Linguaggi di Programmazione

DaveRhapsody

30 Settembre 2019

Indice

Capitolo 1

Introduzione al corso

1.1 Programma del corso

Il corso è volto ad insegnare dei paradigmi di programmazione dei seguenti tipi:

1.1.1 Logica Matematica e Linguaggi logici (Prolog)

Termini, fatti(predicati), regole, unificazione, procedura di risoluzione

1.1.2 Linguaggi funzionali e Lisp (et al.)

Atomi, liste, funzioni e ricorsione

1.1.3 Linguaggi imperativi

Memoria, stato, assegnamenti, puntatori

Il concetto è che con questo corso si vanno a studiare paradigmi più evoluti, usati tutt'ora e comunque aventi un ampio approccio logico, oltretutto LISP è usato nelle pagine web (Si userà moltissimo la ricorsione, A I U T O)

1.2 Modalità d'esame

- il voto finale sarà una media pesata dei voti conseguiti nell'esame relativo alla parte teorica e nell'esame del progetto
  - Occhio, il peso è a discrezione dei prof

1.2.1 Prove parziali

Le prove d'esame sono costituite da uno scritto di 6-10 domande, e da un progetto da consegnare entro una data prefissata

1.3 Appelli regolari

Gli appelli regolari sono composti da un progetto ed un esame scritto, che può essere seguito da un esame orale a discrezione del docente basato sui temi trattati durante il corso
NON C'E' POSSIBILITA' DI RECUPERI, infatti scritto, orale e progetto vanno sostenuti NELLO STESSO APPELLO
Progetto e scritto sono corretti separatamente

NON CI SARANNO ECCEZIONI Lo avete già letto nel passaggio precedente, ma lo ripeto lo stesso perchè deve essere chiaro che N O N S I F A N N O E C C E Z I O N I.

Capitolo 2

Il paradigma

2.1 Cos'è?

E' il metodo di soluzione ad un determinato problema, a seconda dei paradigmi si hanno diversi tipi di linguaggi di programmazione

2.1.1 Storicamente

Il primo paradigma è l'imperativo, cioè il paradigma basato sui tre costrutti di selezione, iterazione e sequenza.
Inoltre si mantiene il concetto di assegnamento di un valore ad una determinata variabile

2.1.2 L'effetto collaterale

Viene definito effetto collaterale quando, a seguito dell'esecuzione di un qualsiasi codice, il contenuto di un'area di memoria viene cambiato; per intenderci, anche solo l'istruzione "x += 1" genera un effetto collaterale, poichè nell'area di memoria di x viene cambiato il valore. Perchè è importante tutto ciò, direte. Semplice: il paradigma puro funzionale si basa proprio sul fatto che un programma non generi mai, mai, *M A I*, effetti collaterali. Successivamente vedremo che in Prolog ci saranno parecchi problemi se provassimo ad assegnare direttamente un valore ad una variabile

2.2 Logica del primo ordine

Prolog è costituito da una serie di clausole derivanti dalla logica del primo ordine

2.3 Linguaggi funzionali

Questi si basano proprio sui concetti matematici di funzione, ad esempio si ragiona sui domini, sui codomini, sull'insiemistica, solite cose. La loro caratteristica è che ogni funzione, dato sempre lo stesso input, restituisce sempre lo stesso risultato. Cosa vuol dire questo? Che non dipende da variabili esterne (e da qui l'importanza degli effetti collaterali, evitati nei linguaggi funzionali).

2.4 Paradigma imperativo

Le caratteristiche essenziali dei linguaggi imperativi sono legate all'architettura di Von Neumann, costituita dai famosi due componenti **Memoria (componente passiva) e Processore (componente attiva)**
In pratica la principale attività che ha la cpu è quella di eseguire calcoli ed assegnare valori alle variabili, che sono delle celle di memoria.

Va considerato Il concetto di variabile è un'astrazione di una cella di memoria, per dire se giochi su assembly vai a toccare i veri e propri registri, mentre su C o Assembly si ragiona per nome di variabile, non vai di indirizzamento fisico

2.4.1 Il concetto di variabile

In Prolog e LISP cambia completamente il concetto di variabile, ma per come saranno presentati vedremo che non c'entra niente.
In matematica abbiamo il concetto di variabile? Sì, quella che sta dentro una funzione, in informatica è diciamo diverso, non è un'astrazione, ma lo vedremo in seguito

2.5 Modello di Von Neumann

Per manipolare la memoria utilizzo la variabile, simbolo che indica la cella di memoria, nei linguaggi funzionali sarà possibile usare il concetto di variabile matematica.

Alla fine il modello di Neumann è composto da I/O, Memoria e CPU con i suoi cicli di clock

2.6 Stile prescrittivo

Un programma scritto in un linguaggio imperativo prescrive le operazioni che la CPU deve eseguire per modificare lo stato di un sistema

Le istruzioni sono eseguite nell'ordine in cui queste appaiono, ad eccezione delle strutture di controllo

Realizzati sia attraverso interpretazione che compilazione, nati più per manipolazione numerica che simbolica.

2.7 Concetto di programma

Un programma è intendibile come un insieme di algoritmi e di strutture dati ma la struttura di un programma consiste in

- Una parte dichiarativa in cui son presenti le dichiarazioni di tutte le variabili del programma e del loro tipo

- Una parte che descrive l'algoritmo risolutivo utilizzato, mediante istruzioni del linguaggio

2.8 Perchè utilizzare paradigmi diversi?

Per esempio l'intelligenza artificiale si sviluppa su linguaggi di programmazione specifici, bisogna usare linguaggi che operino in un determinato modo, considerati tipo di Altissimo super mega galatticissifantastico livello infatti, utilizzabili pure da non programatori

Infatti son generati per manipolazione simbolica non numerica

2.9 Paradigma logico

Concetto primitivo: Deduzione logica, avente una base di logica formale e un obbiettivo, che è intendibile come formalizzazione del ragionamento

Programmare infatti significa descrivere il problema con frasi (Formule logiche) del linguaggio,
Interrogare il sistema, che effettua deduzioni in base alla "conoscenza rappresentata"

Ai lettori Mi rendo conto che non si capisca un cazzo, voi immaginatevi come mi stia sentendo al momento io mentre prendo appunti.. Perdonatemi

Prolog è un insieme di formule ben formate, ragiona con il linguaggio logico, con una descrizione della realtà di interesse, di fatto è una dimostrazione in un linguaggio logico che costituisce un programma. Più semplicemente ho una frase da dare al mio interprete, Prolog icchè fa? Semplicemente la realizza sotto forma di dimostrazione.

2.10 Esempio di un programma Prolog

Ci sono fondamentalmente:

- Asserzioni incondizionate (fatti) A.
- Asserzioni condizionate (regole) A :- B, C, D, ... , Z.
  - A è la conclusione o conseguente (deve avere una sola clausola)
  - B, C, D, . . . ,Z sono le premesse o antecedenti
- Un'interrogazione ha la forma: :- K, L, M, . . . , P.

Ovviamente A, B, C, *TUTTE LE ALTRE*, sono semplicemente predicati
MI RACCOMANDO MASSIMA ATTENZIONE ALLA SINTASSI, ogni clausola Prolog termina con un punto.

La ',' si legge come AND

2.10.1 Esempio:

Due individui sono colleghi se lavorano per la stessa ditta/azienda Regole Fatti Interrogazione

```
collega(X, Y) :-
        lavora(X, Z),
        lavora(Y, Z),
        diverso(X, Y).

lavora(ciro, ibm).
lavora(ugo, ibm).
lavora(olivia, samsung).
lavora(ernesto, olivetti).
lavora(enrica, samsung).

:- collega(X, Y).
```

Programmare Prolog non è come scrivere in un linguaggio di programmazione, non si scrive un algoritmo, in questo caso abbiamo le famose clausole, (regole e fatti),

ATTENZIONE l'interrogazione non è una clausola, occhio a non confondersi

2.10.2 Esempio dell'ordine di una lista

- **Ordine prescrittivo**: Controlla se la lista è vuota, e dà come risultato la lista vuota stessa, altrimenti calcola una permutazione della lista e controlla se è ordinata, dando come risultato L_1 altrimenti fa una permutazione su L etc.
  Il programmatore deve specificare le istruzioni che generano la sequenza di permutazioni della lista L

  $$\begin{cases} il\ risultato\ dell'ordinamento\ di\ una\ lista\ vuota\ \ la\ lista\ vuota \\ Il\ risultato\ dell'ordinamento\ di\ una\ lista\ L\ e\ L_1 \end{cases}$$

  Quindi, lo stile prescrittivo presuppone che OGNI SINGOLO CASO venga considerato e programmato. non esiste il "ma è ovvio che debba fare questo", ogni singolo caso è tua responsabilità. Sì, tua, proprio tu che stai leggendo. Prolog si basa su questo tipo di ordine.

- **Stile Dichiarativo**: L'ambiente si fa carico di generare possibili permutazioni della lista L, secondo deduzione matematica

2.11 Paradigma funzionale

- Si basa sul concetto di funzione matematica, ossia una associazione tra due insiemi che relaziona ad ogni elemento di un insieme (dominio) un solo elemento di un altro insieme (codominio)

- La definizione di una funzione specifica dominio, codominio, e regola di associazione

- ESEMPIO:
  Incr: N $\rightarrow$ N
  Incr(x) - x + 1
- Dopo aver dato definizione, una funzione è aopplicabile ad un elemento del dominio (argomento) per restituire l'elemento del codominio ad esso associato (valutazione)
- incr(3) $\rightarrow$ 4 L'unica operazione utilizzata nel funzionale è l'applicazione di funzioni
- Il ruolo dell'esecutore di un linguaggio funzionale si esaurisce nel calutare l'applicazione di una funzione (il programma) e produrre un valore
- Nel paradigma funzionale puro il valore di una funzione è determinato dagli argomenti che riceve al momento della sua applicazione e non dallo stato del sistema rappresentato dall'insieme complessivo dei valori associati a variabili(e/o locazioni di memoria) in quel momento
- Oggettivamente si ha l'assenza di effetti collaterali

Attenzione Il concetto di variabile che utilizziamo è quello di "costante" matematica, in cui i valori NON sono mutabili, non ho nessun asegnamento

L'essenza della programmazione funzionale consiste nel combinare funzioni mediante composizione e uso della ricorsione

2.11.1 Composizione di funzioni + ricorsione

La struttura di un programma consiste nella definizione di un insieme di funzioni ricorsive mutualmente

L'esecuzione del programma consiste nella VALUTAZIONE dell'APPLICAZIONE di una funzione principale a una serie di argomenti

2.12 LISP

LISt Processing,

Il progretto originale era di creare un linguaggio funzionale puro, infatti nel corso degli anni sono stati sviluppati molti ambienti di sviluppo lisp di cui terremo in considerazione Common Lisp e Scheme, oltre che emacs etc.

2.12.1 Esempio di programma LISP

Controlla un elemento se appartiene ad una lista

```
(defun member (item list))
(cond ((null list) nil)
((equal item (first list)) T)
(T (member item (rest list))))
(member 42 (list 12 34 42))
```

Dopo una parentesi tonda ci va per forza una funzione, è fondamentale, defun definisce una funzione infatti, dopo c'è il nome di tale funzione. In LISP la tabulazione è ESSENZIALE, AUGURI A DISTINGUERE DOVE PORTI LA QUINTA PARENTESI DELLE DODICI CHE HAI SCRITTO PIANGENDO SUL CODICE ALLE 2 DI NOTTE.
Gli elementi si separano con lo spazio NON con la virgola, mi raccomando.
La terza riga è la più ostica e dice: è uguale T al primo elemento della lista? E' molto incastrato ma si riesce a capire, associamo per esempio ad item = 2 (E' un esempio) e list = [1,2]
Noi ci arriviamo con la logica che c'è, ma in realtà cosa faremo? Ragioniamo per gradi

1. null list? false
2. è 2 = al primo elemento della lista? False
3. è 2 = al secondo elemento della lista? True

Se LISP trova T è come se scrivessimo **true**

2.13 Ambienti RunTime di linguaggi logici funzionali e non

- Richiami di nozioni di architettura e programmazione
- Per eseguire un programma di qualsiasi linguaggio il sistema operativo deve mettere a disposizione l'ambiente runtime che dia almeno due funzioni
  - Mantenimento dello stato della computazione(pc, limiti di memoria)
  - Gestione memoria disponibile (fisica e virtuale)
- L'ambiente runtime può essere una vera e propria macchina virtuale tipo la JVM di java
- In particolare la gestione di memoria avviene usando due aree concettualmente ben distinte con funzioni diverse
  - Lo stack, ambiente dell'ambiente runtime che serve per la gestione delle chiamate, a procedure metodi etc
  - L'heap dell'ambiente runtime serve per gestire strutture dinamiche
    * Alberi
    * Liste etc

2.13.1 Activation frame

- La valutazione di procedure avviene mediante la costruzione sullo stack di sistema di activation frames
- i parametri formali di una procedura vengono associati ai valori (si passa tutto per valore, non esistono effetti collaterali)
- E' un altro modo di chiamare i record di attivazione, via
- Il corpo della procedura viene valutato (ricorsivamente) tenendo questi legami in maniera statica

cioè il concetto è che bisogna capire cosa accade con variabili che risultino libere in una sottoespressione

2.14 Activation Frame di una funzione

Contiene:

- Return address
- Registri
- Static / Dynamic link (lo statico punta alle variabili globali)
- Argomenti
- Local definitions (RV)

Se si ha in mente come funziona oggettivamente lo stack (con i record di attivazione) è la stessa cosa
All'esame si potrebbe chiedere cos'è l'activation frame e a icchè serve

2.15 Heap e Garbage Collector

L'heap è l'area di memoria destinata alla memorizzazione delle strutture per i dati dinamiche, mentre invece il garbage collector ha il compito di accumulare lo schifo che si accumula tra variabili non deallocate etc, e le dealloca appunto.

Capitolo 3

Logica e ragionamento

Partiamo con le cose semplici, bisogna passare da quello che è un linguaggio parlato a una stesura di condizioni
Prendiamo un triangolo, vogliamo dimostrare che se due triangoli hanno i due lati uguali allora è isoscele

$$AB = BC \vdash \angle A \angle C$$

1. AB = BC per ipotesi
2. ABH = HBC per (Ipotesi 3)
3. Il triangolo HBC è uguale al triangolo HBC ABH per (Ipotesi 2)
4. A e C per (Ipotesi 1)

3.0.1 Regole di inferenza

Esempio di una regola di inferenza:

$$\frac{F_1, F_2, F_3, ..., F_n}{R}$$

1. Introduzione della congiunzione (L'AND)
2. Modus Ponens
3. Eliminazione della congiunzione

Come lavora il **Modus Ponens**:
E' semplice manipolazione sintattica, osservando la formula che ci vien data possiamo riscriverla scrivendo come base di conoscenza il conseguente, cioè in pratica prendo e sostituisco con il conseguente.

Per far si che le mie formule siano vere, se avessi un A or B può esser vera in ben tre casi diversi, non posso eliminare i due casi disgiunti, se voglio mantenere una solidità non posso, e quindi questa regola (disgiunzione) non esiste.

Attenzione in una dimostrazione non si può dare nulla per scontato, tutto ciò che noi diamo per assodato, un pc non lo dà, dobbiamo essere molto precisi nelle indicazioni, bisogna lavorare in un'ottica più precisa

cerchiamo ora di tradurre tutto in un linguaggio più formale SE AB = BC E BH = BH e ABH = HBC allora il triangolo ABH è uguale a HBC ed abbiamo trasformato (1) in

SE triangolo ABH è uguale al triangolo HBC ALLORA AB = BC e BH = BH e AH = HC, E ABH = HBC E AHB = CHB E A = C

Da un punto di vista formale noi partiamo da un'ipotesi, noi vogliamo dimostrare che A = C.

La dimostrazione è un processo sintattico, non ragiono in termini di verità, perchè NON si sta parlando di interpretazione ma manipolazione delle formula

Ogni passo deve corrispondere ad una formula, e subito dopo le etichette

Differenza tra assiomi e ipotesi Gli assiomi sono conoscenza pregressa del dominio, mentre le ipotesi sono solo supposizioni iniziali, uno si specifica, l'altro no

3.1 Dimostrazione

E' una sequenza di passi dove il finale è la formula da dimostrare e abbiamo un insieme di passi intermedi che possono essere presi dalle conoscenze pregresse, oppure applicando regole di inferenza ai passi PRECEDENTI, solo precedenti mi raccomando.

Le regole di inferenza sono applicabili solo ai passi precedenti rispetto ad una formula

3.2 Logica Proposizionale

Nella logica proposizionale ci si occupa delle conclusioni che possiamo trarre da un insieme di proposizioni, abbiamo infatti un insieme P di proposizioni

Si introduce il concetto di **interpretazione** di un insieme di proposizioni, infatti all'insieme P si associa una funzione di verità (True e False)

Questa funzione associa un valore di verità ad ogni elemento di P, ad ogni proposizione. La valutazione è il ponte tra sintassi e semantica di un linguaggio
Posso derivare sintatticamente una formula, e se ho consistenza delle supposizione avere comunque una formula.

Chiaramente posso legare tra loro le proposizioni con $\vee\wedge$ e $\neg$ Una formula ben formata è un insieme di espressioni sintatticamente corrette di un linguaggio
In prolog le formule atomiche le chiameremo letterali, che possono esser positivi e negativi, e qui si richiamano i concetti di fondamenti della tabella di verità

Negli esercizi potremo usare

$$\frac{F_1, F_2, ..., F_K}{R}$$

E' la forma generale di una regola, sopra hai l'insieme delle formule vere tra le formule ben formate e R è la formula generata da "inserire" in FBF.

L'esempio di inferenza che si usa di solito è il Modus Ponens:

$$\frac{p \rightarrow q, p}{q}$$

Se il conseguente appare come formula negata, si negherà anche la formula originaria

Esempio:

$\frac{p \vee \neg q}{vero}$ Terzo escluso

$\frac{\neg \neg q}{q}$ eliminazione $\neg$

$\frac{p \wedge vero}{p}$ eliminazione $\vee$

$\frac{p \wedge \neg p}{q}$ contraddizione

3.3 Principio di risoluzione

E' una regola di inferenza generalizzata semplice e facile da utilizzare ed implementare, in pratica opera su formule ben formate trasformate in forme normali congiunte, ed ognuno dei congiunti vien detto **clausola**

L'osservazione fondamentale alla base del principio di risoluzione è un'estensione della nozione di rimozione dell'implicazione su base della contraddizione

In pratica si usa per le dimostrazioni per assurdo.

3.4 Unit Resolution

E' un caso particolare di principio di soluzione;

Da un lato ho una formula ben formata disgiuntiva e dall'altro ho un letterale (o asserito), e una di queste formule è costituita da un solo letterale, per questo si chiama unit, è sintassi, non ci sto capendo più un cazzo pure. io, non so davvero che dirvi..

Spiegazione fornita da Jacopo De Angelis

Eccomi, arrivo da autore esterno a spiegare: prendiamo prima lo schemino semplice semplice
$P \leftarrow A \wedge B \wedge C \wedge D...$
Questo vuol dire che P è vera solo se tutte A, B, C e D sono vere. Ora, prendiamo nuovamente il codice prolog visto all'inizio:
collega(X, Y) :-
lavora(X, Z),
lavora(Y, Z),
diverso(X, Y).
Cosa cambia dal dire questo o la regola sopra? La corrispondenza è semplicemente:
P è collega(X, Y) :-
A è lavora(X, Z),
B è lavora(Y, Z),
C è diverso(X, Y).
Questo vuol dire che la nostra unit, P, è vera solo se sono vere le altre. Fine del mio intervento, la linea di nuovo a Dave.

Esempio:

- <non piove>, <piove o c'è il sole>
- <c'è il sole>

La dimostrazione per assurdo di fatto funziona assumendo che la formula negata sia vera, se combinandola con le proposizioni in fbf ottengo una contraddizione , allora si pul concludere con la verità della proposizione.

3.4.1 Esempio di dimostrazione per assurdo

Abbiamo una proposizione λ e dobbiamo dimostrare che essa sia vera:
Per dimostrarlo occorre porre per ipotesi che $\neg\lambda$ sia vera e SE FBF U $\neg\lambda$ genera una contraddizione, ALLORA λ è vera!

3.5 Ricapitolando

Quello che noi definiamo come "Calcolo Logico" delle proposizioni va a toccare

- Dal punto di vista della sintassi
  - Un insieme di proposizioni che chiameremo **P**
  - Un insieme di **FBF**, tale che **P**$\subseteq$ **FBF**
  - Un sottoinsieme di assiomi **A** $\subseteq$ **FBF**
  - Un insieme di regole di inferenza che ci permettono di incrementare **FBF**
- Dal punto di vista della Semantica
  - Una funzione di verità che consente di distinguere **true** e **false** rispetto alle tavole di verità o funzioni di interpretazione

3.6 L'assioma

Un assioma è una conoscenza che si da per assodata, qualcosa di sicuramente vero, se vogliamo anche "scontato"

3.6.1 L'esempio dell'unicorno

Se l'unicorno è mitico, allora è immortale, ma se non è mitico allora è mortale. Se è mortale o immortale allora è cornuto. L'unicorno è magico se è cornuto.

L'unicorno:

- E' mitico?
- E' magico?
- E' cornuto?

Procedimento

1. Esprimere il problema in forma di logica delle proposizioni
2. Individuare i teoremi da dimostrare
3. Dimostrare i teoremi

Cioè concettualmente devo cercare di individuare i predicati, e porli in modo più logico (Tipo mitico(x), magico(x) e cornuto(x)), MA MI RACCOMANDO ATTENZIONE.

NON CONFONDERE I termini a disposizione, se qui hai mortale o "Immortale" stai parlando di qualcosa che è Mortale $\vee\neg$ Mortale.
Ora andiamo a risolvere questo esercizio, e si comincia con il dare un nome ai nostri predicati, che siccome sarebbero lunghi, saranno abbreviati in massimo 5 predicati

- UM = Mitico
- UI = Immortale
- UMag = Magico
- UC = Cornuto

Ritrascriviamo quindi la nostra frase iniziale:
$UM \rightarrow UI$,
$\neg UM \rightarrow \neg\ UI$,
$\neg UI \vee UI \rightarrow UC$
$UC \rightarrow UMag$
Mentre per quanto riguarda le domande poste:

1. $S \vdash UM$?
2. $S \vdash UMag$?

3. S $\vdash$ UC?

Risolvendo l'esercizio otteniamo che Ora, il metodo per risolvere le $\vdash$ potrebbe essere tramite negazione della nostra ipotesi. In che senso? Proviamo con il caso S $\vdash$ UC

- P1: $\neg$ UI $\vee$ UI $\rightarrow$ UC (Da S)
- P2: $\neg$ UI $\vee$ UI (Che era stato dimostrato in un passaggio precedente con la dimostrazione di A$\rightarrow$(B$\rightarrow$C))
- P3: UC (Da P1, P2 e modus ponens)

Il compitino avrà un esercizio di questo livello di difficoltà che consisterà in una dimostrazione, che ovviamente va per assurdo.

3.6.2 Tautologie e modelli

Una fbf che si verifica in ogni caso, è detta **Tautologia** (Fondamenti docet)
Una particolare **Interpretazione** V che rende vere tutte le formule in **S** viene detta **modello** di S

3.7 La logica del primo ordine

Se la logica proposizionale si dimostra utile, avente caratteristiche di computazione che da questo punto di vista sono chiare, c'è da dire che la semantica è chiara allo stesso modo, però purtroppo non ci permette di fare asserzioni in merito ad insieme di elementi in maniera concisa.

Ai lettori Cioè io sto capendo logica adesso e non l'ho capita a Fondamenti.. Meditiamo ragazzuoli, meditiamo.
Sono Jacopo, vi parlo dal terzo anno. Fondamenti si capisce SOLO tramite gli altri corsi.

Con la **Logica del primo Ordine** introduciamo modi diversi di esprimere le proposizioni, prendiamo per esempio Socrate (Ricordate Palmonari? Ecco)

- Tutti gli uomini sono mortali
- Socrate è un uomo
- dalle precedenti ipotesi si deduce che Socrate è mortale

C'è un problemino di fondo, non si può esprimere in alcun modo qualcosa del tipo "Tutti gli uomini sono mortali"

Un linguaggio logico del primo ordine è costituito da **termini** che in pratica si costruiscono con

- V: insieme di simboli di variabili
- C: insieme di simboli di costante
- R: insieme di simboli di relazione o predicati (di qualsiasi arietà)
- F: insieme di simboli di funzione (di qualsiasi arietà)

Ah, giusto L'arietà sarebbe il numero di argomenti di una relazione o predicato. (Chiaramente anche funzioni, in quanto caso particolare di relazione)

Inoltre si hanno i connettivi logici, ovvero $\forall$ (Per ogni) e $\exists$ (Esiste).

In Prolog ci sarà un uso implicito dei quantificatori, è implicita la congiunzione, l'universale non è specificato ma è sottointeso, tutte le formule valgono assumendosi la quantificazione universale.

Una sola causa E' quantificata in modo esistenziale, la query. Se Però questa è una cosa che si gestisce l'interprete, noi non ci si fa problemi da questo punto di vista

Attenzione, in prolog non bisogna usare in modo intercambiabile simboli di predicato con quelli di funzione, perchè supponiamo la funzione "successore di un numero". La funzione successore dato un valore numerico λ, successore mi darà $\lambda + 1$.
In questo caso ha simbolo di funzione, il predicato ti dà o vero o falso, pochi cazzi.

Se ho Successore(a, λ), questo mi dà true o false, è λ successore di a? Se sì mi dà true o altrimenti mi dà false. Questa è la differenza, oltre al fatto che un predicato ragiona sul singolo argomento.

Postilla simpatica: Quello che chiamo Modus Ponens, non è altro che un modo più simpatico di chiamare l'eliminazione dell'implicazione, che è quello che facevamo a fondamenti, ma chiamiamolo modus ponens, perchè sì.

Con la logica di primo ordine si ragiona diversamente, infatti il linguaggio è costruito ricorsivamente, ed i termini minimi sono detti **PREDICATI**

3.7.1 Le Formule Ben Formate nella logica del primo ordine

Qui si fa menzione della definizione **Ricorsiva** di Formula Ben Formata cioè:

$$FBF = \{t_j, r_{t_1,...,t_k}\}$$

Dove t_j è un termine elemento di C, di V, oppure un'applicazione di una funzione f($t_1, ..., t_s$) mentre

$$r(t_1, ..., t_k)$$

Che sarebbe un termine costituito da un predicato (dove le t derivano dai termini appartenenti alle FBF)

Contiamo che diversi elementi di FBF connessi dai connettivi $\forall, \exists \neg, \rightarrow$ appartengono ad FBF

Si denota t($t_1, ..., t_s$) tale combinazione di termini

Grazie alle definizioni precedenti possiamo andare a risolvere l'esempio di Socrate.
Iniziamo dalle cose semplici, definiamo chi sono le costanti (insieme C)→
Socrate è una costante, pertanto apparterrà all'insieme C: C = {Socrate}, e poi possiamo definire quello che sono i predicati, che sono in questo caso uomo e mortale.

I predicati appartengono all'insieme R; R = { uomo. mortale}

Benissimo, proviamo ora a realizzare la frase "Tutti gli uomini sono mortali":
Se tutti gli uomini son mortali, significa che per ogni elemento x tale che esso sia un uomo, si implica che x sia mortale.

Osservazione Quando dico "Sia un uomo" e "Sia mortale", intendo che la funzione uomo(x) e mortale(x) siano tendenzialmente delle booleane

Infatti ritrascrivendolo vien fuori: $\forall$ x, (uomo(x)→mortale(x)), tenendo conto che Socrate è un uomo, pertanto si dirà uomo(Socrate), che se pensate al booleano, sì, darà **true** .

3.7.2 Calcoli logici

Per ottenere il risultato è necessario che si utilizzino delle regole di calcolo

Regola di eliminazione del quantificatore universale

Il discorso è: Come si risolve il $\forall$?

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

Si ok, ma come lo realizziamo?
Con questa super mega iper formula possiamo finalmente derivare la nostra conclusione a partire dalle asserzioni iniziali.

1. uomo(Socrate)
2. $\forall$x, uomo(x) $\rightarrow$ mortale(x)
3. mortale(Socrate)

Andiamo a scrivercela sostituendo alla formula $\rightarrow$

$$\frac{(\forall x, uomo(x) \rightarrow mortale(x)), Socrate \in C}{uomo(Socrate) \rightarrow mortale(Socrate)}$$

Ora andiamo a togliere il quantificatore universale (∀ detto in modo figo)

$$\frac{uomo(Socrate), uomo(Socrate) \rightarrow mortale(Socrate)}{mortale(Socrate)}$$

Cosa notiamo? Il "denominatore" della prima formula è diventato il "numeratore" della seconda, infatti vedete come abbiamo uomo(Socrate) → mortale(Socrate)?

A questo punto bisogna togliere anche l'implicazione, o meglio risolverla. Riflettendoci $A \rightarrow B$ a cosa è uguale? $\neg A \wedge \neg B$, in questo modo abbiamo risolto l'implicazione.

3.8 Altre regole in Logica del Primo Ordine

Abbiamo risolto il ∀, ok, ma l'∃? Introduciamo quindi il quantificatore esistenziale:

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

Per completezza possiamo dedurre il fatto che

$$\begin{cases} Se\ \exists x, \neg T(..., x, ...) = \neg \forall x, T(..., x, ...) \\ Se\ \forall x, \neg T(..., x, ...) = \neg \exists x, T(..., x, ...) \end{cases}$$

Spiegato peggio, se dico che esiste un elemento, per cui non si verifica una proprietà, allora non per ogni elemento essa si verifica.
Spiegato ancora peggio, se uno solo non si verifica, allora significa che non tutte si verificano

Nel secondo caso invece se dico che per ogni x una proprietà non si verifica, allora non esiste alcun x per cui si verifica

Precisazione In effetti negare il ∀ si tradurrebbe in "non tutte le x", mentre è molto più semplice per l'∃ in cui si dice "non esiste", lo specifico perchè io mi ci confondevo spesso.

Capitolo 4

Prolog

Prima ancora di cominciare: Siccome mi seccava scaricarmi tutto l'ambiente di sviluppo Swi-Prolog, vi lascio qui sotto il link per avere il compilatore direttamente online che OVVIAMENTE non è utilizzabile all'esame, però fa comodo, ve lo assicuro. Link al magico sito

---

Data questa confusissima introduzione giungiamo a trattare più nello specifico il linguaggio Prolog. Quando parliamo di Prolog, infatti, smettiamo di ragionare in modo **imperativo** per passare al paradigma di **programmazione logica** .

Premessa: C'è (come lo è stato per Prog 1) da scaricarsi l'interprete di Prolog, si consiglia SWI-Prolog, che si appoggia ad Emacs come editor di testo.

(Sia per la stesura di questi appunti che per gli esercizi io ho sempre usato Sublime Text MA all'esame molto probabilmente dovremo usare quel che ci impongono loro, pertanto, meglio abituarsi da subito ad Emacs.)

4.1 Programmazione logica

Se prolog utilizza il paradigma di **programmazione logica** , quest'ultima non è rappresentata solo da Prolog, ci sono ovviamente altri linguaggi che lo fanno. Perchè scegliere Prolog?

- Il formalismo è più semplice
- E' un linguaggio ad alto livello
- La semantica è comprensibile

Quello che è il nostro programma divente un insieme di formule ed ha un enorme potere espressivo, mantenendo come chiave il fatto che la computazione effettiva è costruzione di una dimostrazione di una affermazione (Definita anche come goal, obbiettivo, meta).

4.1.1 Logica Matematica

Definizione Davis e Putnam: Per logica matematica si intende la dimostrazione automatica di teoremi, secondo Davis e Putnam la logica matematica implica la dimostrazione dei teoremi

Definizione Kowalski: Interpretazione procedurale di formule. Quindi in pratica qua si entra già più nel concetto di un linguaggio di programmazione.

4.2 Cos'è ProLog

Nel titolo di questa sezione la L maiuscola non è a caso, perchè Pro-Log sarebbe acronimo di **PRO**gramming **LOG**ic, ed è un linguaggio che si basa su una restrizione della logica del primo ordine (FOL).

4.2.1 Ambiti di applicazione

Generalmente prolog è utilizzato come linguaggio per gestire i Database, tendenzialmente alcuni DBMS (Database Management System) sono programmati

4.2.2 Caratteristiche di Prolog

- Si basa su una restrizione della logica del primo ordine
- Ha uno stile dichiarativo
- E' usato per determinare quando una affermazione è vera e quali vincoli abbiano fatto da discriminante tipo i vincoli sui valori da dare alle variabili che han generato la risposta

4.3 Formule Ben Formate e Forma Normale a clausole

Qualsiasi formula ben formata può essere riscritta in forma normale a clausole. Esistono due forme normali a clausole:

1. Forma congiunta:
   La formula è una congiunzione di disgiunzioni di predicati o negazioni di predicati (letterali positivi e letterali negativi)
   $$\bigwedge_i (\bigvee_j L_{ij})$$
2. Forma disgiunta:
   E' una disgiunzione di congiunzioni di predicati o negazioni di predicati (letterali positivi e letterali negativi)
   $$\bigvee_j (\bigwedge_i L_{ij})$$

4.3.1 Forma normale disgiuntiva

La clausole che hanno al più un solo letterale positivo (sia con che senza letterali negativi) si chiamano clausole di Horn

Spiegato meglio, perchè così non è chiaro in effetti, in pratica prendete una clausola a caso: $A \wedge B \vee \neg C$, è una clausola di Horn perchè c'è UN SOLO TERMINE (Letterale) che NON è

Negativo. (O **Negato**)

Precisazione: Se abbiamo un solo letterale positivo, esso è clausola di Horn pure non c'è nemmeno un letterale negativo, ne basta uno solo positivo.

Però occhio: Non tutte le formula ben formate si riescono a far diventare un insieme di clausole di Horn, che per la cronaca compongono i programmi Prolog o meglio, **I programmi prolog son collezioni di clausole di Horn**

Le suddette clausole rappresentano (lo si analizzerà successivamente) fatti, regole o interrogazioni (query/goals).

4.3.2 Linguaggio dichiarativo

Uno dei punti di forza di Prolog è il fatto che sia un linguaggio dichiarativo, quindi è pressochè esente dall'avere istruzioni, contiene solo fatti e regole, che dal passaggio precedente sappiamo essere delle **clausole di Horn** .

Ricordiamo che: Un fatto è una asserzione vera nel contesto che si descrive, tipo assioma, mentre la regola è qualcosa che serve per dedurre dei nuovi fatti **partendo** da quelli esistenti.

Un programma scritto in Prolog ci dà informazioni su un sistema, e vien chiamato base di conoscenza (Il programma eh, non il sistema.)

Inoltre, (*già detto in precedenza ma va ripetuto perchè sicuramente anche io stesso quando andrò a studiare da qua mi sarò già dimenticato*) un programma Prolog **non si esegue** , ma si **interroga**.

Quali sono le possibili domande da fargli? Ad esempio:

- Questa serie di fatti è vera?
  E la risposta sarà Si o No, True o False, 1 o 0, "Si broh" o "No frate".

Più precisamente questo sì sarebbe un "Sì, ho dimostrato il mio teorema per assurdo", perchè appunto ricordiamo che qua si sta parlando comunque di dimostrazione di teoremi

4.4 Sintassi Prolog

- Fatto/Asserzione: *nomefatto*.
- Regole: c :- $b_1, ..., b_n$.
- Goal/Query: ?-$q_1, ..., q_n$.

Inciso: Queste vanno scritte nel terminale, non nel programma. Sono tipo richieste che noi facciamo in base al momento per intenderci.

Quando l'interprete prolog si trova di fronte una query (o goal) esegue in sequenza la unit resolution andando in sequenza sulle sue clausole, dalla prima all'ultima, e le valuta tutte perchè potrei avere parecchie differenti soluzioni, posso dimostrare più teoremi.

Ogni lettera avente un pedice tipo p_λ o q_κ sono tutti termini composti, notare che in molte implementazioni il prompt Prolog è anche un operatore che chiede al sistema di valutare il goal

Ogni espressione Prolog diventa **TERMINE**, ne abbiamo diversi esempi:

- Atomi:
  E' una semplice sequenza di caratteri che inizia con carattere **Minuscolo** e può avere il '_', oppure è un numero, o qualcosa racchiusa tra apici (' ')
- Variabili
- COmposizioni di altri termini (Da qui termine composto)

NB. Tutto ciò riguarda la sintassi, questo elenco è legato al "Come si scrive questa determinata cosa in Prolog"

Precisazione Ogni istruzione finisce con il punto, avete presente quando la vostra fidanzata capite che è innervosita con voi e alla fine delle frasi ci mette un '.' che vi fa gelare il sangue? Ecco, se qui non mettete il punto dopo ogni istruzione, vedrete come vi gireranno i **Censura** ;))))

Ecco un esempio di comandi validi (LaTeX le metterà nella prossima pagina)

```
foo           hello
Hello         sam
sam           hello_Sam
hello-sam     40+2
quarantaquattro-4
'hello'       'Hello'
'Hello Sam'   _
a1            '1a'
X                      _hello
_234          hello(X)
f(a)          f(a, b, c)
f(hello, Sam) f( a, b, c )
p(f(a), b)
hello(1, hello(x, X, hello(sam)))
t(a, t(b, t(c, t(d, []))))
```

Come si nota sono un po' di comandi a caso, ma privi di errori sintattici

---

Vediamo ora invece un esempio di comandi NON validi

Non validi

```
hello Sam     Hello Sam
hello sam     1a
f(a, b        f(a,
f a, b)       f (a, b)
X(a, b)       1(a, b)
```

In questo caso come vedete sono errori che tendenzialmente era possibile fare pure su Java, non è che si tratti di chissà che di complesso.

4.4.1 Le variabili (logiche)

La variabile logica è una sequenza alfanumerica che però inizia con un carattere maiuscolo (oppure con l'Underscore _), e se son composte solo dal simbolo _ prendono il nome di Indifferenza o anonime.

Vengono instanziate (legate ad un valore) con il procedere del programma (Nella dimostrazione del teorema)

4.4.2 Termini Composti

In cosa consiste una composizione di termini? In un funtore (Simbolo funzione, o predicato definito come atom) + una sequenza di termini racchiusi tra parentesi tonde e separati da virgole. Questi ultimi sono come gli argomenti dei metodi che facevamo in java, stesso identico concetto.

Non serve nemmeno essere uno spazio tra funtore e parentesi di sinistra, per via di caratteristiche del sistema di parsing di prolog

4.5 Le Regole

Sono utilizzabili per esprimere definizioni:

- Se X è un animale ed X ha le squame, X è un pesce

Ma in modo informale una regola alla fine è una formula ben formata, ed è composta da una testa e da un corpo (collegate da un operatore) quindi per ipotesi, una possibile regola potrebbe essere A :- B, concettualmente si legge A è implicata da B.

La testa di una regola è il conseguente di una implicazione logica, cioè quello che consegue dal corpo, che è appunto l'antecedente.

A :- B

si può tradurre in $B \rightarrow A$, ora proviamo a ritradurre il nostro esempio del pesce:

```
pesce(X) :- animale(x), ha_le_squame(X)
```

In cui la virgola ovviamente è un and ($\wedge$)

Attenzione: già questo esempio presenta una vulnerabilità, nel senso che per intenderci, essere un pesce implica di aver le squame, MA essere un animale ed avere le squame non implica essere un pesce.

4.6 Regole di ricorsione

Proviamo a definire il concetto di un antenato, che è l'esempio più semplice per capire come funzionano le formule ricorsive. Ci conviene ancora ragionare sulle nostre definizioni

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

Traducendolo:
La prima riga si legge: **SE** X è genitore di Y **ALLORA** X è antenato di X.

La seconda riga invece è appena appena più complessa, merita un ritorno a capo solo per lei:

SE X è antenato di Z **E** Z è genitore di Y **ALLORA** X è antenato di Y

4.7 L'interprete prolog: Interrogazioni

Come detto in precedenza possiamo chiamarle interrogazioni, query, goal, non sono altro che comunicazioni dirette all'interprete che una volta fatto partire ci presenterà true o false.

Un esempio di query:

```
?- libro(kowalski, prolog).
```

Prolog ti restituirà come abbian detto prima true o false, yes o no, insomma ci siamo capiti.

Volendo io posso interrogare anche il programma usando delle variabili esistenziali. In pratica tutte le variabili Prolog le instanzia quando prova a darti una risposta, e queste vengono mostrate nella tua risposta, per esempio:
libro(AUTORE, prolog) si legge come: CHI E' L'AUTORE DI PROLOG? (Un folle di sicuro, ma va be'.)

4.8 Unificazione: introduzione

L'operazione di instanziazione di variabili durante la prova di un predicato è il risultato di una procedura particolare detta **Unificazione**

Dati due termini, la procedura di unificazione crea un insieme di sostituzioni delle variabili, e questo insieme permette di rendere uguali i due termini.

Tradizionalmente la procedura di unificazione costruisce un insieme di sostituzioni che prende il nome di "Most general unifier (indicato con MgU)" (No, fan della F1 non c'entra nulla con l'MGU-K o H :C)

Noi non faremo unificazioni, ci pensa l'interprete Prolog, che usa le unificazioni appunto. L'interprete Prolog stupido, che esegue algoritmicamente operazioni semplici, deve tenere

traccia di tutte le unificazioni, quindi rinomina le variabili che userà durante il processo,

Come testiamo questa cosa? Prendiamo le due espressioni che vogliamo testare, le poniamo con il simbolo uguale in mezzo, e ci darà il risultato. Possiamo chiedergli 42 = 42? e 42=x e ci dà yes in entrambi questi casi

4.8.1 Most General Unifier

E' il risultato finale della procedura di valutazione, o meglio, di prova del Prolog.
Il modo più comodo per vedere appunto come la procedura di unificazione funziona è di usare l'=

4.9 Le Liste in Prolog:

Si definisce una lista in Prolog racchiudendo gli elementi (termini e/o variabili logiche) della lista tra parentesi quadre e si separano con le virgole. Gli elementi di una lista in Prolog possono essere termini qualsiasi o liste.
[ ] indica la lista vuota.

Ogni lista non vuota può essere divisa in due parti, la testa e la coda, che sono rispettivamente il primo elemento e "tutti gli altri". Un po' come in F1, la testa è la Mercedes, la coda son tutte le altre squadre (O la Juve in Serie A negli ultimi anni).

4.10 L'operatore |

Si usa per separare e distinguere tra inizio e la coda di una lista. BONI, NON TRA LA TESTA E LA CODA, ma tra alcuni elementi e la nostra lista

Prolog lavora su strutture ad albero , infatti i programmi sono delle vere e proprie strutture per i dati, e sono manipolabili (mediante predicati extra-logici, cioè che vanno oltre a quelli del programma stesso)

Inoltre, già detto varie volte, Prolog è un linguaggio che sfrutta la **Ricorsione**, in cui manca una nozione semplice di assegnamento.

Se con gli algoritmi si studiava il metodo di soluzione di una serie di problemi, in questo caso ci si concentra direttamente sulla specifica del problema.

4.11 La differenza tra Clausola semplice e Clausola di Horn

Una possibile clausola può essere l'implicazione: $A \to B$, che è traducibile in $\neg(A \vee B)$ ed è leggibile in diversi modi, tutti coincidenti.

- B è **implicato** da A

- **Se** A **allora** B
- A **implica** B

Se invece di esserci A e B ci fossero due insiemi di κ e λ termini avremo che:

$$(A_1, A_2, ..., A_\kappa) \wedge \neg(B_1, B_2, ..., B_\lambda)$$

A e B assumono nome di letterali , e son positivi quando non presentano la negazione ($\neg$).

La clausola di Horn oggettivamente non è altro che una di queste clausole MA che ha MASSIMO, AL PIU', NON PIU' DI UN letterale positivo.
In Prolog alla fine quel che si verifica è questo:

- Fatti: A. (E' il caso limite in pratica)
- Regole: A :- $B_1, ..., B_n$
- Goals/Query/Interrogazioni: :- $B_1, ..., B_n$
  Nella trasformazione di DeMorgan viene negato perchè è una clausola da aggiungere, per questo si presenta con questa notazione, ma noi programmatori scriveremo un letterale positivo od una congiunzione di letterali positivi
- Contraddizioni: fail

4.12 Un programma logico

Sì, ok, ma con questo fantasmagorico linguaggio ci si può fare almeno c = a + b?

Sì, si può, (ed è più semplice farlo in assembly), e si deve ragionare pensando alla funzione "successore", che è una funzione in N, che consentirà di sviluppare un programma di questo tipo:

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

's' sta per **successore**, proviamo a leggerlo (ricorsivamente):

Il programma verrà interrogato in questo modo:
$\exists X$ sum(s(0), 0, X) {X / s(0)} $\exists X$ sum(s(s(0)), s(0), W) {W / s(s(s(0)))} MA Scritto in Prolog diventa

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

Spiegazione: Il modo in cui va ragionato è ricorsivo, nel senso, siccome si è nel campo dei naturali si ha uno 0, un centro, ed il nostro caso base chi è? L'elemento neutro rispetto alla somma (0). $\forall x \in N sum(0, X) = X$

Perchè ha 3 argomenti e non solo due? Semplicemente i primi due sono addendi ed il terzo è un risultato, nulla di che. Si legge così:

La somma di X + Y è uguale a Z SE la somma del successore di X + Y dà il successore di Z. Quindi? Se ho 3 + 4 = 7, allora 4 + 4 = 8. Cioè all'atto pratico richiami per Y volte il successore di X, che quindi chiamerà Y volte il successore di Z, e Z diventerà la somma effettiva dei nostri X + Y.

4.13 Sostituzioni

Già dall'esempio qui sopra si nota che W / s(s(s(0))) e N / s(0) sono due sostituzioni, ma in che senso? Praticamente ci dicono con quali valori (che potrebbero pure essere altre variabili) possiamo sostituire le variabili in un termine.

Di solito si denota una sostituzione con questo formalismo:

$$\sigma = \{X_1/v_1, X_2, /v_2, ..., X_\kappa/v_\kappa\}$$

Più nello specifico una sostituzione può essere considerata anche come *funzione* applicabile ad un termine: $\sigma : T \to T$ dove T è l'insieme di termini.

No, non è l'unico modo è un possibile modo, di fatto basta un comando del tipo sum(X,Y):- S is X+Y e ti fa la somma.

4.14 Esecuzione di un programma

Una computazione all'atto pratico è una dimostrazione (tramite le regole di risoluzione) del fatto che una formula sia verificabile (e che quindi sia un teorema).

Va determinata una sostituzione per le variabili della query per cui la query segue logicamente dal programma. Per esempio, se ho un programma **P** ed una query ": − $p(t_1, t_2, ..., t_m$" (Ricordandoci sempre che la query NON è dentro al programma, o comunque non è nel file sorgente del programma) allora:

$$\begin{cases} Se \;\; X_1, X_2, ..., X_n \; sono \; variabili \; che \; compaiono \; in \; t_1, t_2, ..., t_m \\ ALLORA \; \exists X_1, X_2, ..., X_n . p(t_1, t_2, ..., t_m) \end{cases}$$

Ed il nostro obbiettivo sarà quello di trovare una sostituzione del tipo:

$$s = \{X_1/s_1, X_2/s_2, ..., X_n/s_n\}$$

Da dove cicciano fuori queste s? Sono dei termini, per cui si ottiene che

$$P \vdash s[p(t_1, t_2, ..., t_m)]$$

E P cos'è? Un programma. E un programma cos'è? Un'insieme di clausole di Horn. Ed una clausola di Horn cos'è? E' una formula avente al più UN letterale positivo.

- Dato un certo insieme di clausle di Horn è possibile derivare la clausola vuota SE ce n'è almeno senza testa. **Tradotto:** Se abbiamo una query Q_0 da provare
- Si deve dimostrare che da $P \cup Q_0$ si possa derivare la clausola vuota $\Longrightarrow$ Sì, come al solito, mediante dimostrazioni per assurdo applicando il ***Magico principio di risoluzione***

Sì, ok, ma come?
C'è un problema di fondo: Se provassi tutte le risoluzioni possibili per ogni passo e aggiungessi le clausole inferite all'insieme di partenza avresti un' **ESPLOSIONE** combinatoria.

Per evitarsi questo problema va adottata una strategia di soluzione che sia opportuna (No, ci sarà sempre tra le palle il ***Magico principio di risoluzione*** , solo più vincolato.

4.15 Risoluzione ad INPUT LINEARE (SLD)

Prolog dimostra la veridicità o meno di una query con una sequenza di passi di risoluzione (Sì, sequenza di passi, algoritmo, efficienza, $\Theta nLogn$)

Infatti in Prolog la risoluzione avviene sempre tra l'ultima query derivata in ciascun passo e una una **clausola di programma**, ma non accade MAI tra due clausole di programma o fra una clausola ed un goal derivato precedentemente.

Riassumendo: Quello che si è appena descritto è definita **Definizione SLD** (Selection function for Linear and Definite sentences Resolution; in cui le "frasi lineari" sono essenzialmente delle clausole di Horn).

Esempio di risoluzione SLD :

- Partendo dalla Query $Q_i \equiv \ ? - A_{i,1}, A_{i,2}, ..., A_{i,m}$
- e dalla regola: $A_r : - \ B_{r,1}, B_{r,2}, ..., B_{r,m}$
- se esiste un unificatore σ tale che $\sigma[A_r] = \sigma[A_{i,1}]$
  Allora si otterrà una nuova query G_{i+1} tale che:

G_{i+1} si genera in pratica dal principio di risoluzione:

1. A_r
2. $\neg A_{i,1} \vee \neg A_{i,2} \vee ... \vee \neg A_{i,m}$
3. $A_{i,1} \vee \neg B_1, \neg B_2, ..., \neg B_m$
4. $\neg B_1 \vee \neg B_2 ... \vee \neg B_n \vee \neg A_{i,2}, \vee \neg A_{i,3}, ..., \vee \neg A_{i,m}$

$$G_{i+1} \equiv \ ? - B^{'}_{e,1}, B^{'}_{e,1}, ..., B^{'}_{e,1}, A^{'}_{e,1}, A^{'}_{e,1}, ..., A^{'}_{e,1}$$

E questo è solo **uno** dei passi di risoluzione eseguiti dal sistema Prolog, in cui

- Le $A^{'}$ e le $B^{'}$ sono risultati $\sigma[A] = A^{'}$ e $\sigma[B] = B^{'}$

Attenzione: La scelta di unificare la **prima** sottoQuery Q_i E' arbitraria (anche se comoda); Scegliere $A_{i,m} \vee A_{i,c}$ con $c \in [1, m]$ casuale sarebbe comunque accettato.

Se partissi invece dalla query $Q_i \equiv ? - \ A_{i,1}, A_{i,2}, ..., A_{i,m}$ e dalla regola (dal fatto) A_r, se esiste un unificatore σ tale che $\sigma[A_r] = \sigma[A_{i,1}]$, allora ottieni una nuova query $G_{i+1} \equiv \ ? - A^{'}_{e,1}, A^{'}_{e,1}, ..., A^{'}_{e,1}$, ovvero, la nuova query ha dimensioni minori rispetto alla precedente, avendo m-1 sotto-query.

Come già detto infatti nella risoluzione SLD , il passo di risoluzione avviene tra l'ultima query e una clausola di programma.

Osservazione: Per coloro che usano anche le slides noteranno che questa parte non è altro che una riscrittura delle suddette a parole di uno che le sta studiando, per dirvi, quello che nelle slides si chiama Goal lo chiamo query, ma quando una spiega con le slides, un velo di copiatura ci sarà sempre.
Che si chiami goal query o interrogazione, è sempre e comunque un teorema.

Come può essere il risultato finale?

- Successo
  Viene generata la clausola vuota, ovvero se per n finito Q_n è uguale alla clausola vuota $Q_n \equiv$:-
- insuccesso finito se per n finito Q_n non è uguale a :- e non è più possibile derivare un nuovo **risolvente** da Q_n ed è una clausola di 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; applicata alle variabili nei termini del goal iniziale dà la risposta finale.

Durante il processo di generazione di query intermedie si costituiscono delle varianti dei letterali e delle clausole coinvolte mediante la rinominazione di variabili

Una variante per una clausola C è la clausola $C^{'}$ ottenuta semplicemente rinominando le variabili di C (Renaming).

Esempio: p(X) :- q(X, g(Z)). è equivalente alla clausola con variabili con nomi diversi: p(A) :- q(A, q(B)). → Cambiando il nome delle variabili non ottieni nulla di diverso (a meno che esse nel programma sono più volte usate, come in Java eh) MA il codice potrebbe fare schifo da leggere.

Infatti possono esserci più clausole utilizzabili per applicare la risoluzione della query corrente, nello specifico ci sono due strategie di ricerca diverse che si possono adottare.

- Depth first: (In profondità) Si sceglie una clausola e la si mantiene fissa finchè non si arriva alla clausola vuota o all'impossibilità di nuove risoluzioni, ed in quest'ultimo caso si riconsiderano le scelte fatte in precedenza.
- Breadth First: (In ampiezza) Si considerano in parallelo tutte le alternative

Prolog adotta una strategia di risoluzione in profondità con **Backtracking** cioè in pratica

- Permette risparmio di memoria MA
- NON è **completa** per le clausole di Horn

4.16 Alberi di derivazione SLD

Dato un programma logico P (Insieme di clausole), ed una query Q_0 con una regola di calcolo R, un **albero SLD** per P $\cup\{Q_0\}$ via R, è definito sulla base del processo di prova visto precedentemente

- Ciascun nodo dell'albero è una query (Possibilmente vuota)
- La **radice** dell'albero è la query Q_0.
- Dato il nodo: :- $A_1, A_2, ..., A_{m-1}, A_m, A_{m+1}, ..., A_\kappa$, 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:

  - $C_i \equiv A_i$:- $B_{i,1}, ..., B_{i,q}$
  - $C_\kappa \equiv A_\kappa$

  Di P tale che A_i e A_m ($A_\kappa e A_m$) sono unificabili attraverso la sostituzione più generale σ

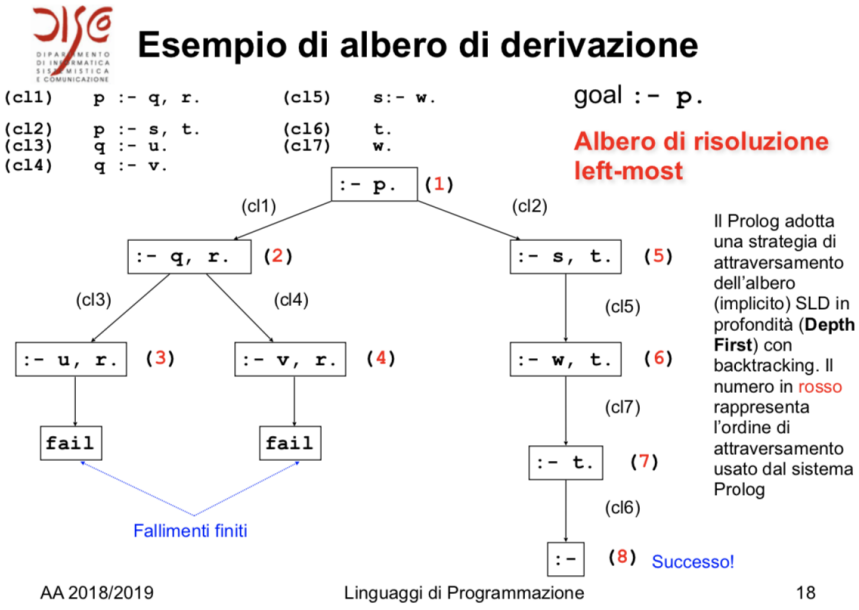
- Il nodo figlio è etichettato con la clausola goal
  :- $\sigma[A_1, ..., A_{m-1}, B_{i,1}, ..., B_{i,q}, A_{m+1}, ..., A_\kappa]$:- $\sigma[A_1, ..., A_{m-1}, A_{m+1}, ..., A_\kappa]$
  E il ramo dal nodo padre al figlio si etichetta sostituendo σ dalla clausola selezionata C_i, C_κ
- Il nodo vuoto indicato da :- non ha figli

Just a reminder: La regola **R** è variabile:

- Può essere scelta dalla sottoquery più a sinistra (se c'è) → Left - most
- Può essere scelta dalla sottoquery più a destra (se c'è) → Right - most
- Oppure può anche essere scelta da una sottoquery a caso
- Infine può essere scelta dalla sottoquery migliore

Prolog in pratica adotta la regola del Left Most, quindi considera la sottoquery più a sinistra e l'albero SLD (implicito) generato 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**.

Illustrazione presa dalle slides



Cerchiamo di ragionare bene sull'albero visto nell'esempio qui sopra. Abbiamo una serie di clausole, sono le varie cl1, 2, 3, ..., 7. Ora, quello che accade è che da sinistra verso andiamo ad applicarle alla clausola principale.

Non so se è stato già detto, tutto questo casino è semplicemente quello che effettua Prolog per quando dimostra un teorema

Quindi quel che si verifica è proprio che si hanno due fail, e chiudiamo poi con una clausola vuota (:-) quindi quello che concludiamo è che tutto è andato a buon fine.

Stesso esercizio MA con tecnica Right-Most fatto da Giulia

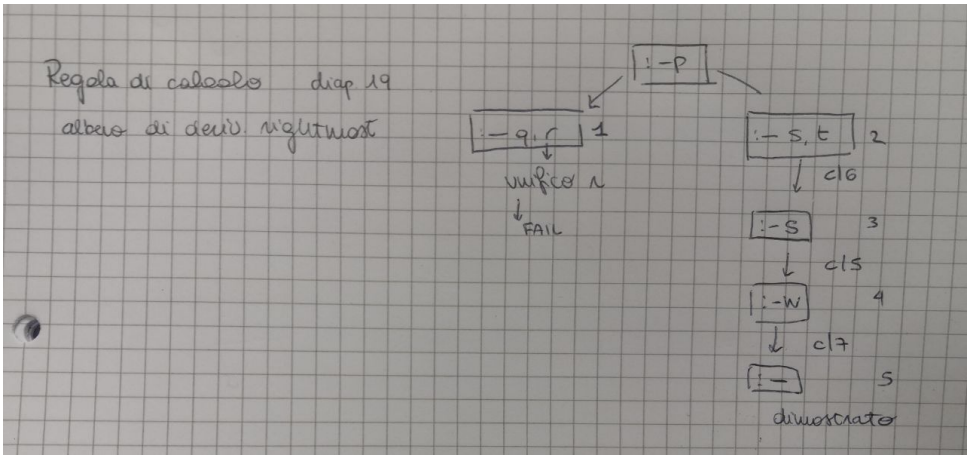
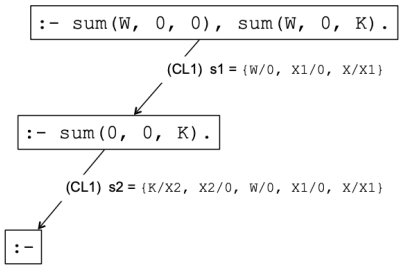


Immagine tratta dalle slides

```
sum(0,X,X).                                  (CL1)
sum(s(X), Y, s(Z)):- sum(X, Y ,Z).           (CL2)
G0 = :- sum(W, 0, 0), sum(W, 0, K).
```

Albero SLD con regola di calcolo "left-most"

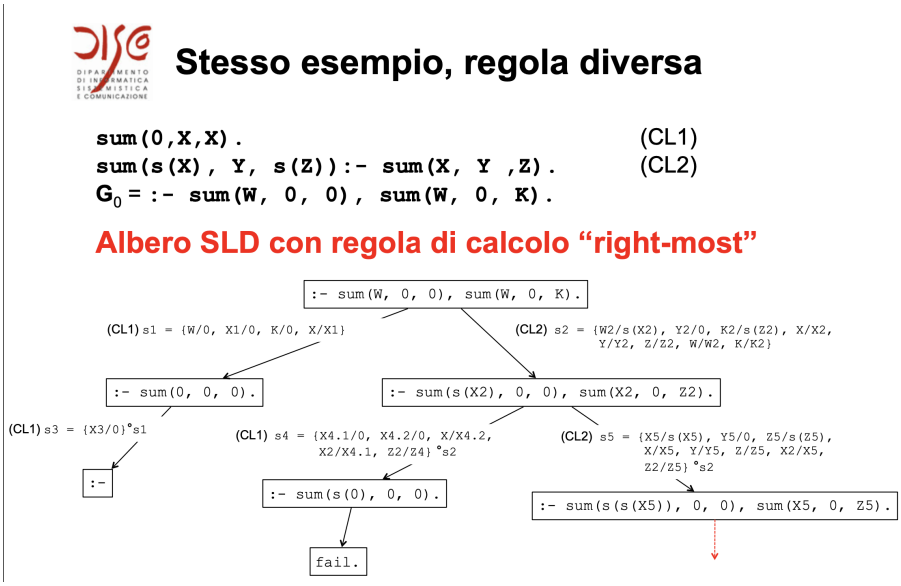


Le variabili `X1` e `X2` sono il risultato dell'operazione di ridenominazione (renaming) della variabile `X` in CL1; appena una clausola viene presa in considerazione le sue variabili sono ridenominate.

La notazione ° indica la **composizione** di sostituzioni

L'esercizio è giusto se per ogni passaggio si specifica quale clausola si usa, con i dati ad essa collegati come in questo esempio.

Nel primo passaggio usiamo solo la clausola 1 perchè la clausola 2 non era applicabile.



In questo caso abbiamo costruito prima da sinistra poi da destra, MA c'è da considerare che Prolog ragiona Left-Most, ANCHE SE al compitino è possibile comunque decidere il metodo di lavoro.

4.17 La regola di calcolo

Per ogni ramo di un albero SLD corrisponde una derivazione SLD. In pratica ogni ramo che termina con il nodo vuoto (:-)

La regola di calcolo influisce sulla scrittura dell'albero per quanto concerne sia l'ampiezza che la sua profondità

La cosa che va garantita è che tutti i cammini di successo siano uguali concettualmente, (la parola regola si incontra per l'ennesima volta tra l'altro, prima son state la regola di inferenze e la regola di Prolog)

Perciò alla domanda "quante accezioni del concetto di regola sono state menzionate", si risponderà: 3 Le regole di calcolo NON influisce su correttezza e completezza , pertanto qualunque sia R, il numero di cammini di successo (se finiti obv) è lo stesso in qualsiasi albero SLD costruibile per $P \cup \{G_0\}$

Capitolo 5

Modello di esecuzione Prolog

Come lavora effettivamente Prolog con le dimostrazioni?

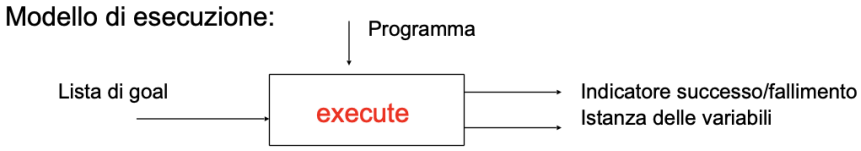
Prendiamo ora per esempio

$$p \ :- \ q, \ r.$$

(E rispettiamo questi spazi, quindi graficamente dovrà proprio essere considerata così) E' possibile dare più di una interpretazione a questa formula:

1. Interpretazione dichiarativa
   p è vera se sono veri q ed r
2. Interpretazione procedurale
   p è scomponibile in due sottoproblemi (q ed r)

Immagine tratta dalle slides



Da quest'immagine possiamo definire alcune cose:

(a) Un goal può essere visto come una chiamata ad una procedura

(b) Una regola può essere vista come definizione di una procedura in cui la testa è l'intestazione mentre la parte di destra è il corpo.

5.1 Le Estensioni

Per rendere Prolog un linguaggio che abbia senso essere usato si aggiungono le notazioni per le liste e meccanismi per il caricamento del codice Prolog, MA NON SOLO , si introducono anche le seguenti funzioni da "vedere":

- Meccanismi di controllo del backtracking
- Operazioni aritmetiche ($+ \ - \ * \ /$)
- Trattamento della negazione
- Possibilità di manipolare e confrontare le strutture dei termini
- Predicati meta ed extra-logici
- Predicati di I/O
- Meccanismi per modificare/accedere alla base di conoscenza

5.2 Il controllo di flusso

Se un sottoGoal fallisce, il dimostratore di Prolog sceglie un'alternativa procedendo in sequenza dall'alto verso il basso della lista O MEGLIO, dalla testa alla coda della lista delle clausole.

Però Prolog mette a disposizione il "**cut**", che non è altro che un taglio, che si indica con un '**!**', che serve per controllare questa megaEnormeListona di scelte.

5.2.1 Caratteristiche del Cut

Ha un problema di fondo, ovvero il fatto che è complesso da interpretare, perchè non ha un'interpretazione logica, ma procedurale.

Inoltre per capire come funziona serve conoscere meglio il funzionamento del dimostratore Prolog, che esegue su una macchina virtuale (SI'! Proprio come Java!)

Neanche a dirlo, la sua importanza non è sottovalutabile, se no che lo studieremmo a fare?

5.3 Il predicato '!' (Cut)

Per capire meglio il funzionamento di questo predicato serve scriverci una clausola generica (avente il cut, e grazie al.. No, non mi abbasserò a tal livello da dire "cut").

$$C = a : -b_1, b_2, ..., b_\kappa !, b_{\kappa+1}, ..., b_n.$$

Cerchiamo di procedere con ordine, cosa fa questo cut?
In pratica se il goal corrente G unifica con a e $b_1, b_2, ..., b_\kappa$ hanno successo ALLORA il dimostratore si impegna inderogabilmente alla scelta di C per dimostrare G

Ogni clausola alternativa (sì, quelle che si scorrono dall'alto verso il basso, quindi sarà quella dopo) per a che unifica con G viene ignorata

ATTENZIONE: Per un qualche b_j con $j > \kappa$ fallisse, allora il backtracking si fermerebbe ai cut (Le altre scelte per i b_i con $i \leq \kappa$ vengono rimosse, cancellate, e rimosse dall'albero di derivazione).

Inoltre, se il backtracking raggiunge il cut, automaticamente lui fallisce, e la ricerca procede dall'ultimo punto di scelta prima che G scegliesse C.

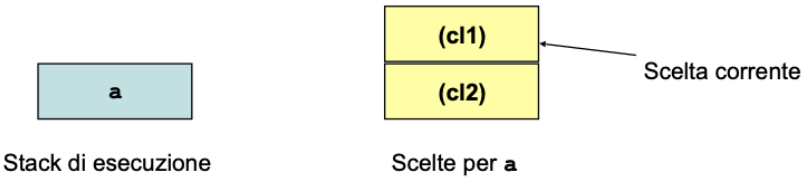
Ragioniamo sul seguente codice:

```
cl1: a :- p, b.
cl2: a :- p, b.
     cl3: p.
```

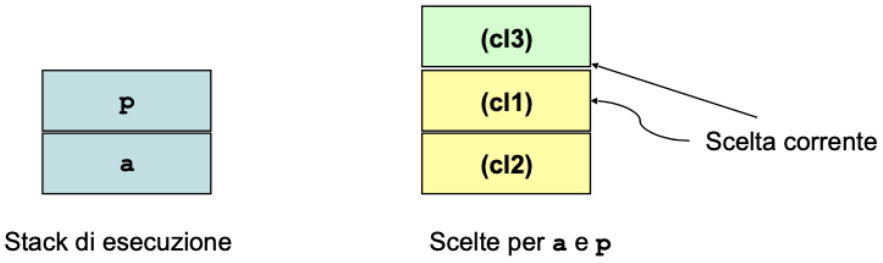
In cui cl indica la clausola, e considereremo la query "?- a.", Questo sarà lo stato interno di Prolog

---

Immagini tratte dalle slides



Nel momento in cui chiedo di dimostrare il seguente problema Prolog mi genera due stack, uno di esecuzione in cui si memorizza il goal corrente, mentre l'altro contiene le clausole unificanti
In pratica quello che accade è che p viene messa in cima allo stack



Se per ipotesi p ha successo, si inserisce b in cima allo stack
Se in questo caso la valutazione di b fallisce, si attiva il meccanismo magico del backtracking e quindi per proseguire si passa a considerare la seconda clausola, e lo stato interno di prolog cambia.
Come cambia?

Quello che accade ora è che p si mette in cima allo stack, ha successo, MA la valutazione di c fallisce, perciò di nuovo backtracking, però attenzione, siccome non ci sono più clausole allora anche a fallisce, e lo stack si svuota.

Analizziamo meglio però questi stack (queste pile)

- Pila di esecuzione:
  Contiene tutti i record di attivazione delle varie procedure (sostituzioni per unificare le varie regole)
- Pila di **backtracking** che contiene l'insieme dei "punti di scelta". Ad ogni fase della valutazione semplicemente contiene dei puntatori alle scelte "aperte" nelle fasi precedenti della dimostrazione.

Cosa succede alle queries con il Cut?

5.4 I tipi di Cut

Sono fondamentalmente 2, o meglio, sono due **USI** diversi, del predicato cut, il predicato rimane lo stesso:

- **Green Cut:** Utile per eseprimere "determinismo" (e quindi pre rendere più efficiente il programma)
- **Red Cut:** Usati solo per efficienza, perchè i pratica omettono alcune condizioni esplicite e modificano la semantica del programma equivalente senza i cuts (Sono indesiderabili anche se una volta ogni 10 anni, utili.. forse.)

Vediamo un esempio: Consideriamo un programma che deve fare il merge di due liste ordinate:

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

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

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

merge([], Ys, Ys).
merge(Xs, [], Xs).
```

Altro esempio: Consideriamo un programma che serva a ricontrollare quale è il minimo tra due numeri:

```
minimo(X,Y,Z) :- X=<Y
minimo(X,Y,Y) :- Y>X
```

Ora supponiamo di avere una query del tipo: merge([1,3,5],[2,3],Xs), qualli che saranno i passaggi saranno i seguenti:

1. merge([1,3,5],[2,3],Xs).
2. $1 < 2$: merge([3,5],[2,3],Xs1).
3. $3 < 2$: merge([3,5],[2,3],Xs1). FALLISCE, si fa backtracking al passaggio 2
4. $1 = 2$: merge([3,5],[2,3],Xs1). FALLISCE, ancora backtracking al passaggio 2
5. $1 > 2$: merge([3,5],[2,3],Xs1).

Solo una clausola avrà successo, le altre non verranno considerate.

Altro possibile problema con Prolog (L'ennesimo dei tanti). Se per ipotesi si avesse una query di questo genere:

```
?- merge([],[],Xs).
```

Quando noi andremo effettivamente a visualizzare i possibili valori della variabile Xs, prolog ci risponderà due volte con Xs = []. Si ha una soluzione in più, c'è ridondanza. Il green cut risolve questo specifico problema.

Determinismo:

Per definizione: Un programma Prolog si dice deterministico quando una sola delle clausole serve (o si vorrebbe servisse) per provare un dato goal.

Un programma sviluppato in Prolog si dice deterministico, ovvero, si ha una corrispondenza tra i dati di input e l'output effettivo. Non si ha questa ridondanza insomma, il codice diventerebbe così:

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

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

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

merge([], Ys, Ys) :- !.
merge(Xs, [], Xs) :- !.
```

Invece per quanto concerne la funzione del minimo:

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

In pratica il green cut taglia le strade che NON verrebbero comunque ad avere successo. Perchè se una strada ha successo, le altre non ha senso provarle. Se vogliamo essere pignoli, il secondo cut è ridondante MA per ragioni di simmetria viene messo nel programma.

5.5 Il Red cut

Lo stesso programma potevamo scriverlo in un modo ancora più sintetico:

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

Come vedete in questo caso manca tutta la condizione, ma si taglia proprio la soluzione. Ed è talmente intelligente che se gli spariamo la query minimo(2,5,5). (Tradotta sarebbe: E' 5 minore di 2?) ci dà TRUE! Geniale!

E' instabile: Già da qui si capisce che non sia corretto il programma sia scritto scorretto, e attenzione

E' SEMPRE COLPA DEL PROGRAMMATORE: Non è inutilizzabile, non è il cut in sè, è semplicemente complesso da capire ed usare. Non è sempre ovvio quando può fallire

Capitolo 6

Altri elementi di Prolog

6.1 Predicati Meta-Logici

La domanda è: Perchè?
Consideriamo il seguente predicato:

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

Non è invertibile ed introdurre la seconda clausola

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

non aiuta, perchè il sistema già sulla prima clausola si blocca.

Il problema sta nel decidere chi fa l'**inout** e chi invece l'**output** del calcolo.

Qua non è che ti dà il *fail* ma letteralmente ti dà errore. E quindi che si può fare? La risposta è nel titolo di questa sezione, ovvero, usare i predicati meta-logici.

Più precisamente però, cosa sono? Allora, abbiamo alcuni predicati NON invertibili (minimo, celsius_fahrenheit), e questo accade a causa dell'uso che abbiamo applicato dei predicati aritmetici dei predicati ($>$, $<$, $\leq$, **is**, etc).

I meta-logici sono predicati che danno un valore agli elementi delle nostre forme, fondamentalmente prendono le variabili, e le usano come oggetti del linguaggio, in modo da effettuare alcune scritture che aiutano a comprendere la semantica.

Mi spiego peggio: Prendete un'espressione del tipo variabile(X), che tradotto è "X è una variabile", bene, il predicato meta-logico è quello che definisce per esempio notVariabile(X)

Tutto ciò è inseribile nel corpo delle nostre regole, è quello il motivo del perchè i predicati metalogici assumono molta importanza.

Riprendendo l'esempio di prima:

```
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.
```

L'uso di var(X) permette di capire quale clausola usare! Cosa se ne ricava? Che l'uso di questi predicati magici consente di scrivere dei programmi efficienti ED allo stesso tempo corretti semanticamente.

Bene, ma se C ed F fossero entrambe variabili? Scritto così il programma ragiona che SE C è variabile esegue il primo blocco ALTRIMENTI il secondo, ma se fossero entrambe delle variabili?

Risposta: I due "if" sono se una è variabile e l'altra no, ALLORA fai cose, ma non è implementato SE entrambe son variabili o costanti, darebbe errore. Per intenderci, è uno XOR

Nel senso: Se la query fosse: celsius_fahrenheit(X, Y)? Possiamo usare dei cut per scrivere un programma più completo??

6.2 Ispezione di termini

Finora abbiamo visto come si usano dei termini per rappresentare strutture per i dati, ed abbiamo inoltre intuito che esistono termini atomici e composti, ed ora vedremo come implementare tutto ciò in Prolog:

- atomic(X): SE x è un numero od una costante ALLORA true
- compound(X): SE non atomic(X) ALLORA true

A questo punto, consideriamo un termine **Term**, ci saranno ben tre predicati che tornano utili per manipolarlo:

1. functor (Term, F, Arity)
   vero se Term è un termine, con Arity argomenti, il cui funtore (simbolo di funzione o di predicato) è F
2. arg (N, Term, Arg)
   Ritorna l'ennesimo argomento del termine, o meglio, è vero SE l'ennesimo argomento di **Term** è **Arg**
3. Term =.. L
   Questo è il più complesso, bisogna capirci bene a fondo, perchè qua si ha un predicato che è "=..", è un predicato binario di cui uno è termine ed uno è la lista.

   Nei termini del linguaggio universale si chiama anche **univ**.

   Ma cosa fa questo predicato? Prende un termine e lo trasforma in una lista, ed il primo elemento sarà il funtore, e tutti gli altri saranno semplici argomenti, in questo modo si appiattisce tutto. Diventa Lista = $argomento_1, argomento_2, ..., argomento_\lambda$

6.3 Predicati di ordine superiore

Quando si formula una domanda a Prolog, ci si aspetta una risposta (che poi alla fine non è altro che istanza individuale derivabile). Di fatto il backtracking come abbiamo visto ci permette di estrarre tutte le istanza derivabili una alla volta.

Ma se volessimo come risultato l'insieme di tutte le istanze che soddisfano una certa query?

Questa non è una richiesta associabile alla logica del primo ordine, noi ora si sta facendo una domanda in cui X si associa ad un insieme, e quest'idea di assegnare ad una variabile un insieme è quello che distingue la logica del secondo ordine dal primo.

Nella logica del primo ordine un elemento va ad un elemento, nel secondo ordine una variabile va ad un insieme, è un po' diverso insomma. **Prolog** mette a disposizione una serie di **Predicati su Insiemi** che **estendono** il modello computazionale del linguaggio di base.

6.4 Predicati su Insiemi

Sono fondamentalmente 3:

1. **findall** (Template, Goal, Set):
   E' un predicato di arità 3, forse non è chiaro da questa sintassi, ma l'idea di fondo è:

   Noi abbiamo un template, che è una variabile, ma può essere pure qualcosa di più complesso, una lista di oggetti, un oggetto, e come funziona?

   - SE **set** contiene tutte le istanze di **Template** che soddisfano **Goal** ALLORA vero
   - Le istanze di **Template** vengono ottenute tramite **Backtracking**

2. **bagof** (Template, Goal, Bag):
   Ha una semantica che matematicamente si chiama BagSemantic, non è un insieme ma un multi-insieme, e funziona così:

   - SE **bag** contiene tutte le alternative di **Template** che soddisfano il **Goal** ALLORA restituisce true

   Le alternative fondamentalmente vengono costruite facendo backtracking, solo se vi sono delle variabili libere nel **Goal**, che non appaiono in **Template**

   Inoltre si pulò pure dichiarare QUALI variabili non vanno considerate libere per dire che non si vuole fare backtracking rispetto a queste ultime, variabili di tipo esistenziale (si indicano con var^G)

3. **setof** (Template, Goal, Set):
   Si comporta esattamente come **bagof** MA **set** non contiene soluzioni duplicate, semplicemente

Prolog ci mette a disposizione anche altri predicati di ordine superiore, buona parte di questi funziona grazie al meccanismo delle meta-variabili, ovvero variabili interpretabili come query

Un esempio tipico è il predicato **chiama** che si può pensare essere definito come: chiama (G) :- G
In Swi-Prolog esiste un predicato che si chiama **call**, che fa questo

Grazie a queste meta-variabili, possiamo definire il predicato **Applica** che valuta una query composta da un funtore e da una lista di argomenti

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

Esempio

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

?- applica(father(terach), [C]).
C = abraham;
C = nachor;
false
```

6.5 Manipolazione della base di dati

Perchè manipolare direttamente la base di conoscenza? E' di base una cosa utile il fatto di poter manipolare un database, soprattutto se si tiene in mente il fatto che diventa possibile memorizzare risultati intermedi di una computazione. (Memorization o catching) Dato un programma Prolog, sappiamo che questo è praticamente costituioto da un database (per favore, chiamatelo knowledge base, basi di dati è al prossimo semestre) contenente fatti e regole

Il Prolog però mette a disposizione anche altri predicati che servono a manipolare direttamente la base di dati. Ovviamente, questi predicati vanno usati con molta attenzione, dato che modificano dinamicamente lo stato del programma.

I suddetti predicati sarebbero:

1. Listing
2. Assert, asserta, assertz
3. retract
4. abolish

Consideriamo come base il caso in cui si ha una Knowledge Base vuota ($\varnothing$), se interroghiamo Prolog con listing. lui ci risponderà sì.

Hey Prolog, mostrami la mia KB! Sì! *Prolog left the server*
Più o meno è quello che accade visto che non ha nulla da mostrare.

Se consideriamo un assert del tipo **assert(meravigliosa(firenze))**, lui ritornerà *true*, come giusto che sia! Scherzi a parte, l'assert è un comando che ha **SEMPRE** successo, però attenzione, non è per via del suo essere praticamente una masterball dei comandi l'importante, ma è il come può cambiare lo stato del nostro Database.

Riproviamo listing: ora verrà restituito il nostro assert di prima e un true:

```
?- listing.
meravigliosa(firenze).
true
```

Riassunto, l'assert gli ha iniettato una verità assoluta, che lui accetta a prescindere, paradossalmente se gli ripassiamo due volte lo stesso assert, lui lo prende di pacco, così come glielo diamo e lo riaggiunge al database.

Mi spiego peggio: Se passo a Prolog assert(bello(dave)), per due volte, lui quando farò il listing mi dirà:

```
?- listing.
bello(dave).
bello(dave).
true
```

E se lo dice Prolog, allora è proprio vero!

6.5.1 Asserzione di regole

Ciò che abbiamo appena visto non è altro che un'asserzione di **fatti**, ma come da titolo di questa sezione, è possibile asserire anche **regole**.

Esempio: Supponiamo di voler inserire una regola che dica che chiunque passa LP sia felice (E beati loro):

```
?- assert(felice(X) :- passatoLP(X)).
bello(dave).
bello(dave).
true
```

Precisazione: Per essere risucito a scrivere questa cosa in modo corretto ringrazio RC per la correzione. Ho fatto l'errore di ragionare come se :- si traducesse in "Implica" MA NON E' COSI'. Quel simbolo è traducibili in "E' implicato".

6.6 Varianti dell'assert

6.6.1 Asserta

Sarebbe Assert-a, ovvero l'asserzione viene messa in cima alla lista delle clausole

6.6.2 Assertz

Stesso discorso dell'assert-a ma in questo caso invece di inserire all'inizio te lo aggiunge alla fine in coda Ora indubbiamente abbiamo asserito delle verità assolute e dimostrabili, come la magifica bellezza di Dave MA, supponiamo che per sbaglio qualcuno asserisca una clausola che dopo non servirà più, come si può eliminare?

6.7 Retract

Il retract è l'operatore inverso dell'assert, molto semplicemente rimuove delle clausole dalla lista. Per intenderci, è l'operatore PERFETTO per clausole tipo:

carbonara(X, Y), ingredienti(parmigiano, Y).

Che tradotto è: LA CARBONARA E' SACRA.

Il listing produrrà una lista che chiaramente sarà priva della clausola che abbiamo rimosso (E grazie al **censura**).

Ad ogni modo è un comando che tornerà utile se abbiamo asserito più volte la stessa clausola, perchè non elimina ricorsivamente tutte le occorrenze.

Mi spiego peggio: Se avessimo due volte la stessa clausola (felice(personaGenerica)), e volessimo rimuoverle entrambe, non sarebbe possibile. Se ne rimuove solo una alla volta.

Capitolo 7

Input/Output in Prolog

Come ogni comando visto finora questi saranno predicati, che avranno la possibilità di acquisire e stampare valori (read, write). Inoltre sono presenti anche comandi di gestione di file e degli stream (open, close, seek etc.)

7.1 Read & Write

Con Read e Write è possibile stampare e leggere dei **termini** Prolog, con il write che sarebbe l'equivalente del toString() in Java, mentre read invoca il parser di Prolog, vediamo alcuni esempi presi dalle slides:

```
?- 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).
```

Analizziamo il terzo di questi che è già abbastanza complesso: read(What). praticamente cosa fa? Legge ed unifica con la variabile What. Quindi quando passiamo foo(42, Bar), lui farà l'unificazione con questa. Cosa fa poi? Praticamente Prolog ti scrive What = foo(42, _G270). (Per i gamers, sì, anche io ho pensato al G27).

7.2 Open & Close

Sempre dalle slides vi riporto gli esempi:

```
?- 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)
```

Come notiamo, l'open ha 3 argomenti che sono il percorso file, la modalità di apertura e infine lo stream, che è rappresentato da Out. (Mi manca sinceramente l'fopen di C/C++).

Sotto di fatto gli spariamo un write in Out con su scritto foo(bar). Il put dopo? Niente, ci mette il punto dopo la funzione Prolog. nl? Niente, fa il newline, il ritorno a capo, semplicemente. Inoltre

- Prolog usa la notazione 0'c per rappresentare i caratteri come termini
- Put spara fuori SOLO un carattere, write una stringa.
- Il read di un file legge tutto quanto MA vedremo un altro parametro che aggiunge pure un limite di caratteri da leggere

Capitolo 8

Interpreti in Prolog

La domanda che spesso mi sono posto: Che schifo di linguaggio è un linguaggio in cui per fare una somma devo soffrire ricorsivamente ogni singolo comando che scrivo? Risposta provvisoria: Bravissimi, ad un emerito **censura**

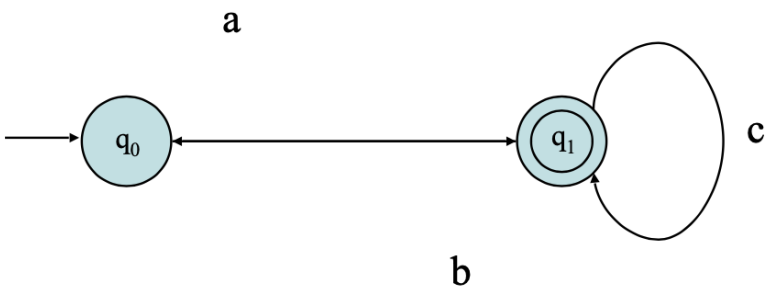
Per l'appunto: Prolog si presta ad utilizzi del tipo parsers o interpreti, ma come si può costruire un interprete che (non deterministicamente) riconosca dei linguaggi regolari? Dalle slide spunta questo codice:

```
accept([I | Is], S) :-
    delta(S, I, N),
    accept(Is, N).
accept([], Q) :- final(Q).
```

Data una stringa, questo frammento ti dice se si accetta o no la suddetta. Come ragiona?

- Se ho una transizione da S ad N con un simbolo i (delta indica le transizioni) allora accetta la stringa

E mi rendo conto che darti un codice senza contestualizzarlo sia un po' infame da un certo punto di vista, perciò vediamo un automa che almeno è più comprensibile



Come va a codificarsi questo automa?

```
initial(q0). final(q1).

delta(q0, a, q1).
delta(q1, b, q0).
delta(q1, c, q1).
```

Per decidere se una certa sequenza di simboli è riconosciuta dall'automa possiamo costruire il seguente predicato:

```
recognize(Input) :- initial(S), accept(Input, S).
```

8.1 Interprete di un CFG

```
%%% accept(Input, Stato, Pila).

accept([I | Is], Q, S) :-
     delta(Q, I, S, Q1, S1),
     accept(Is, Q1, S1).
accept([], Q, []) :- final(Q).
```

Considerando il linguaggio $L = \{wrw^R | w \in \{a, b, c\}^n \wedge n \geq 0$. Quello che accade è che l'automa verrà codificato così

```
initial(q0).
final(q1).

delta(q0, a, P, q0, [a | P]).
delta(q0, b, P, q0, [b | P]).
delta(q0, c, P, q0, [c | P]).
delta(q0, r, P, q1, P).
delta(q1, c, [c | P], q1, P).
delta(q1, b, [b | P], q1, P).
delta(q1, a, [a | P], q1, P).
```

Come nel caso degli automi a stati finiti, per decidere se una certa sequenza di simboli è riconosciuta dall'automa possiamo costruire il seguente predicato

```
recognize(Input) :- initial(S), accept(Input, S, []).
```

8.2 Meta-Interpreti

Il predicato call in pratica è il più semplice **meta-interprete**, chiaramente possiamo scrivere degli interpreti pi complessi e specializzati se accettiamo di rappresentare i programmi con una sintassi leggermente diversa.

Ok, ma cosa ci possiamo fare? Definiamo un predicato

```
solve(Goal :- solve(Goal, [])).
solve([], []).
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).
```

solve diventa un meta-interprete per i predicati **rule**che compongono il nostro sistema.

```
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.
```

Devo mi sa rivedermi l'indentazione, spero sia corretta. Appena riesco sistemo questa cosa.

Conclusione di Prolog

Prolog non è altro che un linguaggio in grado di esprimere problemi, conoscenze e soluzioni in modo naturale (Eh.. 'na robba. naturalissimo guarda.). Lo stile è lo stile **Dichiarativo**, il suo uso è efficace nella programmazione di sistemi deduttivi.

Lo stile di programmazione del Prolog e le idee alla sua base sono i componenti principali di una serie di nozioni utili alla gestione di relazioni semantiche su Web (RDF)

Capitolo 9

Linguaggi Funzionali (SECONDO PARZIALE)

Prima di parlare effettivamente di questo paradigma di programmazione, bisogna menzionare un argomento importante.

9.1 Trasparenza referenziale

Allora, con calma, è una proprietà, ed è valida per espressioni matematiche, e rende possibili le sostituzioni di espressioni con altre, basta che hai gli stessi valori.

Eh.. Cioè? Prendi f(x) + g(x), sono sostituibili con delle nuove funzioni f e h Se e SOLO se effettivamente producono gli stessi valori. (Si, oddio, più che funzioni, f e h scritte così sembrano variabili)

Eeeee... Quindi? Niente, per quanto riguarda LISP, ed i linguaggi funzionali in generale, si ha questa trasparenza referenziale come "fondamento".

Lo stile NON è più dichiarativo, qua è una funzione, una super mega giganteschissima funzione, più precisamente è una combinazione di tutte le mie varie funzioni. La composizione di funzioni è la chiave. Quello che accade è che definiremo funzioni per poi andare ad usarle, tutto lì. Le regole delle funzioni vengono applicate.

Anche qui è tutto ricorsivo: I cicli non ci sono, come non c'erano in Prolog, gli assegnamenti si possono circa fare, ed è tutto di nuovo ricorsivo. Ah sì, per il resto tutto funzioni, tutto completamente a funzioni.

Capitolo 10

LISP

LIST Processing, cioè elaborazione delle liste, nasce verso la fine degli anni 50 (Siamo solo nel 2019, not bad), successivamente poi LISP si è evoluto in altri sottoLinguaggi (Tipo le fork su GitHub) come Common LISP e Scheme, noi useremo Common LISP.

L'ambiente NON è compilato, o meglio, dovrebbe non esserlo, perchè si ha un interprete tipo Prolog. L'interprete **VALUTA TUTTO**, argomenti della funzione compresi. T U T T O. Non metterò più di tanta roba, vi lascio giusto qualche esempio.

```
prompt> (+ 40 2)
42

prompt> (- 84 42)
42

prompt> (* 2 3 7)
42
```

- Come si può notare, le funzioni aritmetiche in LISP accettano un numero variabile di argomenti

```
prompt> (+ 2 10 10 20)
42
```

- Le funzioni in LISP si combinano secondo le ovvie norme

```
prompt> (+ 2 (* 2 10) 20)
42
```

Ovvero al posto di un valore, possiamo inserire un'espressione (e.g., una chiamata ad una funzione) che lo denota

10.1 Introduzione

Di base LISP è composto di funzioni. Un esempio di funzione potrebbe essere ad esempio

```
prompt> (+ 3 5)
8
```

Come già da qua si vede c'è una funzione definita tutta tra parentesi. Si apre con + (indica che funzione è) e poi gli argomenti della funzione che si considera. Ora, c'è un detto famoso: **LISP è il miglior simulatore di parentesi**. Basti pensare che se voglio fare una somma dentro una somma mi vien fuori

```
prompt> (+ (+ 4 6) 5)
15
```

Immaginatevi cosa vien fuori tra poco. Dalle slide vi riporto qualche esempio più preciso:

- Numeri
  - Interi: `42 -3`
  - Virgola mobile: `0.5 3.1415 6.02E+21`
  - Razionali: `3/2 -3/42`
  - Complessi: `#C(0 1)`
- Booleani (*)
  `T NIL`
- Stringhe
  `"sono una stringa"`
- Operazioni su booleani
  `null and or not`
- Funzioni su numeri
  `+ - / * mod sin cos sqrt tan atan plusp > <= zerop`

Inoltre bisogna anche mettersi nell'ottica che se Prolog funzionava tutto al contrario, invece LISP no, funziona in ordine da sinistra verso destra. Anche qui non sto a insistere troppo, riporto diretto dalle slide.

Data un'espressione LISP

$$(f\ x_1\ x_2\ \ldots\ x_N)$$

la valutazione procede da sinistra verso destra a partire da x_1
fino a x_N producendo I valori v_1, …, v_N.
La funzione *f* viene "valutata" successivamente e viene applicata
ai valori v_1, …, v_N.

Questa regola è inerentemente ricorsiva.
Alcuni operatori speciali valutano gli argomenti in modo diverso
(**if**, **cond**, **defun**, **defparameter**, **quote**, etc)

L'interprete realizza il suo parsing partendo da sinistra e procedendo ricorsivamente verso destra, easy, riconosce i simboli etc.

Osservazione: v_1 è prodotto da x_1, nel senso che siccome l'interprete ragiona su ogni elemento delle parentesi

10.1.1 Come definisco una funzione o una variabile

Ci son due funzioni predefinite per questo scopo che sono **defparameter** e **defun**, uno definisce un parametro, per ipotesi se avessi:

```
prompt> (defparameter x 15)
15
```

In java avremmo scritto

```
x = 15;
```

(Mancherebbe il tipo per i dati ma.. Non c'è in questo caso). Ci sono diversi modi per fare gli assegnamenti, in questo caso si ha defparametere come base ma ne vedremo migliori più avanti. Definiamo ora una ipotetica funzione:

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

E' come aver fatto in java

```
public int quadrato(int x){
return x * x;
}
```

E' abbastanza intuitivo, una volta definite le nostre funzioni e costanti, possiamo poi utilizzarle ovviamente

```
prompt> (quadrato x)
1764
```

Qua si lavora con simboli, non stringhe, occhio a non confondersi su questa cosa.

10.2 Funzioni λ (Funzione anonima)

In pratica si possono costruire delle funzioni che siano anonime, dette anche funzioni λ. In parole povere lambda è una funzione che però non ha un nome.

A che serve? Puoi direttamente applicarla a degli argomenti, tipo per dire

```
prompt> (lambda (x) (+ 2 x))
%In pratica aggiungo 2 ad x
prompt> ((lambda (x) (+ 2 x))40)
```

Ogni volta si deve star lì a ridefinirsela insomma, capiamo già da qui che ha più senso (in base alla riusabilità) definirsi di pacco una funzione e stop. Per ora lasciamole un po' da parte, più tardi ci serviranno di più. Ora, proviamo a definirci una funzione del tipo valore_assoluto. A cosa è uguale?

$$valore\_Assoluto = \begin{cases} x \ se \ x > 0 \\ 0 \ se \ x = 0 \\ -x \ se \ x < 0 \end{cases}$$

Bene, questo da un punto di vista prettamente matematico, ma invece come sarebbe dal punto di vista più programmativo? In questo caso c'è il **cond**. E' una funzione che associa semplicemente elemeti tra loro.

```
(cond (x a) (y b))
```

In pratica associa x con a e y con b, con x, a, y, b che sono delle espressioni.

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

Da qui abbiamo un esempio carino anche di indentazione del codice. Non c'è molto da dire su < > e =, sono delle booleane che o danno True o False. Se ci son le booleane c'è anche l'**if**, che funziona come una qualsiasi funzione booleana

```
defun(maggiore_di_zero (x))
(if (> x 0) T)
```

In pratica ti spara True se lo è, come si osserva non c'è l'else, perchè va in successione all'altra espressione. Proviamo a definire una funzione ricorsiva, il fattoriale.

```
(defun fattoriale (n)
(if (= n 0) 1
(* n (fattoriale (- n 1)))))
```

Tradotto, se n è 0 allora dammi 1 altrimenti dammi il prodotto di n con il fattoriale di n-1. Manca l'else, ma da qua si capisce che vien fatto in sequenza.

E' tutto ricorsivo: Come in Prolog tutto va ricorsivamente, PERCIO' sarà possibile in qualche modo tentare di simulare un'iterazione.

```
(defun fatt-ciclo (n acc)
(if (= n 0)
acc
```

```
(fatt-ciclo (- n 1) (* n acc))))

(defun fattoriale (n)
(fatt-ciclo n 1))
```

In pratica gli chiedo, è n = 0? restituisco l'accumulatore, quindi questo si incrementerà costantemente, infatti dopo si ha il fattoriale di n - 1 che va a prendere l'accumulatore e lo moltiplica per n. Sotto infatti abbiamo che il fattoriale di n è il risultato di questa iterazione.

Questa struttura si chiama ricorsione in coda: Ovvero la ricorsione è **L'ultima** operazione che eseguo. Per intenderci, la funzione di fibbonacci NON è eseguibile così, perchè funziona con due ricorsioni attive.

Ricorsione

Dato un insieme di funzioni mutualmente ricorsive, questo può rappresentare una macchina di Turing, pertanto i linguaggi funzionali puri (senza assegnamenti e salti) sono **Turing Completi**

10.3 Strutture per i dati e funzioni

Ipotizziamo di dover costruire una libreria per fare dei calcoli su numeri razionali. Di cosa ci sarà bisogno? Innanzitutto si deve assumere di avere a disposizione una funzione che costruisce una *Reppresentazione* di un numero razionale:

Banalmente: (**crea-razionale n d**) $\Longrightarrow <razionale \frac{n}{d}>$ Ora però ci servono due funzioni per definirci il numeratore ed il denominatore

- numer
- denom

A questo punto è easy crearsi una libreria (Sieh, è già tanto se so scriverti una somma, sicurissimo so farti la libreria guarda), e vien fuori una roba del genere:

- La libreria è la seguente (le funzioni mancanti sono lasciate per esercizio)

```
(defun somma-raz (r1 r2)
    (crea-razionale (+ (* (numer r1) (denom r2))
                       (* (numer r2) (denom r1)))
                    (* (denom r1) (denom r2))))

(defun molt-raz (r1 r2)
    (crea-razionale (* (numer r1) (numer r2))
                    (* (denom r1) (denom r2))))

(defun =-raz (r1 r2)
    (= (* (numer r1) (denom r2))
       (* (numer r2) (denom r1))))
```

10.4 Cons-Cells e funzione CONS

Una struttura essenziale è la cons-cell, cioè una coppia di puntatori a due elementi. Sono create dalla funzione **cons** che praticamente alloca della memoria (E' tipo malloc di C oppure un new in Java)

cons : *<oggetto Lisp>* × *<oggetto Lisp>* → *<cons-cell>*

I due puntatori di una cons cell sono chiamati - per ragioni storiche - **car** e **cdr**, a cui corrispondono due funzioni

```
prompt> (defparameter c (cons 40 2))
c

prompt> (car c)
40

prompt> (cdr c)
2
```

10.5 Rappresentazione dei numeri razionali

In pratica tu hai la tua primitica cons, e hai delle funzioni che sono

- Crea-razionale
- numer
- denom

che vengono schematizzate così:

```
(defun crea-razionale (n d)
    (cons n d))

(defun numer (r) (car r))

(defun denom (r) (cdr r))
```

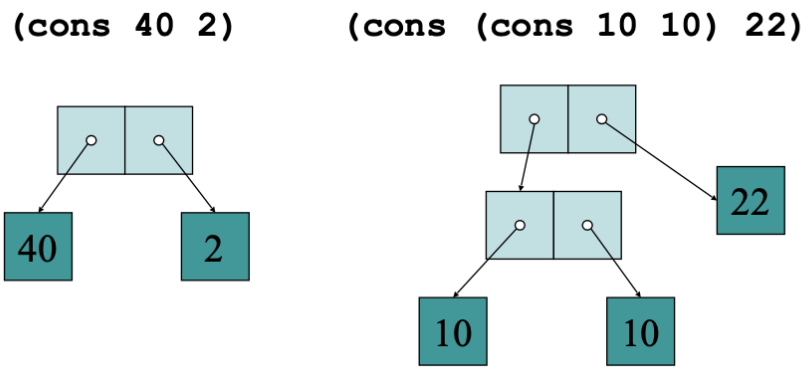
Esempio

```
prompt> (denom (crea-razionale 42 7))
7
```

Visto che si è menzionato qualcosa di più grafico, proviamo a dare un che di informativo a ciò che abbiam detto. Come detto primo quello che fa la conf è:

- Generare una coppia di puntatori

Si possono ottenere dei veri e propri grafi che sono in una notazione box and pointer, come in questa illustrazione:



Precisazione: Quando disegnate questi grafi aspettate un attimo prima di scrivere i numerelli dentro i quadratini. Ok e se voglio fare un puntatore che punta ad un solo elemento? Niente, invece di metterci un valore gli do NIL che sarebbe un puntatore a.. niente. Nel senso lista vuota, ma il senso è quello.

Come si comporta LISP quando gli diamo un cons ?
Esplode, perchè ci odia palesemente :)
Scherzi a parte, si ha una notazione "dotted-pair" che praticamente è una coppia separata da un punto, in cui gli spazi son significativi. (NIL . T), che è il corrispettivo di una funzione (cons NIL T).

Lamentela mia: Sarò fastidioso, ma mi urta il sistema nervoso "NIL", ma non potevan chiamarlo NULL come tutte le persone normali? No? Vabbeh. Ora sono offeso :c

Edit: A quanto pare più avanti si vedrà che null è una funzione, e quindi.. Han dato il nome di un valore ad una funzione, giusto per confondermi meglio

Occhio a una cosa: Se io pongo (cons 42 nil) ottengo una lista avente solo un numero che è 42, invece se fosse (cons nil 42) non è una lista, è una cosa un po' diversa, **altra cosa importante:** Se notiamo, quello che accade è che facendo (cons a nil), si ottiene una lista, perciò se al posto di nil ci fosse un'altra lista tipo (cons a (cons a nil)), in pratica aggiungi una lista a dentro una lista quindi hai una lista con dentro una lista, mentre se vuoi aggiungere solo l'elemento devi usare (cons a (cons nil a)), che concatena alla prima lista il nuovo carattere semplicemente.

10.6 Esempi sulle liste

Esempio 1

Come si estrae l'*n*-esimo elemento da una lista?

```
(defun list-ref (n list)
    (if (<= n 0)
        (car list)
        (list-ref (- n 1) (cdr list)))
```

Come calcoliamo la lunghezza di una lista?

```
(defun lunghezza (l)
    (if (null l)
        0
        (+ 1 (lunghezza (cdr l)))))
```

dove la funzione `null` ritorna il valore `T` se l'argomento passatole è il valore `NIL`

In pratica quello che accade è che:
La funzione list-ref si definisce con il nome nth (Per via di Common Lisp), mentre invece la funzione cdr ritorna il resto di una lista, inteso come "tutti" gli altri elementi meno la testa, quindi si ha in common lisp il "rest".

Ora, la funzione car torna invece il primo elemento della lista, la testa, e qua si segna con first infatti, al fine di evitarci le combinazioni standard si ha che :

```
(car (cdr L))
(car (cdr (cdr L))
(car (cdr (cdr (cdr L))))
...
```

Common lisp ha in libreria le funzioni second, third, fourth fino a tenth, cioè fino al decimo elemento.. credo.. della lista(?) A logica è quello.

Esempio 2: Concatenazione delle liste Anche qui c'è l'append, che letteralmente prende due parametri che sarebbero due liste (pure la lista vuota) e quel che fa è concatenarle, se però una delle due liste è vuota, ti spara quell'altra e basta.

10.7 Espressioni simboliche

Se in prolog avevamo le clausole di horn qui ci sono fondamentalmente

- Numeri
- Simboli
- Stringhe
- Cons-cells
- Oggetti di base di LISP

Cloè fondamentalmente pure qua ci sono espressioni simboliche (sexp's) che sono a tutti gli effetti costituite dalle cons-cells

- Dato che programmi e sexp's in Lisp sono equivalenti, possiamo dare le seguenti regole di valutazione (ed implementarle nella funzione eval!)

Data una sexp:

- Se è un atomo (ovvero, se non è una cons-cell)
  - Se è un numero ritorna il suo valore
  - Se è una stringa ritornala così com'è
  - Se è un simbolo
    - Estrai il suo valore dall'*ambiente corrente* e ritornalo
    - Se non esiste un valore associato allora segnala un errore
- Se è una cons-cell (O A_1 A_2 … A_n) allora si procede nel seguente modo
  - Se O è un operatore speciale, allora la lista (O A_1 A_2 … A_n) viene valutata in modo speciale
  - Se O è un simbolo che denota una funzione nell'ambiente corrente, allora questa funzione viene applicata (**apply**) alla lista (VA_1 VA_2 .. VA_n) che raccoglie i valori delle valutazioni delle espressioni A_1, A_2, …, A_n.
  - Se O è una Lambda Expression la si applica alla lista che (VA_1 VA_2 .. VA_n) che raccoglie i valori delle valutazioni delle espressioni A_1, A_2, …, A_n
  - Altrimenti si segnala un errore

Ora vi riporto gli esempi dalle slide di funzioni ricorsive:

- Fattoriale

```
(defun fatt (n)
      (if (zerop n) 1 (* n (fatt (- n 1)))))
```

- Al prompt:

```
prompt> (fatt 3)
      n diventa associato a 3 ("bound to" 3):

(eval '(fatt 3))
→ (eval '(if (zerop 3) 1 (* 3 (fatt (- 3 1)))))
      … → (eval '(* 3 (fatt 2))))
                … → (eval '(*3 (* 2 (fatt 1))))
                … → (* 3 (* 2 (* 1 (fatt 0))))
      … → (* 3 (* 2 (* 1 1)))
… → … 6
```

- **eval** applica la funzione al suo argomento, definendo i legami di N "in cascata" e costruendo una espressione che alla fine verra' valutata, a partire dalla sotto-espressione piu' annidata

Il concetto è che la forma ricorsiva delle liste si presta bene alla programmazione ricorsiva.. Cioè ovviamente direi. Come?

- Scrivo il valore della funzione nel caso base
- Ricorsivamente mi devo ridurre al caso base operando su un argomento ridotto, minore, decrementato. FINCHE' non sarà 0 o il caso base.

10.8 Operatori di uguaglianza

Si hanno due differenti operatori per verificare l'uguaglianza tra due oggetti:

- eql
- equal

10.8.1 Eql

E' applicato a simboli e numeri interi e puntatori. In pratica ti dice se due valori sono uguali, tipo (eql 'a 'a):

In pratica gli chiedi: è la stringa "a" uguale alla stringa "a"? Ma potrebbe pure esserci un valore numerico

10.8.2 Equal

equal è esattamente come **eql** ma va a scavare in profondità alle liste, nel senso che applica ricorsivamente a tutte le sottoliste presenti l'eql.

Esempio: prendi la lista (((((a))))), in pratica hai una serie di sottoliste che ricorsivamente devi andarti a calcolare, equal fa anche questo, quindi è più figo, potente.

10.9 Il vantaggio del paradigma funzionale

Una volta capito cos'è il quote, la lista possiamo finalmente dare un senso all'esistenza di questo tipo di paradigma.

Per esempio: prendete la lista

```
(defparameter pari (list 2 4 6 8 10))
```

E supponismo ora che per ipotesi volessimo moltiplicare tutti gli elementi di questa lista per un determinato valore (In pratica prodott di un vettore per κ)

```
(defun scala-lista (l fattore)
        (if (null l)
                nil
                (cons (* fattore (car l))
                                (scala-lista (cdr l) fattore))))
```

Notare che questa funzione che certamente mi ricordo già a memoria.. Vero? usa una ripetizione di append, l'operazione fa questa funzione si può astrarre se andiamo ad astrarre il concetto di valore funzionale.

Cioè? L'astrazione " applica la funzione f a tutti gli elementi della lista L e ritorna una lista dei valori"è nota come "map"; in Common Lisp la funzione mapcar svolge questo compito.

Mapcar è una funzione predefinita ma è riscrivibile come:

```
(defun mapcar* (funzione lista)
       (if (null lista)
                        nil
                        (cons (funcall funzione (car lista))
                                        (mapcar* funzione (cdr lista)))))
```

In cui mapcar* è usato per evitare errori in CommonLisp, e si noti inoltre anche **funcall** che invece chiama la funzione prendendo un certo argomento.

Qua ci sono alcuni esempi presi dalle slide

Supponiamo di avere una serie di funzioni chiamate **scala-4**, **scala-10**, **scala-pi** etc

```
(defun scala-4  (x) (* x 4))
(defun scala-10 (x) (* x 10))
(defun scala-pi (x) (* x pi))
```

dove **pi** è la costante **3.14....**

La funzione **scala-lista-10** può essere scritta come

```
prompt> (defun scala-lista-10 (lista)
          (mapcar 'scala-10 lista))
scala-lista-10

prompt> (scala-lista-10 '(1 2 3 4 5))
(10 20 30 40 50)
```

10.10 λ-Funzioni

Avevo voglia di scrivere λ come simbolo a caso, ma quando si useranno queste funzioni si usa **lambda**, che serve per indicare delle funzioni anonime, praticamente costruisce delle funzioni quando ce n'è bisogno.

Cioè? In LISP è possibile definire queste funzioni in modo che ti definisci unas funzione solo quando ti serve. Una volta creata, la usi, ma appena finito di usarla *puf*, morta, schiattata, defunta, REST IN PEACE BOI. Consente sicuramente di ottimizzarsi meglio la memoria

Esempi:

```
prompt> (lambda (x) (+ x 42))
#<funzione>

prompt> ((lambda (x) (+ x 42)) 42)
84

prompt> (scala-lista '(1 2 3)
                     (lambda (x) (* x 3)))
(3 6 9)
```

In pratica il concetto è che con il lambda puoi generare delle funzioni che si basano su quest'ultima: Per intenderci se voglio fare una funzione che si chiama "IncrementaDi" con x come parametro, la **lambda** praticamente si può strutturare con un (+ x y), perciò si definisce una funzione:
(defun incrementa-x (x) (lambda (y) (+ x y)))
Se per ipotesi ora si volesse fare una funzione che ti somma 5 diventerebbe (defun incrementa-5 (x) (incrementa-x 5))

10.11 Let

Prendiamo una funzione di questo tipo:

$$f(x,y) = x(1+xy)^2 + y(1-y) + (1-y)(1+xy)$$

In pratica tu usando lambda puoi costruirti tutti i valori intermedi, e quel che accade è che la funzione vien chiamata con due valori che saranno i valori intermedi da usare:

```
(defun f (x y)
       ((lambda (a b)
           (+ (* x (quadrato a))
              (* y b)
              (* a b)))
(+ 1 (* x y))
(- 1 y)))
```

Questo tipo di chiamate a funzioni anonime è così utile da essere stato ri-codificato con un nuovo operatore speciale: **let** La funzione riportata qua sopra diventerebbe:

```
(defun f (x y)
       (let ((a (+ 1 (* x y)))
                 (b (- 1 y))
             )
          (+ (* x (quadrato a))
             (* y b)
             (* a b))))
```

Ovvero, l'operatore let ci permette di introdurre dei nuovi nomi (variabili) locali da poter riutilizzare all'interno di una procedura; la sua sintassi è la seguente.

10.12 Funzioni di un ordine superiore

Si intende ordine di una funzione quello che consiste nel prendere una funzione e usarla come argomento di un'altra ma potrebbero essere anche di più. Il fatto che possiamo implementare queste funzioni è fondamento del paradigma funzionale. Per ora si è visto il **mapcar** ma ce ne sono altre tipo:

- compose
- fold (o reduce)
- complement
- filter (variante di remove e delete)

10.12.1 Compose

E' l'equivalente della composizione di funzioni in matematica. La sua semantica è: date due funzioni (di un unico argomento) f e g, mi deve tornare f di g(x), qui ci torna di nuovo utile il **lambda**

```
(defun compose (f g)
   (lambda (x)
              (funcall f (funcall g x))))

prompt> (funcall (compose 'first 'rest)
                                        '(1 2 3 4 5))
2
```

Se per dire volessimo fare il fattoriale del

10.12.2 Filter

E' un filtro, quindi rimuove tutti gli elementi che non rispettano una condizione. Nel senso all'atto pratico è, prendo un elemento, SE quell'elemento non ha una determinata caratteristica, non considero l'elemento. Praticamente è uno switch.

```
      (cond ((null lista) nil)
            ((funcall predicato (car lista))
             (cons (car lista)
                   (filter predicato (cdr lista))))
            (T (filter predicato (cdr lista)))))
```

10.12.3 Accumula

Detta anche fold o reduce, praticamente progressivamente ha un valore che si accumula in base al risultato ottenuto su ogni elemento della lista. Esempio: La sommatoria di una lista, o la produttoria.

```
(defun accumula (f iniziale lista)
       (if (null lista)
           iniziale
           (funcall f (car lista)
                    (accumula f iniziale (cdr lista)))))
```

Capitolo 11

Le Keywords

I parametri invece di essere scritti in ordine possono essere anche associati a dei nomi, delle keywords. Molte funzioni LISP hanno già questo di comportamento. Tipo io prendo una lista (1 2 3 4 5 6 7 8 9 10) e posso dire che :start sia 3 e :end sia 10, quindi non mi serve averceli più in ordine. Se ad una funzione voglio dire che ciò che le arriva è una keyword devo usare &key che (defun make-point (&key x y))
(list x y)
Quando noi lo richiamiamo possiamo chiamarla con: (make-point :y 5 :x 6), andrà ad assegnare ad x 6 e y 5 malgrado l'ordine sia invertito.

Capitolo 12

Input/Output in Common LISP

Come in ogni linguaggio anche qui ci sono funzioni che permettano di scrivere in output e ricevere in input dati. E queste funzioni sono READ e PRINT. Oltre all'input output su schermo posson pure prendere file.

12.1 Read

La Read semplicemente acquisisce e legge valori, è un input di oggetti LISP, in cui per oggetto si intende una lista, un valore, un carattere, tutto.

12.2 Output

Ovviamente è bene avere metodi per stampare tipo con la write in Prolog, e Common Lisp ci mette a disposizione la funzione **FORMAT** (Avete presente fprintf in c? Ecco quello è il concetto). Va detto che è complessa la **FORMAT**, prendiamo ad esempio qualcosa del tipo:

```
prompt> (format t "il␣fattoriale␣di␣~D␣e'␣~D~%" 3 (fact 3))
il fattoriale di 3 e' 6
NIL
```

Le direttive in una stringa da formattare sono introdotte da ~ vediamone alcune

- ~D stampa numeri interi
- ~% ritorna a capo
- ~S stampa un oggetto Lisp secondo la sua sintassi standard
- ~A stampa invece un oggetto secondo una sintassi piacevole.. Che non so cosa voglia dire, ma oh, le cose piacevoli sono belle.

12.3 Streams in CommonLisp

Allora, ci son 3 tipi di stream, uno standard input, uno output, e uno error, che è più o meno come in java (syserr, sysout, sysin), inoltre le funzioni **Read, print, e Format** accettano un numero variabile di argomenti, e uno di questi stream (di **output** per format e print e **input** per read).

12.3.1 Manipolazione dei file

Vi avviso, non è così semplice, perchè ci son diversi passaggi (è come in Java più o meno, il funzionamento è lo stesso), non andremo in profondità su come si gestiscono file di testo in LISP **MA**.

Di base: per aprire un file si deve usare "**with-open-file**", la sintassi precisa è la seguente:

```
(with-open-file (<var> <file> :direction :input) <codice>)

(with-open-file (<var> <file> :direction :output) <codice>)
```

La variabile <var> è il nome dello stream, o meglio si associa allo stream aperto sul file e può venire usata all'interno di <codice> cioè del segmento di codice che andiamo a considerare. Tutto questo è **dentro** una funzione.

with-open-file è una figata perchè si preoccupa di chiudere sempre e comunque lo stream associato a <var>, anche se ci sono errori.

```
(with-open-file (out "foo.lisp"
                                        :direction :output
                                        :if-exists :supersede
                                        :if-does-not-exist :create)
        (mapcar (lambda (e)
                        (format out "~S" e))
                '((1 . A) (2 . B) (42 . QD) (3 . D))))
```

Come già detto lo stream si chiude con l'ultima parentesi, ora apriamo lo stesso stream ma in formato di input

```
(with-open-file (in "foo.lisp"
                        :direction :input
                        :if-does-not-exist :error)
        (read-list-from in))
```

In pratica in questo caso andiamo a vedere se un file c'è, se c'è appost, possiamo modificarlo, altrimenti ritorniamo che c'è stato un errore.

```
(defun read-list-from (input-stream)
        (let ((e (read input-stream nil 'eof)))
                (unless (eq e 'eof)
                        (cons e (read-list-from input-stream)))))
```

La cosa importante è il simbolo 'eof che indica per l'appunto l'end of file, ed in questo caso precisamente quello che stiamo dicendo è che quando arriviamo alla fine deve uscire.

12.4 Interazione con l'ambiente LISP

Quando si parla di lisp, lavoriamo a linea di comando, ciò che fa quest'ultima sono 3 operazioni fondamentali

1. Leggere i comandi (**Read**)

2. Valuta con l'**eval** finchè non trova il risultato della valutazione

3. Stampa il risultato della valutazione

E tutto questo è praticamente un ciclo while(**true**), all'infinito. Ma come funziona più precisamente questo loop? Come posso implementarlo in lisp? La parte che segue è un po' più cicciosa di quella che abbiam visto finora, ci saranno un po' più cose da tenere a mente.

ù