all'aver creato una contraddizione da cui posso dedurre qualsiasi cosa, compresa anche la clausola vuota.

Vediamo un'altra regola di inferenza, la**unit resolution**:

$$\frac{\neg p,\ q_1 \vee q_2 \vee ... \vee q_k \vee p}{q_1 \vee q_2 \vee ... \vee q_k}$$

o anche:

$$\frac{p,\ q_1 \vee q_2 \vee ... \vee q_k \vee \neg p}{q_1 \vee q_2 \vee ... \vee q_k}$$

è una regola di risoluzione molto generale, chiamata anche procedura di Davis-Putnam, e se una delle due clausole da risolvere è un *letterale* si parla di *unit resolution*.
Come esempio si può avere:

- non piove, piove e c'è il sole
- si desume quindi che c'è il sole

Ogni FBF può essere in *forma normale a clausola*:

- *formula normale congiunta:* congiunzione di disgiunzioni o di negazione di predicati (sia positivi che negativi):

$$\bigwedge_i (\bigvee_i L_{ij})$$

Esempio. *ecco degli esempi:*

  - $(p(x) \vee q(x,y) \vee \neg t(z)) \wedge (p(w) \vee \neg s((u) \vee \neg r(v))$
  - $(\neg t(z)) \vee (p(w) \vee \neg s(u)) \wedge (p(x) \vee s(x) \vee q(y))$

- *forma normale disgiunta*: disgiunzione di congiunzioni o di negazione di predicati (sia positivi che negativi)

$$\bigvee_i (\bigwedge_j L_{ij})$$

Le clausole con un solo letterale positivo solo le *clausole di Horn*, con lo presenza o meno di letterali negativi, usate per rappresentare tutti i termini delle formule in Prolog.
Ovviamente non tutte le formule ben formate possono essere rappresentate tramite le clausole di Horn per cui il Prolog è un sottoinsieme della logica del Primo Ordine. Attraverso questa restrizione nel 1974 Kowalski produsse un interpretazione procedurale delle dimostrazione e ciò diventò la base della semantica del Prolog.

3.2 PRINCIPIO DI UNIFICAZIONE

Il principio di unificazione è il cuore del modello di computazione dei programmi logici ed è alla base della deduzione automatica e dell'uso dell'inferenza logica nell'intelligenza artificiale.

Per poter definirlo dobbiamo introdurre le seguenti definizioni:

Defi. *Un termine t è un instanza comune di t_1 e t_2 se esistono le sostituzioni α_1 e α_2 che rendono t uguale a $\alpha_1 t_1$ e a $\alpha_2 t_2$.*
Un termine s è un termine generale del termine t se t è un'istanza di s ma s non è un istanza di t.
Un termine s è una variante alfabetica del termine t se t è un istanza di s e s è un istanza di t.

L'unificazione tra due termini consiste nell'effettuare una sostituzione che rende i due termini identici per cui c'è una relazione chiusa con l'instanza comune di due termini.

Il principio di unificazione consiste nel trovare il *mgu*(Most general unifier), l'unificazione la cui instanza comune è un termine generale, e in caso non lo trova riporta un fallimento.
Presentiamo ora alcuni esempi per capire come funziona il Mgu:

```
Mgu(42, 42) % ->{} non serve nessuna sostituzione
Mgu(42, X) % ->{X/42} ovvero X deve essere 42
Mgu(X,42) % ->{X/42} ovvero X deve essere 42
Mgu(foo(bar, 42), foo(bar, X)) % ->{X/42} ovvero X deve essere 42
Mgu(foo(Y, 42), foo(bar, X)) % ->{Y/bar, X/42} ovvero X deve essere 42
Mgu(foo(bar(42), baz), foo(X, Y))  /* ->{X/bar (42), Y/baz}
 ovvero X deve essere  bar(42) e Y baz*/
Mgu(foo(X), foo(bar(Y)))
Mgu(foo(bar(42), baz), foo(X, Y))  /* ->{X/bar (y), Y_:G001}
 ovvero non si ha soluzione */
```

L'Mgu non è altro che il risultato finale della procedura di valutazione del Prolog ed il modo più semplice per vedere se l'unificazione è corretta effettuiamo le seguenti interrogazioni al sistema Prolog:

```
?- 42 = 42.
Yes

?- 42 = X.
X = 42 % per rendere vera l'affermazione serve x = 42
Yes

?- foo(bar, 42) = foo(bar, X).
X = 42
Yes

?- foo(Y, 42) = foo(bar, X).
Y = bar
X = 42
Yes

?- foo(bar(42), baz) = foo(X, Y).
X = bar(42)
Y = baz
Yes

?- foo(X) = foo(bar(Y)).
X = bar(Y)
Y = _G001
Yes

?- foo(42, bar(X), trillian) = foo(Y, bar(Y), X).
No
```

Vediamo ora un programma che effettua la somma tra due numeri naturali:

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

con $s(n)$ interpretato come il successore ($0 = 0$, $s(0) = 1$, $s(s(0)) = 2, \ldots$).
Quando viene specificato un goal esso viene confrontato con tutte le clausole di programma, che usa il procedimento di negazione e di trasformazione in sintassi Prolog come viene mostrato in dettaglio con questo esempio:

- interroghiamo il programma:

$$\exists X \;\; sum(s(0), 0, X) \quad \{X \,/\, s(0)\}$$
$$\exists W \;\; sum(s(s(0)), s(0), W) \quad \{W \,/\, s(s(s(0)))\}$$

dove $\{X \,/\, s(0)\}$ e $\{W \,/\, s(s(s(0)))\}$ sono le sostituzioni che rappresentano il risultato

- usiamo il procedimento di negazione e trasformazione in sintassi:

```
:- sum(s(0), 0, N).     %{N / s(0)}
:- sum (s(s(0)), s(0), W).      %{W / s(s(s(0)))}
```

con $\{N \,/\, s(0)\}$ e $\{W \,/\, s(s(s(0)))\}$ che sono le sostituzioni.

Una computazione corrisponde al tentativo di dimostrare, tramite la regola di risoluzione, che una formula segue logicamente da un programma e si ha inoltre che si deve determinare una sostituzione per le variabili del goal per cui la query segue logicamente dal programma:

Esempio. *Sia dato il programma P e la query:*

```
:- p(t1, t2, ..., tm).
```

se ***X1,..., Xn*** *sono le variabili in* ***t1,..., tm****, il significato della query è:*

$$\exists X1,\ ...,\ Xn\ .\ p(t1, t2, ..., tm)$$

e si cerca una sostituzione:

$$s = \{X1 \,/\, s1, ...,\ Xn \,/\, sn\}$$

con gli s i termini tali per cui:

$$P \vdash s[p(t1, t2, ..., tm)]$$

Dato un insieme di clausole di Horn è possibile derivare la clausola vuota solo se c'è almeno una clausola senza testa, ovvero una query G_0 , in cui si deve dimostrare che da $P \cup G_0$ si può derivare la clausola e ciò avviene per assurdo col principio di risoluzione.
Nel prossimo paragrafo vediamo come effuttua la risoluzione il sistema Prolog, che per efficienza usa una forma particolare di risoluzione.

3.2.1 Risoluzione ad Input Lineare (SLD)

Come già visto, il sistema Prolog per dimostrare la veridicità di un goal esegue una sequenza di passi di risoluzione, il cui ordine di esecuzione determina sistemi di prova più o meno efficienti.
In Prolog la risoluzione avviene sempre fra l'ultimo goal derivato in ciascun passo e una clausola di programma e questa forma è detta *Risoluzione-SLD*, dove le sentenze lineari sono le clausole di Horn.

Defi. *Partiamo dal goal G_i:*

$$G_i \equiv\ ?-\ A_{i,1}, ..., A_{i,m}.$$

e dalla regola:

$$A_r\ \ :-\ B_{r,1}, ..., B_{r,k}.$$

Se esiste un unificatore σ tale che $\sigma[A_r] = \sigma[A_{i,1}]$ allora si ottiene il nuovo goal:

$$G_{i+1} \equiv\ \ B'_{r,1}, ..., B'_{r,k}, A'_{i,1}, ..., A'_{i,m}.$$

che è un passo di risoluzione eseguito dal sistema Prolog (con $\sigma[A_{i,m}] = A'_{i,m}$ 4 $\sigma[B_{i,m}] = B'_{i,m}$.
La scelta di unificare il primo sottogoal di G_i è arbitraria infatti si sarebbe potuto scegliere un arbitario sottogoal di G_i.

Partiamo dal goal G_i:

$$G_i \equiv\ ?-\ A_{i,1}, ..., A_{i,m}.$$

*e dalla regola (ovvero dal **fatto**): A_r. se esiste un unificatore σ tale che $\sigma[A_r] = \sigma[A_{i,1}]$ allora si ottiene il nuovo goal:*

$$G_{i+1} \equiv \quad A'_{i,2}, ..., A'_{i,m}.$$

che ha dimensioni minori di G_i avendo $m-1$ sottogoal

Nella risoluzione SLD si possono avere i seguenti risultati:

- **successo:** si genera la clausola vuota, ovvero se per n finito G_n è uguale alla clausola vuota $G_{:}n \equiv :-$
- **insuccesso finito**: se per n finito G_n non è uguale alla clausola vuota $G_{:}n \equiv :-$ e non è più possibile derivare un nuovo *risolvente* da G_n ed una clausola d programma.
- **insuccesso infinito:** se è sempre possibile derivare nuovi risolventi tutti diversi dalla clausola vuota.

La **sostituzione di risposta** è la sequenza di unificatori usati ed applicata alle variabili nei termini del goal iniziale determina la risposta finale restituita dal sistema Prolog. Durante il processo di generazione di goal intermedi si costruiscono delle varianti dei letterali e delle clausole coinvolti mediante larinominazione di variabili infatti una variante per una clausola C é la clausola C' ottenuta da C rinominando le sue variabili

Esempio. *esempio:*

```
p(X) :- q(X, g(Z)).
% è uguale alla clausola con variabili rinominate:
p(X1) :- q(X1, g(FooFrobboz)).
```

Possono esserci più clausole di programma utilizzabili per applicare la risoluzione con il goal corrente ed esistono diverse strategie di ricerca:

- **in profondità (Depth First)**: si sceglie una clausola e si mantiene fissa questa scelta, finché non si arriva alla clausola vuota o alla impossibilità di fare nuove risoluzioni: in questo ultimo caso si riconsiderano le scelte fatte precedentemente e si riparte a dimostrare in profondità fino a quando è possibile effettuare delle scelte.
- **in ampiezza (Breadth First)**: si considerano in parallelo tutte le possibili alternative

Ogni sistema di programmazione logica sceglie la propria strategia e il Prolog ha deciso di adottare una strategia di risoluzione in profondità con backtracking, ossia la possibilità di ritornare indietro e provare tutte le possibilità.

Introduciamo ora l'importante argomento degli alberi di derivazione, per analizzare come un sistema Prolog effettua la computazione e quali passi esegue per riuscire a stabilire se una data query deriva dal programma o meno.

Defi. *Dato un programma logico P, un goal G_i e una regola di calcolo R si ha che un albero SLD per $P \cup G_i$ via R è definito sulla base del processo di prova visto precedentemente:*

- *ciascun **nodo** dell'albero è un goal (possibilmente vuoto)*
- *la **radice** dell'albero SLD è il goal G_0*
- *dato il nodo: $:- A_1, ..., A_{m-1}, A_m, A_{m+1}, ..., A_k$ se A_m è il sottogoal selezionato dalla regola di calcolo R, allora questo nodo (genitore) ha un nodo figlio per ciascuna clausola del tipo:*

$$C_i \equiv A_i \ :- \ B_{i,1}, ..., B_{i,q}$$
$$C_k \equiv A_k$$

di P tale che A_i e A_m (A_K e A_m) sono unificabili attraverso la sostituzione più generale σ.
Il nodo figlio è etichettato con la clausola goal

$$:- \ \sigma[A_1, ..., A_{m-1}, B_{i,i}, ..., B_{i,q}, A_{m+1}, ..., A_k]$$
$$:- \ \sigma[A_1, ..., A_{m-1}, A_m, A_{m+1}, ..., A_k]$$

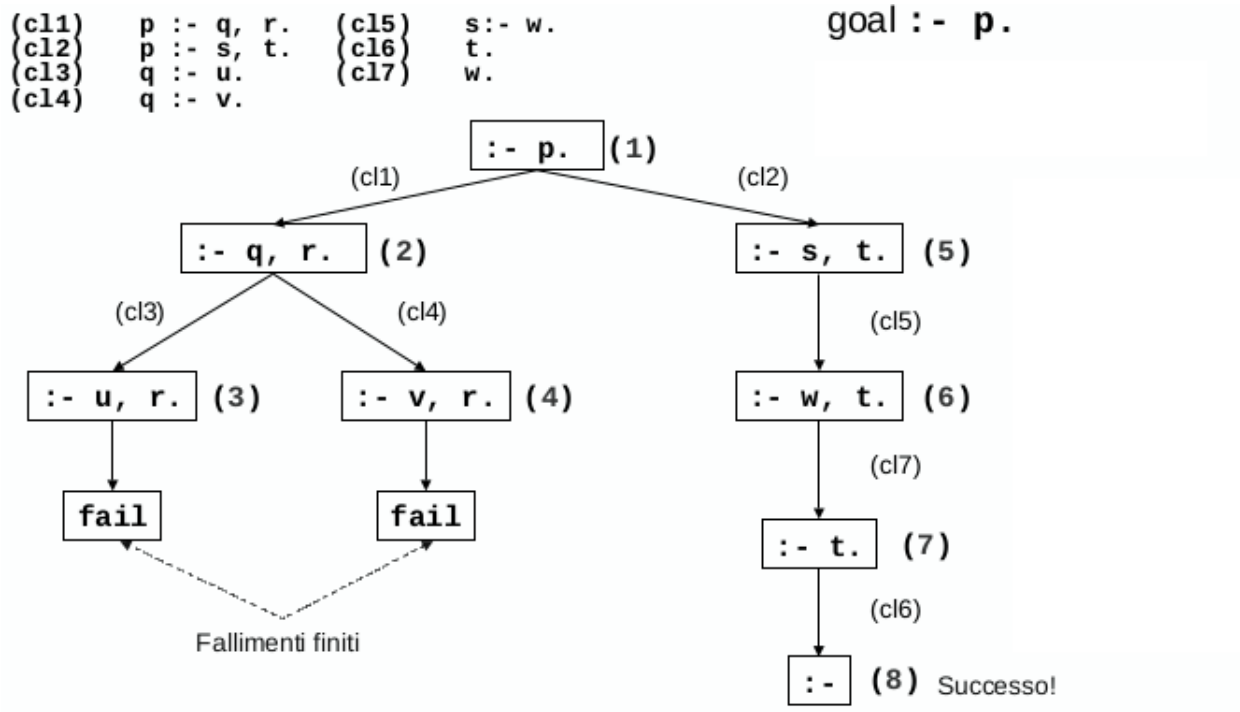
e il ramo dal nodo padre al figlio è etichettato dalla sostituzione σ e dalla clausola selezionata C_i o C_k ed infine si ha che il nodo $:-$ non ha figli.

La regola R è variabile e le più comuni utilizzate sono:

- **Left-most**: si sceglie sempre di risolvereil sottogoal più a sinistra
- **Right-most**: si sceglie sempre la risoluzione del sottogoal più a destra
- si può avere la scelta di un sottogoal a caso
- se si ha un modo per decider il miglior sottogoal

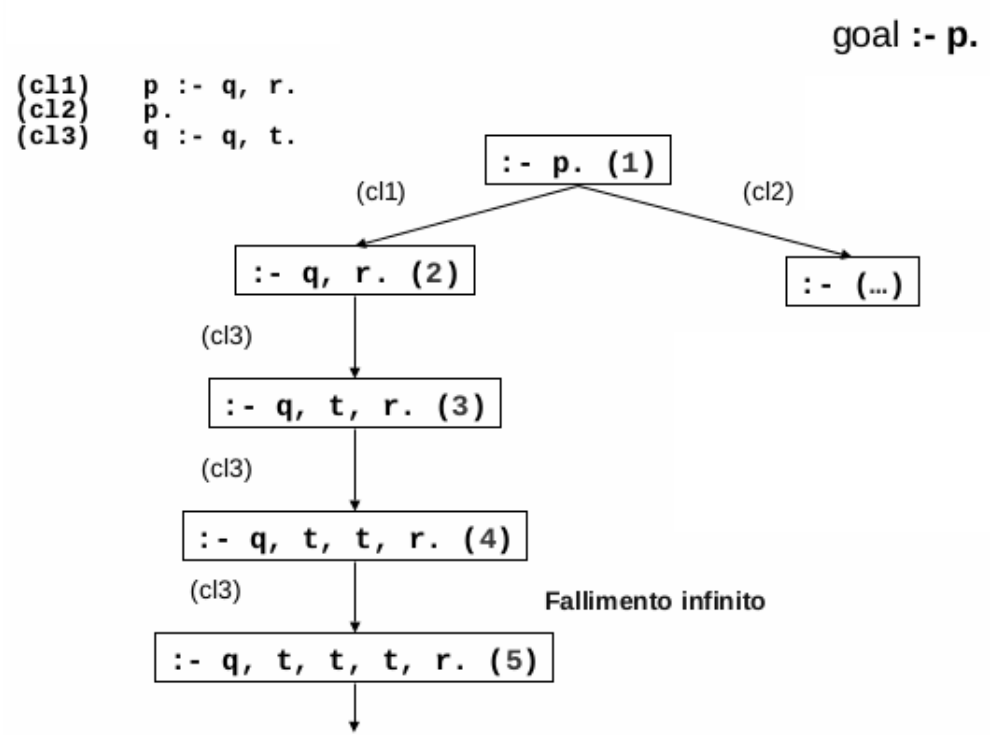
Ogni sistema logico decide la propria modalità di applicazione della regola e il sistema Prolog ha deciso di usare sempre la regola left-most.
L'albero SLD, generato implicitamente dal sistema Prolog, ordina i figli di un nodo secondo l'ordine dall'alto verso il basso delle regole e dei fatti del programma P, come si nota nell'esempio mostrato di seguito.

Esempio. *Ecco un esempio:*



Ad ogni ramo di un albero SLD corrisponde una derivazione SLD e ogni ramo che termina con il nodo vuoto $:-$.
La regola di calcolo influisce sulla struttura dell'albero per quanto riguarda l'ampiezza e la profondità ma non influisce sulla correttezza e completezza, infatti tra un albero left-most e un albero right-most cambia solo l'ordine e il tempo necessario per stabilire se un ramo è un successo oppure no.

Esempio. *Ecco un altro esempio, con fallimento infinito, dove la clausola vuota può essere generata ma il Prolog non è in grado di trovare questa soluzione dato che la sua strategia di percorrimento dell'albero (implicito) di soluzioni è depth-first con backtracking :*



3.2.2 Cut e Backtracking

Introduciamo ora il predicato *Cut*, attraverso cui possiamo controllare il backtracking, in quanto si tagliano certe possibilità di ritornare indietro nelle scelte durante la computazione.
Il predicato cut si indica con ! e il Prolog effettua un'interpretazione procedurale, sempre per il fatto che viene eseguito su un calcolatore.

Come abbiamo affermato precedentemente, le clausole nel data base di un programma Prolog vengono considerate "da sinistra, verso destra" e "dall'alto al basso" per cui se un (sotto)goal fallisce, allora il dimostratore Prolog, sceglie un'alternativa, scandendo "dall'alto" verso "il basso" la lista delle clausole.
Questa procedurà può venire controllata dal *cut* infatti per esempio:

```
a :- b1, b2, ..., bk, !, ..., bn.
```

Questo è l'effetto del cut:

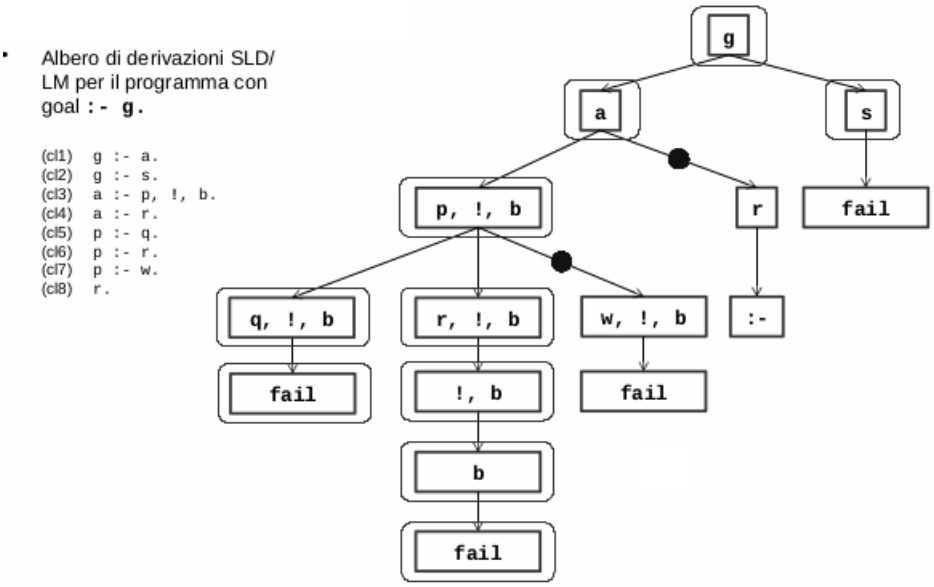
- se il goal corrente G unifica con a e $b_1, ..., b_k$ hanno successo, allora il dimostratore si impegna inderogabilmente alla scelta di C per dimostrare G.
- ogni clausola alternativa (successiva, in basso) per a che unifica con G viene ignorata
- se un qualche b_j con $j > k$ fallisse, il backtracking si fermerebbe al cut e le altre scelte sono rimosse dall'albero di derivazione
- quando il backtracking raggiunge il cut, allora il cut fallisce e la ricerca procede dall'ultimo punto di scelta prima che G scegliesse C

Il Prolog per la gestione della computazione utilizza due stack:

- stack delle scelte: contiene l'insieme delle scelte possibili ed ad ogni fase della valutazione contiene i puntatori alle scelte aperte nelle fasi precedenti della dimostrazione

- stack di esecuzione: contiene i record di attivazione delle varie procedure, ovvero le sostituzioni per l'unificazione delle varie regole.

Vediamo un albero di derivazione in caso di cut:



Si hanno due tipi di cut:

- **green cut:** utili per esprimere il "determinismo", come vedremo ora, e quindi per rendere più efficiente il programma in quanto non vengono analizzate clausole certamente false.

- **red cut:** usati per soli scopi di efficienza ed hanno come caratteristica principale quella di omettere alcune condizioni esplicite in un programma, ma soprattutto quella di modificare la semantica del programma.

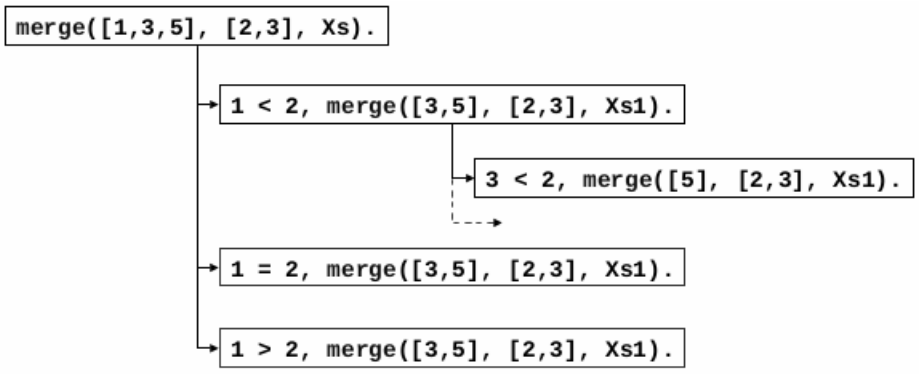
Per capire l'importanza e l'utilizzo del cut, consideriamo ora il seguente codice:

```
/* merge di due liste ordinate*/

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

/+ minimo tra due numeri */
minimum(X, Y, X) :- X =< Y.
minimum(X, Y, Y) :- Y < X.
```

Vediamo cosa succede come computazione nell'interprete Prolog:



Solo la prima clausola ha successo, le altre due falliscono al momento del confronto numerico; ciononostante tutte e tre le clausole vengono considerate

consideriamo la seguente query:

```
?- merge([], [], Xs).
Xs = [];
Xs = [];
False.
```

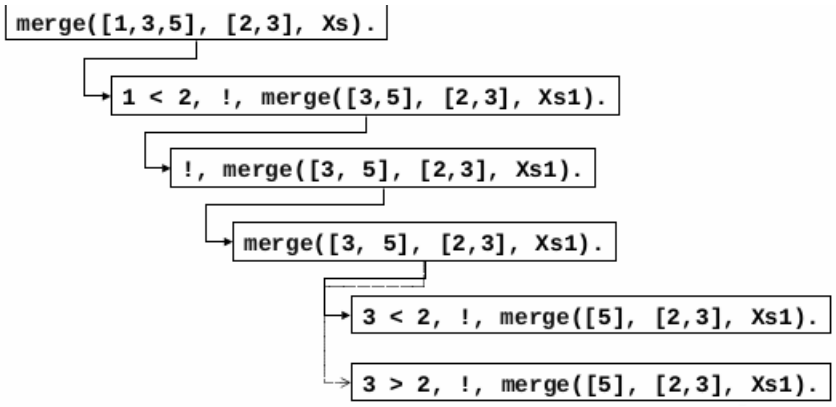
Questa implementazione del merge ha purtroppo una soluzione di troppo e questo è un esempio di determinismo, ossia quando una sola delle clausole serve (o si vorrebbe servisse) per provare un dato goal.
Si usano quindi i seguenti green cuts:

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

Interrogando il sistema Prolog come il seguente esempio otteniamo:

```
?- merge([], [], Xs).
Xs = [];
False.
```

Guardando ai passi necessari, effettuati dall'interprete Prolog, per rispondere ad un query abbiamo:



Solo la prima clausola ha successo, ed il cut fa si che la seconda e la terza clausola **non** vengano considerate

Il predicato per calcolare il minimo diventa:

```
minimum(X, Y, X) :- X =< Y, !.
minimum(X, Y, Y) :- Y < X, !.%è ridontante ma viene inserito per simmetria
```

Una volta che il programma ha fallito la prima clausola (ovvero il test X =< Y) al sistema Prolog non rimane che controllare la clausola seguente.
Riscrivo in maniera non simmetrica:

```
minimum(X, Y, X) :- X =< Y, !.
minimum(X, Y, Y).
```

Qui si ha un red cut dato che taglia solo delle soluzioni, portando anche a risultati errati, infatti $minimum(2,5,5)$ risulta verificato dato che il secondo termine è un fatto sempre verificato.

3.3 STRUTTURE DATI ED AMBIENTE PROLOG

Si definisce una lista in Prolog racchiudendo gli elementi (termini e/o variabili logiche) della lista tra parentesi quadre [e] e separandoli da virgole.
Gli elementi di una lista in Prolog possono essere termini qualsiasi o liste e la lista vuota si indica con [].
Una lista non vuota si può dividere in *testa* e *coda*:

- la testa è il primo elemento della lista
- la coda rappresenta tutto il resto ed è sempre una lista

Presentiamo ora degli esempi di liste in Prolog:

```
[a, b, c] % a è la testa e [b, c] la coda
[a, b] % a è la testa e [b] la coda
[a] % a è la testa e [] la coda
[[a]] % [a] è la testa e [] la coda
[[a, b], c] % [a, b] è la testa e [c] la coda
[[a, b], [c], d] % [a, b] è la testa e [[c], d] la coda
```

Prolog possiede uno speciale operatore usato per distinguere tra l'inizio e la coda di una lista: l'operatore |:

```
?- [X | Ys] = [mia, vincent, jules, yolanda].
X = mia
Ys = [vincent, jules, yolanda]
Yes
```

```
?- [X, Y | Zs] = [the, answer, is, 42].
X = the
Y = answer
Zs = [is, 42]
Yes

?- [X, 42 | _] = [41, 42, 43, foo(bar)].
X = 41
Yes
```

La lista vuota in prolog è gestita come una lista speciale infatti risulta:

```
?- [X | Ys] = [].
No
```

Nonostante le liste sono definite di default in Prolog, con anche molte operazioni implementate come append, length ed ecc..., definiamo una libreria personale per gestire le liste infatti questo è il codice per implementare una lista in Prolog:

```
%%%% -*- Mode:Prolog -*-

%%%% Libreria per gestire le liste
%%%% e le sue operazioni elementari.

%Definizione di Lista
lists([]).
lists([X| Xs]) :-
          lists(Xs).

%Operazione lengthList/2 per calcolare
%la lunghezza di una lista.
lengthList([], 0).
lengthList([X | Xs], R) :-
    lists([X | Xs]),
    lengthList(Xs, M),
    R is M+1.

%predicato is_empty/1 per stabilire se una lista è vuota
is_empty([]).

%Predicato member/2 per stabilire se un elemento appartine o meno alla lista
member(X, [X | Xs]).
member(X, [Y | Ys]) :- member(X, Ys).

%Predicato appendList/3 per effettuare la concatenazione di due liste
appendList([], X, X).
appendList([X | Xs], Y, [X | Z]):-
    appendList(Xs, Y, Z).

%Predicato nth/3 per ottenere l'elemento n-esimo
nth(0, X, X).
nth(N, [_| X], X) :-
    N1 is N-1,
    nth(N1, X, X).

%Predicato insert/3 per inserire un elemento in una lista
insert(X, [], X).
```

```
insert(X, Y, [X | Y]).

%predicato delete/3 per eliminare un elemento di una lista
delete(X, X, []).
delete(X, [X | Xs], Xs).

%predicato search/2 per ricercare un elemento di una lista
search(X, X).
search(X, [Y | Ys]) :-
    search(X, Ys).

%predicato min/2 per calcolare il minimo degli elementi di una lista
min([X], X).
min([X | Xs], X) :-
    min(Xs, Z),
    number(X),
    number(Z),
    X < Z.
min([X | Xs], Z):-
    min(Xs, Z),
    number(X),
    number(Z),
    Z < X.

%predicato max/2 per calcolare il massimo degli elementi di una lista
max([X], X) :- number(X).
max([X | Xs], X) :-
    max(Xs, Z),
    number(X),
    number(Z),
    X > Z.
max([X | Xs], Z) :-
    max(Xs, Z),
    number(Z),
    number(X),
    Z > X.
%% end of file lists.pl
```

In Prolog la base di conoscenza è nascosta al controllo diretto degli utenti ed è accessibile solo tramite opportuni comandi per cui bisogna poter caricare un insieme di fatti e regole nell'ambiente Prolog, nel nostro caso SW-Prolog.
Per farlo si ha il comando **consult**, che appare come un predicato da valutare (un goal) e prende almeno un termine che denota un file come argomento, contenente la nostra base di conoscenza.

```
?- consult('guida-astrostoppista.pl').
Yes

?- consult('Projects/Lang/Prolog/Code/esempi-liste.pl').
Yes
```

Il predicato **reconsult** viene usato quando si vuole ricaricare un file nell'ambiente Prolog.
L'effetto è di prendere i predicati presenti nel file, rimuoverli completamente dal data base interno e di reinstallarli utilizzando le nuove definizioni:

```
?- reconsult('guida-astrostoppista.pl').
Yes
```

```
% A questo punto la base di dati Prolog contiene il
% nuovo contenuto del file.

?- reconsult(user). % Notare il sotto-prompt.
|- foo(42).
|- friends(zaphod, trillian).
|- ^D
Yes

% A questo punto la base di dati Prolog contiene i due
% fatti inseriti manualmente.
?- friends(zaphod, W).
W = trillian
Yes
```

3.4 CARATTERISTICHE PARTICOLARI DEL SISTEMA PROLOG

Dopo aver visto i background teorici e alcuni esempi basilari di programmi logici, introduciamo alcune caratteristiche più avanzate per poter semplificare e rendere più efficiente un programma prolog, ed incominciamo dall'aritmetica.

3.4.1 Aritmetica in Prolog

Il Prolog fornisce dei predicati standard per gestire ed effettuare le operazioni aritmetiche, i quali a differenza dei soliti predicati che derivano dalla logica, questi effettuano ed eseguono direttamente usando l'hardware, perdendo la possibilità di effettuare l'instanzazione tramite l'unificazione ma dovendola fare direttamente alla chiamata dei predicati aritmetici.
Il Prolog prevede ed usa gli operatori matematici, con la loro relativa precedenza, $+ - */$.
I predicati standard comunemente usato per la gestione dell'aritmetica sono i seguenti:

- >(Expr1, Expr2): stabilisce se Expr1 è maggiore dell'Expr2
- <(Expr1,Expr2): stabilisce se Expr1 è minore di Expr2
- <=(Expr1,Expr2): vero se Expr1 è minore o uguale a Expr2
- >=(Expr1,Expr2): verificato se Expr1 è maggiore di Expr2
- =(Expr1,Expr2): verificato se Expr1 è diverso ad Expr2
- =(Expr1,Expr2): verificato se l'Expr1 viene valutata come l'Expr2
- *is*(Number,Expr2): verificato se Number è la valutazione di Expr2

Esempi:

3.4.2 Predicati Meta-Logici

Vediamo il predicato:

```
celsius_fahrenheit(C, F) :- C is 5/9 * (F - 32).
```

Questo predicato non è invertibile infatti si deve decidere qual è l'input e qual è l'output ma per risolvere il problema usiamo i predicati meta-logici, introdotti in

questo paragrafo.
La ragione dell'impossibilità di invertire deriva dall'uso che abbiamo fatto di vari predicati aritmetici nel corpo dei predicati (>, <, =<, is, etc), infatti per poter usare i predicati aritmetici che usano direttamente l'hardware abbiamo sacrificato la semantica dei nostri programmi.

I predicati meta-logici principali trattano le variabili come oggetti del linguaggio e ci permettono di riscrivere molti programmi che usano i predicati aritmetici di sistema come predicati dalla semantica "corretta" ed dal comportamento invertibile. I comuni predicati predicati importanti:

- *var(X):* vero se *X* è una variabile logica
- *nonvar(X):* vero se *X* non è una variabile logica
- *integer(X):* risulta verificato se *X* è un intero
- *number(X):* risulta verificato se X è un numero intero o in virgola mobile
- *float(X)*:: risulta verificato se X è un numero in virgola mobile

ovvero:

```
?- var(foo).
False
?- var(X).
True
?- nonvar(42).
True
?- integer(45).
True
?- integer(abc).
False
?- number(12.456).
True
?- number(a).
False
```

il nostro programma dei gradi diventa:

```
celsius_fahrenheit(C, F) :-
        var(C), nonvar(F), C is 5/9 * (F - 32).
celsius_fahrenheit(C, F) :-
        var(F), nonvar(C), F is (9/5 * C) + 32
```

con var decido che clausola usare e l'uso di questi predicati ci permette di scrivere programmi efficienti e semanticamente corretti.

Posso chiedere a prolog se ho a che fare con termini atomici o scomposti, con i seguenti predicati:

- *atomic(X):*vero se *X* è un numero od una costante
- *compound(X):* vero se non *atomic(X)*
- *atom(X)*: verificato se e solo se X è una costante ma non accettale stringhe

Esempi:

```
?- atomic(43).
True
?- atomic(foo(bar)).
False
?- compound(42).
```

```
False
?- compound(foo(X)).
True
```

Ho per manipolare un termine, denotato con *Term*, i seguenti principali predicati:

- **functor(Term, F, Arity)**, vero se *Term* è un termine, con *Arity*argomenti, il cui funtore (simbolo di funzione o di predicato) è *F*.
- **arg(N, Term, Arg)**, vero se l'N-esimo argomento di *Term* è *Arg*.
- **Term =.. L** questo predicato, =..,viene chiamato (per motivi storici) **univ**; risulta verificato quando L è una lista il cui primo elemento è il funtore di *Term* ed i rimanenti elementi sono i suoi argomenti.

ovvero:

```
?- functor(foo(24), foo, 1).
YES
?- functor(node(x, _, [], []), F, 4).
F = node
Yes
?- functor(Term, bar, 2).
Term = bar(_0,_1)
Yes
?- arg(3, node(x, _, [], []), X).
X = []
Yes
?- arg(1, father(X, lot), haran).
X = haran
Yes
?- father(haran, lot) =.. Ts.
Ts = [father, haran, lot]
Yes
?- father(X, lot) =.. [father, haran, lot].
X = haran
Yes
```

3.4.3 Programmazione di ordine superiore

Quando si formula una domanda per il sistema Prolog, ci si aspetta una risposta che è un'istanza (individuale) derivabile dalla knowledge base e col backtracking ne otteniamo una alla volta, ma se io volessi tutte le risposte si entrerebbe nel campo della logica del secondo ordine.
Per risolvere questo problema il sistema Prolog fornisce i seguenti predicati:

- **findall(Template, Goal, Set):**
  - Vero se Set contiene tutte le istanze di Template che soddisfano Goal
  - *Le istanze di Template vengono ottenute mediante backtracking*
- **bagof(Template, Goal, Bag):**
  - Vero se Bag contiene tutte le alternative di Template che soddisfano Goal
  - Le alternative vengono costruite facendo backtracking solo se vi sono delle variabili libere in Goal che non appaiono in Template
  - È possibile dichiarare quali variabili non vanno considerate libere al fine del backtracking grazie alla sintassi Var^G come Goal; In questo caso Var viene pensata come una variabile esistenziale

- *setof(Template, Goal, Set)*, che si comporta come bagof, ma Set non contiene soluzioni duplicate

Per comprendere al meglio presentiamo i seguenti esempi:

```
/* findall */

?- findall(C, father(X, C), Kids).
C = _0
X = _1
Kids = [abraham, nachor, haran, isaac, lot, milcah, yiscah]
True.

/* bagof */

?- bagof(C, father(X, C), Kids).
C = _0
X = terach
KIDS = [abraham, haran, nachor];
C = _0
X = haran
KIDS = [lot, yiscah, milcah];
C = _0
X = abraham
KIDS = [isaac];
False.

/* bagof con variabile esistenziale */

?- bagof(C, X^father(X, C), Kids).
C = _0
X = _1
Kids = [abraham, haran, lot, yiscah, nachor, isaac, milcah];
False.
```

Buona parte dei predicati di ordine superiori, funzionano grazie al meccanismo delle meta-variabili, ovvero variabili interpretabili come goals.

Per esempio si ha il predicato *call*:

```
call(G) :- G.
```

Possiamo quindi definire il predicato *apply* che valuta una query composta da un funtore e da una lista di argomenti:

```
apply(P, Argomenti) :-
  P =.. PL, append(PL, Argomenti, GL), Goal =.. GL, call(Goal).
```

```
?- apply(father, [X, C]).
X = terach
C = abraham;
X = terach
C = nachor;
False

?- apply(father(terach), [C]).
C = abraham;
```

```
C = nachor;
False
```

3.4.4 Manipolazine della base di Dati

Un programma Prolog è costituito da una base di conoscenza(knowledge base) che contiene **fatti** e **regole**, su cui il programma interroga il sistema per sapere se una data query risulta verificata.
Il Prolog però mette a disposizione anche altri predicati che servono a manipolare direttamente la base di dati ma ovviamente, questi predicati vanno usati con molta attenzione, dato che modificano dinamicamente lo stato del programma:

- listing: per mostrare tutti gli elementi della base di conoscenza
- assert, asserta, assertz: per effettuare un'asserzione ossia inserire elementi nella nostra base di conoscenza
- retract: per eliminare gli elementi in una base di conoscenza
- abolish: per abolire degli elementi

se si ha una knowledge base vuota si avrà:

```
?- listing.
True
```

ovvero solo True e il *listing* è vuoto.
Aggiungo ora qualcosa alla base dati con:

```
?- assert(happy(maya)).
true
```

che risponderà sempre *true*, ora si avrà:

```
?- listing.
happy(maya).
true
```

e la base dati non sarà più vuota.
Aggiungo altro alla base dati:

```
?- assert(happy(vincent)).
true

?- assert(happy(marcellus)).
true
```

```
?- assert(happy(butch)).
true

?- assert(happy(vincent)).
true
```

il *listing* ora darà:

```
?- listing.
happy(mia).
happy(vincent).
happy(marcellus).
happy(butch).
happy(vincent).
true
```

Si possono anche asserire regole usando assert e mettendo la regola tra parentesi:

```
?- assert( (naive(X) :- happy(X)) ).
true

?- listing.
happy(mia).
happy(vincent).
happy(marcellus).
happy(butch).
happy(vincent).
naive(A) :-
happy(A).
true
```

possiamo anche rimuovere fatti e regole con retract, riprendendo dagli esempi sopra:

```
?- retract(happy(marcellus)).
true

?- listing.
happy(mia).
happy(vincent).
happy(butch).
happy(vincent).
naive(A) :-
happy(A).
true
```

inoltre si ha che retract rimuove solo la prima occorrenza:

```
?- listing.
happy(mia).
happy(vincent).
happy(butch).
happy(vincent).
naive(A) :-
happy(A).
true

?- retract(happy(vincent)).
true

?- listing.
happy(mia).
happy(butch).
happy(vincent).
naive(A) :-
happy(A).
true
```

Per rimuovere tutte le nostre asserzioni possiamo usare una variabile:

```
?- retract(happy(X)).
X = mia;
X = butch;
X = vincent;
false

?- listing.
naive(A) :-
happy(A)
true
```

Per avere più controllo su dove vengono aggiunti fatti e regole possiamo usare le due varianti di assert:

1. assertz: inserisce l'asserzione alla fine della knowledge base
2. asserta: inserisce l'asserzione all'inizio della knowledge base

ovvero, partendo da una base dari vuota:

```
?- assert(p(b)), assertz(p(c)), asserta(p(a)).
true

?- listing.
p(a).
p(b).
```

```
p(c).
true
```

La manipolazione dati può essere usata per memorizzare i risultati intermedi di varie computazioni, in modo da non dover rifare delle queries dispendiose in futuro: semplicemente si ricerca direttamente il fatto appena asserito e questa tecnica si chiama **memorization** o **caching**.

Esempio. *Creiamo una tavola di addizioni manipolando la knowledge base:*

```
addition_table(A) :-
  member(B, A),
  member(C, A),
  D is B + C,
  assert(sum(B, C, D)),
  fail.
```

dove member(X, Y) controlla che il predicato X appartenga alla lista Y.

```
?- addition_table([0, 1, 2, 3, 4, 5, 6, 7, 8, 9]).
false
```

La risposta è false; ma non è la risposta che ci interessa, bensì l'effetto (collaterale) che l'interrogazione ha sulla knowledge base:

```
?- listing(sum).
sum(0, 0, 0).
sum(0, 1, 1).
sum(0, 2, 2).
sum(0, 3, 3).
...
sum(9, 9, 18).
true
```

potremmo ora rimuovere tutti questi fatti non con il solito

```
?- retract(sum(X, Y, Z)).
```

che dovremmo ripetere ogni volta per ogni fatto ma con:

```
?- retract(sum(_, _, _)), fail.
false
```

Ancora una volta, lo scopo del fail *è di forzare il backtracking infatti il Prolog rimuove il primo fatto con funtore sum dalla base di dati e poi fallisce.*

Quindi fa backtrack e rimuove il fatto successivo e così via per cui alla fine, dopo aver rimosso tutti i fatti con funtore sum, la query fallirà completamente ed il Prolog risponderà (correttamente) con un false.
Ma anche in questo caso a noi interessa unicamente l'effetto collaterale sulla knowledge base.

3.4.5 Input e Output in Prolog

I predicati primitivi principali per la gestione dell'I/O in Prolog sono essenzialmente due, **read** e **write**, a cui si aggiungono i vari predicati per la gestione dei files e degli streams: **open, close, seek**, etc... Il predicato write è l'equivalente del metodo *toString* in Java su un oggetto di classe "termine" mentre *read* invoca il parser prolog:

```
?- write(42).
42
true

?- foo(bar) = X, write(X).
foo(bar)
X = foo(bar)
?- read(What).
|: foo(42, Bar).
What = foo(42, _G270).

?- read(What), write('I just read: '), write(What).
|: read(What).
I just read: read(_G301)
What = read(_G301).

?- open(`some/file/here.txt`, write, Out),
write(Out, foo(bar)), put(Out, 0'.), nl(Out),
close(Out).
true % But file "some/file/here.txt" now contains the term 'foo
(bar).'

?- open('some/file/here.txt', read, In),
read(In, What)
close(In).
What = foo(bar)
```

open e **close** servono per leggere e scrivere files; la versione più semplice di open ha con tre argomenti: un atomo che rappresenta il nome del file, una "modalità" con cui si apre il file ed un terzo argomento a cui si associa l'identificatore del file.

Esiste anche il predicato **put** che emette un carattere sullo stream ed il predicato nl che mette un 'newline' sullo stream ed il Prolog usa la notazione *0'c* per rappresentare i caratteri come termini.

3.5 INTERPRETI IN PROLOG

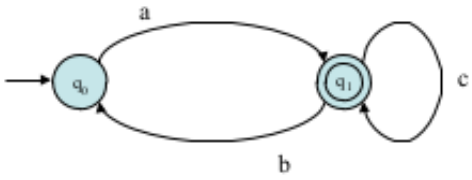
Uno degli utilizzi del Prolog consiste nella costruzione di interpreti per la manipolazione di linguaggi specializzati(Domain Specific Languages), i più famosi sono i seguenti:

- intepreti per Automi a Stati Finiti, Automi a Pila e Macchine di Turing(collegamento con l'informatica Teorica)
- sistemi per la deduzione automatica, utilizzati per il Machine Learning e AI
- sistemi per la manipolazione del Linguaggio Naturale (Natural Language Processing)

In questo corso vediamo come definire un minimale, molto minimale, interprete per gli automi, incominciato per prima con quello a stati finiti:

3.5.1 Interprete degli automi a stati finiti

Dato il seguente automa:



in cui i nodi rappresentano degli stati e si ha uno stato inziale (q_0) e uno finale, denotato con un doppio cerchio, (q_1).
Gli archi sono le transizioni tra due stati e nel nostro caso generano caratteri e nell'esempio ho soltanto stringhe che iniziano con a.
In q_1 posso fermarmi e generare solo la stringa a o produrre altri caratteri in numero variabile e le stringhe qui prodotte saranno del tipo $ac^n(ba)^n$, con $n \geq 0$.
Per decidere se una certa sequenza di simboli è riconosciuta dall'automa possiamo costruire il seguente predicato:

```
/* definisco l'interprete */
accept([I | Is], S) :-
  delta(S, I, N),
  accept(Is, N).
accept([], Q) :- final(Q). /* è l'uscita, se ho una lista vuota
                              ho generato tutto */

/* richiesta di riconoscere un automa con stato iniziale S
    poi si usa l'accept con input che rappresenta la stringa
     considerando lo stato inziale S */
recognize(Input) :- initial(S), accept(Input, S).

initial(q0). % stato iniziale
final(q1). % stato finale

/* definisco le transizioni, con delta a tre argomenti */
delta(q0, a, q1).
delta(q1, b, q0).
delta(q1, c, q1).
```

```
?- recognize([a, b, a, c, c, b, a]).
Yes

?- recognize([a, b, a, c, b]).
No
```

Per riuscire a comprendere come funziona l'interprete vediamo il suo funzionamento considerando la lista $[a, b, a, c, c, b, a]$, in cui si hanno le seguenti sostituzioni: $I\backslash a$ e $S\backslash q0$.
Avrò quindi:

```
delta(q0, a, N).
accept([b, ...], N).
```

Ora si effettuano le seguenti sostituzoni: $N\backslash q1$:

```
accept([b, ...], q1). % è il nuovo goal
```

unifico ancora con le sostituzioni $I2\backslash b$ e $Is\backslash[a, c, c, b, a]$:

```
delta(q1, b, N2).
accept([a, c, c, b, a], N2).
```

ad un certo punto arriveremo alla fine:

```
accept([], q1).
```

che unifica con la seconda clausola, sostituendo $q1\backslash Q$, arrivando a:

```
final(q1).
```

Dopo aver considerato come si definisce un interprete per automi, possiamo considerare degli interpreti più sofisticati se accettiamo di rappresentare i programmi, usando una sintassi leggermente diversa:

```
rule(append([], X, X)).
rule(append([X | Xs], Ys, [X | Zs]), [append(Xs, Ys, Zs)]).

solve(Goal) :- solve(Goal, []). /* lista goal e lista di appoggio
                                   dove mettere i goal restanti */

solve([], []). % clausola di uscita

solve([], [G | Goals]) :-
  solve(G, Goals).
solve([A | B], Goals) :-
  append(B, Goals, BGoals),
  solve(A, BGoals).
solve(A, Goals) :-
  rule(A),
  solve(Goals, []).
solve(A, Goals) :-
  rule(A, B),
  solve(B, Goals).
```

Il programma *solve* è un meta-interprete per i predicati rule che compongono il nostro sistema/programma Prolog. Ovviamente fino ad ora abbiamo considerato di rappresentare dei sistemi perfetti, in cui si può inferire in maniera certa su dei fatti/regole cosa che difficilmente nella realtà avviene, per cui soprattutto per definire degli interpreti per l'apprendimento automatico, per questo aggiungiamo ad ogni predicato/fatto un argomento numerico, tra 0.0 e 1.0, indicante la probabilità che il fatto/predicato risulti verificato.
Ad esempio modifichiamo il precedente esempio per considerare l'incertezza:

```
solve_cf(true, 1) :- !.
solve_cf((A, B), C) :-
  !,
```

```
  solve_cf(A, CA),
  solve_cf(B, CB),
  minimum(CA, CB, C).
solve_cf(A, 1) :-
  builtin(A),
  !,
  call(A).
solve_cf(A, C) :-
  rule_cf(A, B, CR),
  solve_cf(B, CB),
  C is CR * CB.
```

Il programma $solve_cf$ è un meta-interprete per stabilire se un goal G è vero e quanto siamo certi che sia vero.

4 LISP:PROGRAMMAZIONE FUNZIONALE

Dopo aver affrontato ed analizzato il Prolog e il paradigma Logico, è arrivato il momento di analizzare il paradigma "matematico" funzionale, in cui la computazione risiede nel definire e applicare una serie di funzioni, anche composte e soprattutto ricorsive.
Nel paradigma funzionale risulta verificato il concetto matematico della **trasparenza refenziale**, funzione senza effetti collaterali e che quando riceve lo stesso parametro in input, restituisce sempre lo stesso valore.

L'uso della trasparenza referenziale ha il grande vantaggio di permettere al programmatore di poter contare su un comportamento univoco delle funzioni, a priori non sempre prevedibili e ciò aiuta molto nel processo di testing e debug, semplifica l'implementazione di algoritmi, rende più facili la modifica e l'ottimizzazione dei programmi senza cambiarne radicalmente la struttura.

Nel paradigma funzionale vi sono oggetti di vario tipo e strutture di controllo, ma vengono raggruppati logicamente in modo diverso da come invece accade nel paradigma imperativo, infatti risulta utile pensare in termini di:

- espressioni, il quale rappresenta i fondamenti del linguaggio come le funzioni semplici e primitive.
- modi di combinare le espressioni per ottenerne di più complesse, tramite l'operazione di composizione
- modi e metodi di costruzione "astratte" per poter far riferimento a gruppi di espressioni per "nome" e per trattarle come unità separate
- operatori speciali (condizionali ed altri ancora, che verranno introdotti in seguito)

Noi affronteremo come esempio di linguaggio funzionale, il linguaggio Common Lisp, una dei principali dialetti, come anche lo Scheme, della famiglia di linguaggi chiamati Lisp.
Lo studio del Lisp, anche se in una delle sue incarnazioni, è importante dato che è il primo linguaggio di programmazione funzionale e le sue versioni minimali ammettono:

- funzioni primitive su **liste**
- un'operatore speciale **lambda** per creare funzioni
- un'operatore codizionale **cond**
- un piccolo insieme di predicati ed operatori speciali

Incominciamo a mostrare come valuta e computa le espressione il Common Lisp e la prima cosa notata è che ogni "espressione" denota un valore.

```
prompt> 42
42

prompt> "Sapete che cos'e` '42'?"
"Sapete che cos'e` '42'?"
```

In Lisp sono presenti le operazioni aritmetiche principali(+ - * /), rappresentate in notazione prefissa, il quale permettono di essere applicate a più argomenti ed evitano di generare ambiguità su quale operazione viene applicata in caso di più operazioni, come si nota nei seguenti esempi:

```
prompt> (+ 137 349)
486

prompt> (- 1000 334)
666

prompt> (+ 2.7 10)
12.7

prompt> (* 2 34)
68

prompt> (/ 10 5)
2
```

Queste espressioni, formate delimitanto una lista di espressioni all'interno di parentesi per mostrare l'applicazione di una funzione, sono chiamate *combinazioni* e il valore viene determinato applicando la procedura, specificata dall'operatore, agli argomenti, i quali sono i valori degli operandi. Le espressioni possono essere annidate, per cui per motivi di lettura si allineano gli argomenti di chiamata verticalmente, come nell'esempio:

```
prompt> (+ (* 3
              (+ (* 2 4)
                 (+ 3 2)))
           (+ (- 10 8)
              1))
42
```

Notiamo come le funzioni aritmetiche elementari + e * in (Common) LISP rispettano i vincoli di "campo" algebrico:

```
prompt> (+)
0

prompt> (*)
1
```

Un aspetto dei linguaggi di programmazione è quello di fornire dei nomi, per riferirsi agli oggetti computazionali, ed ovviamente anche nel Common Lisp ciò avviene tramite l'istruzione *defparameter*, che si comporta nel seguente modo:

```
(defparameter pi_greco 3.14)
(defparameter size 2)
```

Per valutare una espressione combinata, l'interprete Lisp esegue le seugenti operazioni:

1. viene valutata la sottoespressione, presente nella combinazione
2. viene applicata la procedura, rappresentata dalla sottoespressione left-most, agli argomenti rappresentati dai valori delle altre sottoespressioni.

Ovviamente come si può notare la valutazione risulta ricorsiva e i componenti basilari, su cui si riesce ad effettuare correttamente la valutazione sono i seguenti:

- il valore dei numeri sono ovviamente il numero stesso
- il valore degli operatori built-in sono le istruzioni macchina che corrispondono a quell'operatore.

Figura 3: Esempi di definizioni di funzioni

```
prompt> (defun square (x) (* x x))
square

prompt> (square 5)
25

prompt> (defun sum-of-square(x, y) (+ (square x) (square y)))
sum-of-square

prompt> (sum-of-square 3 4)
25
```

- il valore degli altri nomi sono gli oggetti associati a quel nome nell'ambiente definito nel programma Lisp

Questo concetto per valutare le espressioni non si applica agli operatori speciali, come ad esempio defparameter.

In Lisp si hanno come dati e procedure predefinite il seguente elenco di elementi:

- numeri interi, numeri in virgola mobile (es. 3.5 o $6.02E+21$), numeri razionali (es. $-3/42$) o numeri complessi ($\#C(01)$)
- booleani T e NIL
- stringhe (es "*sono una stringa*")
- operatori sui booleani *null*, *and*, *or*, *not*
- funzioni sui numeri + - / * *mod sin cos sqrt tan atan plusp* > <= *zerop*

Per definire nuove funzioni, per fornire una forma di astrattezza e fornire il nome ad una serie di operazione, in Lisp si usa il comando

```
defun (nome (argomenti) (corpo funzione))
```

: Ovviamente questa possibilità di definire nuove funzioni può essere combinato per creare in maniera semplice nuove funzioni, come si può notare nel listato di esempio: Per valutare una espressione combinata dove un operatore indica una funzione composta, definita tramite *defun*, l'interprete Lisp segue lo stesso processo visto per valutare le funzioni/operazioni primitive, infatti l'interprete valutano gli elementi presenti nell'espressioni e poi applicano la procedura agli argomenti, indicati dagli operandi nell'espressione.
Per applicare una funzione composta agli argomenti si valuta il corpo della procedura in cui ogni parametro formale viene sostituito dall'argomento corrispondente e per illustrare il processo vediamo come viene valutata la seguente espressione

Il processo descritto e mostrato nel seguente esempio viene chiamato **substitution model** e lo scopo di questo processo è quello di aiutare a pensare a come si applica alla procedura, ma non fornisce una descrizione fedele di come un interprete realmente lavora.

Inserire differenza tra applicazione e ordine normale

La potenza espressiva delle funzioni definite fino ad ora è limitata, dato che non abbiamo ancora la possibilità di effettuare delle scelte e per risolvere introduciamo ora il costrutto speciale **cond**$(cond\ (p_1\ e_1),\ \ldots\ (p_n\ e_n))$: consiste in una coppia di espressioni, dove il primo elemento è sempre un predicato "logico", come si può notare nel listato di esempio.
L'operatore cond verifica ogni coppia di espressioni e se il risultato è T ritorna il

```
prompt> (and (> 42 0) (< -42 0))
T

prompt> (not (> 42 0))
NIL

prompt> (and)
T

prompt> (or)
NIL
```

```
(defun valore-assoluto (x)
   (cond ((> x 0) x)
         ((= x 0) 0)
         ((< x 0) (- x))))

prompt> (valore-assoluto 3)
3

prompt> (valore-assoluto -42)
42
```

valore della seconda espressione altrimenti passa alla coppia successiva fino a che se non ci sono più coppie valutabili restituisce NIL.
In aggiunta ai predicati aritmetici, < > =, per effettuare dei confronti si possono utilizzare i predicati **and, or, not**, definiti come si nota nel seguente listato:

Per rappresentare il valore assoluto, come si vede nel listato Y, si poteva utilizzare anche il predicato

```
(if <predicate> <conseguent> <alternative>)
```

, applicabile quando si hanno solo due casi da analizzare.
Per valutare il predicato *if* l'interprete inizia a valutare la parte <predicate> dell'espressione: in caso viene valutata con *T* allora l'interprete valuta la parte <conseguent> e ritornaa il suo valore, altrimenti viene valutato la parte <alternative> e ritorna il suo valore.

```
prompt> (define (abs x)
           (if (< x 0)
           (-x)
            x))
```