Linguaggi di Programmazione 2021-2022
Cenni di logica formale

Marco Antoniotti
Gabriella Pasi
Rafael Peñaloza

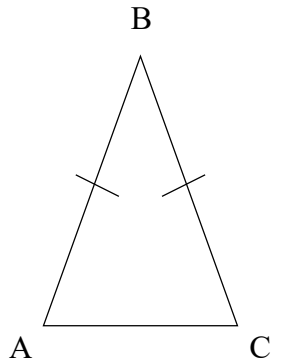
Logica e ragionamento

Euclide di Alessandria

Cosa state per vedere…

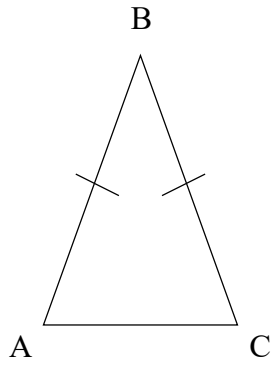
- La cosa principale che dobbiamo acquisire è un **linguaggio** che ci permetta di parlare dei **linguaggi logici**
- Quindi dobbiamo capire come:
  - Un **ragionamento**
  - Può essere **formalizzato**
  - In un numero di **passi**
  - Connessi da **regole**
  - A partire da **premesse**
  - Per arrivare alle **conclusioni**
- In altre parole: **teoremi** e **dimostrazioni**
- Tutto ciò può essere visto come una «*computazione*»

Semplice Teorema di Geometria



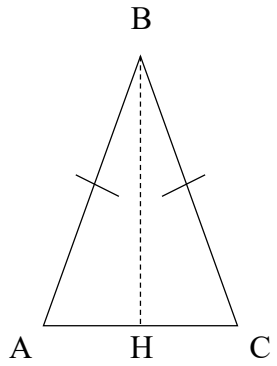
Dato un triangolo isoscele ovvero con AB = BC, si vuole dimostrare che gli angoli ∠A e ∠C sono uguali

Semplice Teorema: conoscenze pregresse



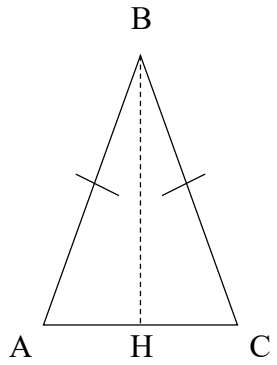
1. Se due triangoli sono uguali, i due triangoli hanno lati ed angoli uguali
2. Se due triangoli hanno due lati e l'angolo sotteso uguali, allora i due triangoli sono uguali
3. BH bisettrice di ∠B cioè
   ∠ABH = ∠HBC

Semplice Teorema: Dimostrazione



- **Dimostrazione**
  - AB = BC *per ipotesi*
  - ∠ABH = ∠HBC *per* (3)
  - Il triangolo HBC è uguale al triangolo ABH *per* (2)
  - ∠A e ∠C *per* (1)

Semplice Teorema: Dimostrazione



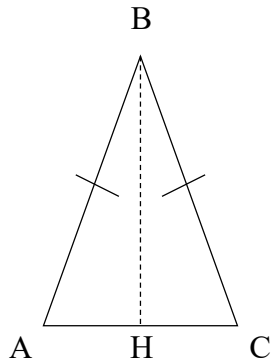
Abbiamo trasformato (2) in

⇒ ***Se*** AB = BC ***e*** BH = BH ***e*** ∠ ABH = ∠ HBC,
allora il triangolo ABH è uguale al triangolo 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

Semplice Teorema: Formalizzazione



Obbiettivo

Razionalizzare il processo che permette affermare:

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

dove il *simbolo di* ***derivazione logica***, $\vdash$, significa «*consegue*», «*segue che*», «*allora*» ecc. ecc.

Semplice Teorema: Formalizzazione

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

- Abbiamo assunto che:

  P = {$AB = BC, \angle ABH = \angle HBC, BH = BH$}

- Avevamo conoscenze pregresse:

  1. $AB = BC \wedge BH = BH \wedge \angle ABH = \angle HBC \Rightarrow \triangle ABH = \triangle HBC$
  2. $\triangle ABH = \triangle HBC \Rightarrow AB = BC \wedge BH = BH \wedge AH = HC \wedge \angle ABH = \angle HBC \wedge \angle AHB = \angle CHB \wedge \angle A = \angle C$

Semplice Teorema: Formalizzazione

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

Abbiamo costruito una catena di formule:

P1: $AB = BC$	da **P**
P2: $\angle ABH = \angle HBC$	da **P**
P3: $BH = BH$	da **P**
P4: $AB = BC \wedge BH = BH \wedge \angle ABH = \angle HBC$	da P1, P2, P3, e **introduzione della congiunzione**
P5: $\triangle ABH = \triangle HBC$	da P4, Regola$_1$ e **Modus Ponens**
P6: $AB = BC \wedge BH = BH \wedge AH = HC \wedge \angle ABH = \angle HBC \wedge \angle AHB = \angle CHB \wedge \angle A = \angle C$	da P5, Regola$_2$ e **Modus Ponens**
P7: $\angle A = \angle C$	da P6 e l'**eliminazione della congiunzione** (il simbolo $\wedge$)

Regole di inferenza

Il processo di dimostrazione

Una «prova» D, dove **S** è l'insieme «affermazioni note» e *F* la frase (i.e., la formula) che vogliamo provare.

$$D \equiv \mathbf{S} \vdash F$$

(ovvero: *F* **è una conseguenza di S)** è una sequenza di «passi»

$$D = < P_1, P_2, \ldots, P_n >$$

dove

$$P_n = F$$
$$P_i \in \mathbf{S} \cup \{ P_j \mid j < i \}$$

o P_i può essere ottenuto da $P_{i1},\ldots,P_{im}$ (con $i1 < i, \ldots, im < i$) mediante l'applicazione di una **regola d'inferenza**

Regole di inferenza e calcoli logici

- Un insieme di regole di inferenza costituisce la base di un **calcolo logico**
- *Diversi insiemi di regole danno vita a diversi calcoli logici*
- Lo scopo di un calcolo logico è di manipolare delle formule logiche in modo completamente **sintattico** al fine di stabilire una connessione tra un insieme di formule di partenza (di solito un insieme di formule dette assiomi) ed un insieme di conclusioni
- Nel seguito presenteremo due tipi di logica (due linguaggi logici) con il calcolo logico ad esse associato:
  - **Logica Proposizionale (Logica delle Proposizioni)**
  - **Logica dei Predicati del Primo Ordine**

Riferimenti

- Moltissimi libri, articoli, pagine web, etc
  - ***How to Prove It***, Daniel J. Vellman, Cambridge University Press, 2nd Edition, 2006
  - Capitolo 1 di ***Discrete Mathematics and Its Applications***, Kenneth H. Rosen, McGraw Hill, 2007 (Nth Edition and later ones)
  - ***A Mathematical Introduction to Logic***, H. B. Enderton, Academic Press, 2000

- Per i più avventurosi (regali di Natale o compleanno)
  - ***Gödel, Escher and Bach, An Eternal Golden Braid***, Douglas R. Hofstadter, Basic Books, 1979
  - ***Logicomix***, Apostolos Daxiadis, Christos Papadimitriou, Alecos Papadatos, Annie Di Donna, Bloomsbury, 2009
  - ***Il diavolo in cattedra***, Piergiorgio Odifreddi, Einaudi, 2003
  - ***Le menzogne di Ulisse***, Piergiorgio Odifreddi, Longanesi & C., 2004
    - I due libri di Odifreddi sono molto simili; il primo un po' più tecnico

Logica Proposizionale

- La logica proposizionale si occupa delle conclusioni che possiamo trarre da un insieme di - per l'appunto – **proposizioni.**

- Una logica proposizionale è sintatticamente definita da un insieme **P** di proposizioni

- **Esempi**

  P = {AB = BC, ∠ABH = ∠HBC, BH = BH}

  oppure

  P = {piove, l'unicorno è un animale mitico}

  oppure

  P = {p, q, r, s, w}

Logica Proposizionale

- All'insieme **P** è associata una funzione di verità, o di valutazione, ***V*** (spesso indicata con ***T*** o con ***I***)

  $V : P \rightarrow \{vero, falso\}$

  che associa un valore di verità ad **ogni** elemento di **P** (cioè ad ogni proposizione).

- La funzione di valutazione è il ponte di connessione tra la sintassi e la semantica di un linguaggio logico

- **Esempi**
  V(`q`) = *vero*, ***V***(`p`) = *vero*, ***V***(`w`) = *falso*
  V(`l'unicorno è un animale mitico`) = *vero*
  V(`piove`) = *falso*

Logica Proposizionale

- Le proposizioni in una logica proposizionale possono essere combinate utilizzando una serie di **connettivi logici**
  - Congiunzione ∧
  - Disgiunzione ∨
  - Negazione ¬
  - Implicazione ⇒

- Chiamiamo **FBF** (formule ben formate) l'insieme di tutte le formule formate dagli elementi di **P** e dalle loro combinazioni

- Le formule atomiche in **P** e le loro negazioni vengono anche chiamati **letterali** (positivi e negativi)

- **Esempi**
  `piove ∧ l'unicorno è un animale mitico`
  `p ⇒ q`
  `p ∨ q ⇒ ¬s`

Logica Proposizionale

- Il valore di verità di una proposizione dipende dalla funzione di verità ***V***
- Il valore di verità di una formula composta dipende dal valore di verità delle sue componenti
- La definizione della funzione di valutazione ***V*** viene quindi estesa sul dominio **FBF** (formule ben formate)
- La funzione ***V*** associa un valore di verità ad un elemento di **FBF** secondo le regole seguenti:

V(¬s)	≡	non ***V***(s)
V(a ∧ b)	≡	***V***(a) e ***V***(b)
V(a ∨ b)	≡	***V***(a) o ***V***(b)
V(p ⇒ q)	≡	(non ***V***(p)) o ***V***(q)

Tavole di verità

- Un modo per calcolare il valore di verità di una proposizione (composta) è quello di utilizzare la **tavola di verità**:

P	Q	P∧Q	P∨Q	¬P	P⇒Q
v	v	v	v	f	v
v	f	f	v		f
f	v	f	v	v	v
f	f	f	f		v

Calcoli logici – ovvero, dimostrazioni

- Mentre la funzione di verità (o la tavola di verità) costituisce la parte ***semantica*** di un insieme di proposizioni - dice ciò che è vero e ciò che è falso sotto l'interpretazione considerata -, un calcolo logico dice come **generare** nuove formule (cioè espressioni sintattiche) a partire da un insieme di partenza (gli assiomi) **A**
  - **A** $\subset$ **WFF**
  - **S** $\subseteq$ **WFF**
  - **A** = **S** (inizialmente; dopo una prova, **S** può espandersi includendo la conseguenza *F*)

- Un calcolo logico deve garantire che tutte le nuove formule generate siano «vere» se l'insieme di assiomi consiste solo di formule «vere»

- Il processo di generazione si chiama **dimostrazione**

Calcolo Proposizionale – regole di inferenza

- Il calcolo proposizionale è basato su una serie di regole di inferenza ben testato (sino dai tempi di Aristotele and Crisippo) che ci permettono di ottenere delle nuove formule a partire da un insieme di assiomi

- Una regola di inferenza ha la seguente forma generale:

$$\frac{F_1, F_2, ..., F_k}{R} \quad \text{[nome regola]}$$

  dove ogni F_i rappresenta una formula (vera) in **FBF** e R è la formula generata da «inserire» in **FBF**; il «**nome regola**» ci dice che regola di inferenza stiamo usando

- L'esempio tipico di regola di inferenza è il cosiddetto modus ponens

Regole di inferenza: modus ponens

$$\frac{p \Box q, \quad p}{q} \quad \text{[modus ponens]}$$

- **Esempio**
  - $p \Longrightarrow q$: **Se** piove, **allora** la strada è bagnata
  - p : piove
  - q : (allora) la strada è bagnata

Ovvero, la regola (sintattica) del modus ponens ci permette di aggiungere le conclusioni di un'implicazione (chiamate «regola» in altri contesti) al nostro insieme di formule ben formate «vere» da **FBF**

Regole di inferenza: modus tollens

$$\frac{p \Box q, \quad \downarrow q}{\downarrow p} \quad \text{[modus tollens]}$$

- **Esempio**
  - $p \Longrightarrow q$: **Se** piove, **allora** la strada è bagnata
  - $\neg q$: la strada non è bagnata
  - $\neg p$: (allora) non piove

Ovvero, la regola sintattica del modus tollens ci permette di aggiungere la premessa negata di una «regola» al nostro insieme di formule ben formate «vere»

Regole di inferenza
eliminazione ed introduzione di 'E'

$$\frac{p_1 \mid p_2 \mid \ldots \mid p_n}{p_i} \quad \text{[eliminazione } \mid\text{]}$$

$$\frac{p_1, p_2, \ldots, p_n}{p_1 \mid p_2 \mid \ldots \mid p_n} \quad \text{[introduzione } \mid\text{]}$$

Esempio di applicazione di eliminazione di 'e':

- Piove **e** la strada è bagnata
- (segue che) piove

Ovvero, la regola sintattica dell'eliminazione della congiunzione ci permette di aggiungere all'insieme **FBF** i singoli componenti di una congiunzione

queste regole si chiamano anche "di congiunzione" e "di semplificazione"

Regole di inferenza
introduzione di 'O'

$$\frac{p}{p \int q} \quad \text{[introduzione } \int \text{]}$$

- **Esempio**
  - Piove
  - Piove **o** c'è vita su Marte

- Ovvero, la regola sintattica dell'introduzione della disgiunzione ci permette di aggiungere i singoli componenti di una formula complessa
  - Questa regola è anche detta «di addizione»

Regole di inferenza
varie ed eventuali (e tautologie)

$$\frac{p \int \downarrow p}{\text{vero}}$$ [terzo escluso]

$$\frac{\downarrow\downarrow p}{p}$$ [eliminazione $\downarrow$]

$$\frac{p \mid \text{vero}}{p}$$ [eliminazione $\mid$]

$$\frac{p \mid \downarrow p}{q}$$ [contraddizione]

L'ultima regola (contraddizione) ci dice che da una contraddizione si può trarre qualunque conseguenza

Regole di inferenza

- Le regole di inferenza che abbiamo visto (introduzione ed eliminazione di congiunzione, modus ponens, introduzione della disgiunzione ed altre non citate) fanno parte del cosiddetto calcolo naturale o di Gentzen, il nome del loro formalizzatore.

- Il calcolo di Gentzen, e molte varianti, formalizzano i modi di derivare delle conclusioni a partire da un insieme di premesse
  - In particolare permettono di derivare "direttamente" una formula ben formata mediante una sequenza di passi ben codificati

- La regola del *modus ponens* (eliminazione dell'implicazione) assieme al principio del terzo escluso, possono però essere usati in un *altro* modo, procedendo «per assurdo» alla dimostrazione di una data formula
- Questa vista "estesa" della regola del modus ponens è conosciuta come **principio di risoluzione**
- *Ma prima, un po' di esempi …*

Il principio di risoluzione

- Il **principio di risoluzione** è una regola di inferenza generalizzata semplice e facile da utilizzare e implementare (assieme all'algoritmo di unificazione, che vedremo tra breve)
  - Opera su formule ben formate trasformate in **forma normale congiunta** (che vedremo successivamente)
  - Ognuno dei congiunti di queste formule viene detto **clausola**
- L'osservazione fondamentale alla base del principio di risoluzione è un'estensione della nozione di rimozione dell'implicazione sulla base del principio di contraddizione
  - Solitamente è usata per fare ***dimostrazioni per assurdo***

$$\frac{p \lor \downarrow r, \quad s \lor r}{p \lor s}$$

Clausola *risolvente*

$$\frac{\downarrow r, \quad r}{\bot}$$

Clausola *vuota*

Il principio di risoluzione

- Si noti che la generazione della clausola vuota, corrisponde all'aver dimostrato che il mio insieme di formule ben formate contiene una **contraddizione**
  - Se ho derivato r e $\neg r$ allora posso dedurre qualunque cosa, quindi anche la clausola vuota

$$\frac{\downarrow r, \quad r}{\bot}$$

Clausola *vuota*

Regole di inferenza: risoluzione (unitaria)

$$\frac{\downarrow p, \quad q_1 \rfloor\ q_2 \rfloor \cdots q_k \rfloor\ p}{q_1 \lceil\ q_2 \lceil \cdots q_k} \quad \text{[unit resolution]}$$

$$\frac{p, \quad q_1 \lceil\ q_2 \lceil \cdots q_k \lceil \downarrow p}{q_1 \lceil\ q_2 \lceil \cdots q_k} \quad \text{[unit resolution]}$$

- La regola di risoluzione è molto generale (si vedano, ad esempio le nozioni di «risoluzione generale» e «risoluzione Davis-Putnam»)
- Quando una delle due clausole da risolvere è un *letterale* (ovvero una proposizione o, come vedremo, un predicato) anche negato, come nel caso di p nei due esempi qui sopra, allora si parla di *risoluzione unitaria* (*unit resolution*)

- **Esempio**
  - (Da) <*Non piove*>, <*piove o c'è il sole*>
  - (Segue che) <*C'è il sole*>

Dimostrazioni per assurdo

- Supponiamo di avere a disposizione un insieme di formule **FBF** (vere, data una certa interpretazione ***V***)
- Supponiamo di voler dimostrare che una certa proposizione *p* (o formula atomica) è vera
- Possiamo procedere usando il metodo della **reductio ad absurdum** (dimostrazione per assurdo)
  - Assumiamo che ¬*p* sia vera
  - Se, combinandola con le proposizioni in **FBF** ottengo una contraddizione, allora concludo che *p* deve essere vera

Dimostrazioni per assurdo

- Esempio 1: proviamo che p è vera (o, meglio, derivabile)
  - **FBF** = {p}
  - Assumiamo $\neg p$
  - **FBF** $\cup$ {$\neg p$} genera una contraddizione
  - Quindi p deve essere vera

- Esempio 2: proviamo che q è vera (o, meglio, derivabile)
  - **FBF** = {$p \Rightarrow q$, p, $\neg w$, e, r}
  - Assumiamo $\neg q$
  - **FBF** $\cup$ {$\neg q$} genera una contraddizione
    - Infatti

      $p \Rightarrow q \equiv \neg p \vee q$ combinato con p (preso da ***FBF***, ovvero con un'applicazione del modus ponens) produce q

      quindi abbiamo $q \wedge \neg q$, la contraddizione che cerchiamo.

      Formalmente, se applico a q e a $\neg q$ il principio di risoluzione ottengo la clausola vuota $\perp$, cioè una contraddizione.

Ricapitolazione

- Un calcolo logico (proposizionale) manipola i seguenti elementi

- **Sintassi**
  - Un insieme di proposizioni **P**
  - Un insieme di formule ben formate **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**

- **Semantica**
  - Una *funzione di verità* che ci permette di distinguere ciò che è vero da ciò che è falso rispetto a una data interpretazione
    - Funzione di Interpretazione: ***V*** (o ***I***)
    - tavole di verità

Assiomi (Conoscenze pregresse)

- Alcune proposizioni sono sempre vere, indipendentemente dalla loro interpretazione (sono tautologie)
  - **A1**: $A \Rightarrow (B \Rightarrow A)$
  - **A2**: $(A \Rightarrow (B \Rightarrow C)) \Rightarrow ((A \Rightarrow B) \Rightarrow (A \Rightarrow C))$
  - **A3**: $(\neg B \Rightarrow \neg A) \Rightarrow ((\neg B \Rightarrow A) \Rightarrow B)$
- Alcune tautologie sono codificabili come regole di inferenza e viceversa
  - **A4:** $\neg(A \wedge \neg A)$ principio di non-contraddizione
  - **A5:** $A \vee \neg A$ principio del terzo escluso

Come si costruiscono delle prove: qualche esempio

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.

Domande

a) L'unicorno è mitico?

b) L'unicorno è magico?

c) L'unicorno è cornuto?

Procedimento

1. Esprimere il problema in forma di logica delle proposizioni
   – Molto importante: selezionare accuratamente l'insieme di proposizioni
2. Individuare i teoremi da dimostrare
3. Dimostrare i teoremi

Esempio

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

Proposizioni:

UM = **unicorno è mitico**

UI = **unicorno è immortale**

UMag = **unicorno è magico**

UC = **unicorno è cornuto**

Esempio

*Se l'**(unicorno è mitico)**^{UM}, allora l'**(unicorno è immortale)**^{UI}, ma se non **(è mitico)** $^{\neg UM}$ allora **(è mortale)**$^{\neg UI}$. Se l'**(unicorno è mortale)**$^{\neg UI}$ o l'**(unicorno è immortale)**^{UI}, allora **(unicorno è cornuto)**^{UC}. L'**(unicorno è magico)**^{UMag} se l'**(unicorno è cornuto)**^{UC}.*

Trascrizione:

$UM \Rightarrow UI$

$\neg UM \Rightarrow \neg UI$

$\neg UI \vee UI \Rightarrow UC$

$UC \Rightarrow UMag$

Esempio

Supponiamo di voler rispondere alle domande di seguito (leggi: provare questi teoremi)

a) L'unicorno è mitico?

b) L'unicorno è magico?

c) L'unicorno è cornuto?

Rappresentazione:

$\boldsymbol{S} = \{UM \Rightarrow UI,\ \neg UM \Rightarrow \neg UI,\ \neg UI \vee UI \Rightarrow UC,\ UC \Rightarrow Umag\}$

a) $\boldsymbol{S} \vdash UM$?

b) $\boldsymbol{S} \vdash Umag$?

c) $\boldsymbol{S} \vdash UC$?

Esempio

$$S \vdash UC$$

P1: $\neg UI \vee UI \Rightarrow UC$	da ***S***
P2: $\neg UI \vee UI$	da **A5** (cfr., slides precedenti)
P3: UC	da **P1**, **P2** e *modus ponens*

Esempio

$$S \vdash UMag$$

P1: $\neg UI \vee UI \Rightarrow UC$	da ***S***
P2: $\neg UI \vee UI$	da **A5**
P3: UC	da **P1**, **P2** e *MP*
P4: $UC \Rightarrow Umag$	da **S**
P5: $UMag$	da **P3**, **P4** e *MP*

Esercizio

Dimostrare (a)

Sintassi (tipografia) e semantica

- Come abbiamo già accennato esiste una differenza tra **sintassi** e **semantica** in logica
- Un argomento a sostegno dell'esistenza di questa differenza consiste nel considerare le seguenti "stringhe" di caratteri

  2A 42 42 XLII 33 101010

  Esse sono la rappresentazione (in "linguaggi" diversi) dello stesso oggetto

- Un calcolo logico fornisce una manipolazione sintattica (simbolica)
  - L'operatore di **derivazione** $\vdash$, è un operatore sintattico

- La semantica di un insieme di formule dipende dalla funzione di valutazione ***V***
  - Simmetricamente viene introdotto l'operatore di "**conseguenza logica**" (in Inglese **entailment**), denotato da $\models$

Sintassi e semantica

In particolare, data un particolare calcolo logico:

$$\mathbf{S} \vdash \boldsymbol{f} \quad \text{se e solo se} \quad \mathbf{S} \models \boldsymbol{f}$$

dove **S** è un insieme di formule iniziale ed ***f*** è una fbf; il tutto in dipendenza da una particolare funzione di verità ***V***

Questo è il Teorema di **Completezza e Validità**

Tautologie e modelli

- Una particolare **interpretazione** ***V*** che rende vere tutte le formule in **S** viene detta **modello** di **S**
- Una fbf sempre vera indipendentemente dal valore assegnato dei **letterali** viene detta **tautologia**
- Ovvero, una tautologia è vera «in» ogni modello

Logica del primo ordine

Logica proposizionale vs. logica del primo ordine

- La logica proposizionale è molto utile
  - Ha caratteristiche computazionali chiare
  - Ha una semantica altrettanto chiara
- La logica proposizionale ha numerose limitazioni
  - Non ci permette di fare asserzioni circa insiemi di elementi in maniera concisa
- **Esempio**
  Il classico sillogismo
  1. Tutti gli uomini sono mortali
  2. Socrate è un uomo
  3. Quindi Socrate è mortale

  Il problema è la prima frase, che non è "esprimibile" in logica proposizionale

Logica proposizionale vs. logica del primo ordine

- Per risolvere questi problemi, la **Logica del Primo Ordine** (*First Order Logic* - FOL; ne esistono di "ordine" superiore) introduce le nozioni di:
  - Variabile
  - Costante
  - Relazione (o predicato)
  - Funzione
  - *Quantificatore*
- Ovvero, un linguaggio logico del primo ordine è costituito da **termini** (notare il nuovo termine!) costruiti a partire da:
  - **V** insieme di simboli di variabili
  - **C** insieme di simboli di costante
  - **R** insieme di simboli di relazione o predicati di varia arità
  - **F** insieme di simboli di funzione di varia arità
  - Connettivi logici (già visti) e simboli di quantificazione ∀ (universale) e ∃ (esistenziale)

Sintassi della logica del primo ordine

- La costruzione di un linguaggio logico del primo ordine è (ovviamente!!!) **ricorsiva**

- I termini più semplici sono i predicati

$$r \subseteq \mathbf{C}_0 \times \mathbf{C}_1 \times \ldots \times \mathbf{C}_k$$

ovvero **relazioni** cartesiane su **C**, scritte come $r(c_1, c_2,\ldots,c_k)$

- Le funzioni sono definite con il seguente dominio e codominio

$$f : \mathbf{C}_0 \times \mathbf{C}_1 \times \ldots \times \mathbf{C}_m \rightarrow \mathbf{C}$$

una funzione si scrive come $f(c_1, c_2,\ldots,c_m)$

  - *Attenzione!* Una funzione $f(c_1, c_2,\ldots,c_m)$ NON è una fbf!

Sintassi della logica del primo ordine

- Le **formule ben formate** (**FBF**; in Inglese, *well-formed formulae* - WFF) di un linguaggio logico del primo ordine sono costruite ricorsivamente nel seguente modo

  - Un **termine** t_j può essere un elemento di **C**, di **V**, oppure un'applicazione di funzione $f(t_1, t_2, \ldots, t_s)$
  - Un termine costituito da un **predicato** $r(t_1, t_2, \ldots, t_k)$, dove ogni t_i è un termine, appartiene ad **FBF**
  - Diversi elementi di **FBF** connessi dai connettivi logici standard (congiunzione, disgiunzione, negazione, implicazione) appartengono ad **FBF**
    - Denotiamo con $\boldsymbol{t}(t_1, t_2, \ldots, t_r)$ tale combinazione di termini

  - Le formule

    $\forall x . \boldsymbol{t}(t_1, t_2, \ldots, x, \ldots, t_r)$ e $\exists x . \boldsymbol{t}(t_1, t_2, \ldots, x, \ldots, t_r)$ appartengono ad **FBF**

Logica del primo ordine

- Con le definizioni precedenti, possiamo ora rivedere l'esempio socratico
  - Socrate è un uomo.
  - Tutti gli uomini sono mortali.
  - Allora Socrate è mortale.

- Costanti individuali
  - **C** = {**Parmenide, Socrate, Platone, Aristotele, Crisippo, Pino, Gino, Rino, Ugo**}

- Predicati
  - **R** = {**uomo**, **mortale**}

Logica del primo ordine

- Rappresentiamo le frasi seguenti
  - Tutti gli uomini sono mortali.
  - Socrate è un uomo.
  - Allora Socrate è mortale.

- Traduzione delle asserzioni principali
  - $\forall$ **x . (uomo(x)** $\Rightarrow$ **mortale(x))**
  - **uomo(Socrate)**

- Traduzione della conclusione a cui vogliamo arrivare
  - **mortale(Socrate)**

Logica del primo ordine e calcoli logici

- Naturalmente non possiamo semplicemente dire che la nostra conclusione “segue” dalle asserzioni iniziali
- Dobbiamo giustificare questa conclusione sulla base della sintassi e della semantica del nostro linguaggio logico (del primo ordine)
- In altre parole dobbiamo usare di nuovo delle regole di calcolo per ottenere la conclusione
- Dato che il linguaggio è più ricco, dobbiamo costruire delle regole più sofisticate

Logica del primo ordine

Regola di eliminazione del quantificatore universale

$$\frac{\forall x.\, T(..., x, ...), \quad c \lfloor \mathbf{C}}{T(..., c, ...)} \quad [\text{eliminazione } \forall]$$

Logica del primo ordine

- Con la regola appena introdotta possiamo derivare la nostra conclusione a partire dalle asserzioni iniziali
  1. **uomo(Socrate)**
  2. $\forall x$ **.(uomo(**x**)** $\Rightarrow$ **mortale(**x**))**
  3. **mortale(Socrate)**

$$\frac{(\forall x.\ \mathbf{uomo}(x) \rightarrow \mathbf{mortale}(x)),\quad \mathbf{Socrate} \vdash \mathbf{C}}{\mathbf{uomo(Socrate)} \rightarrow \mathbf{mortale(Socrate)}}\ [\text{eliminazione } \forall]$$

$$\frac{\mathbf{uomo(Socrate)},\quad \mathbf{uomo(Socrate)} \rightarrow \mathbf{mortale(Socrate)}}{\mathbf{mortale(Socrate)}}\ [\text{eliminazione } \rightarrow]$$

Altre regole in LPO

- Date le regole per l'eliminazione del quantificatore universale, dobbiamo esibire delle regole per la manipolazione del quantificatore esistenziale

$$\frac{T(..., c, ...),\quad c \vdash \mathbf{C}}{\exists x.\ T(..., x, ...)}\ [\text{introduzione } \exists]$$

- Valgono anche le seguenti identità riguardanti l'interazione tra quantificatori e negazione

$$\exists x. \downarrow T(..., x, ...) \quad \ldots \quad \downarrow \forall x.\ T(..., x, ...)$$
$$\forall x. \downarrow T(..., x, ...) \quad \ldots \quad \downarrow \exists x.\ T(..., x, ...)$$

Sommario

- Introduzione alla Logica Matematica

- Dimostrazioni
  - Concetto di Calcolo Logico e Regola di Inferenza
  - Concetto di Derivazione
  - Distinzione tra Sintassi e Semantica
  - Concetto di Conseguenza Logica (Entailment)
  - Significato dei teoremi di Consistenza e Completezza

- Calcoli Logici
  - Calcoli Naturali
  - Calcoli per Risoluzione

- Logica Proposizionale
- Logica del Primo Ordine
  - Universo, variabili, funzioni, predicati ed operatori universali ed esistenziali

- Per chi ha intenzione di vedere più in dettaglio questi argomenti, i libri citati precedentemente contengono tutte le informazioni necessarie

Linguaggi di Programmazione 2021-2022
Prolog e programmazione logica I

Marco Antoniotti
Gabriella Pasi
Rafael Peñaloza

Prolog

- Questa parte del corso è dedicata al linguaggio Prolog ed al paradigma di **programmazione logica**
- Il libro di testo è *The Art of Prolog* di Sterling e Shapiro, MIT Press (oppure I. Bratko, *Prolog Programming for Artificial Intelligence*)
- Sistema consigliato: SWI-Prolog

Programmazione logica

- La programmazione logica nasce all'inizio degli anni settanta da studi sulla deduzione automatica: il Prolog costituisce uno dei sui risultati principali.
- La programmazione logica **NON** è soltanto rappresentata dal Prolog: essa costituisce infatti un settore molto ricco che cerca di utilizzare la logica matematica come base dei linguaggi di programmazione. Prolog è l'esempio più noto.
- Obiettivi del linguaggio
  - semplicità del formalismo
  - linguaggio ad alto livello
  - semantica chiara

Programmazione logica e la logica matematica

- Logica matematica: formalizzazione del ragionamento

- Con l'avvento dell'informatica:
  - Logica Matematica ⇒ Dimostrazione automatica di teoremi (procedura di Davis e Putnam e principio di risoluzione)
  - Interpretazione procedurale di formule (Kowalski)
    - Logica come linguaggio di programmazione

- Utilizzata in varie applicazioni: dalle prove di correttezza dei programmi alle specifiche, da linguaggio per la rappresentazione della conoscenza in Intelligenza Artificiale a formalismo per DB (esempio, `datalog`).

Stile dichiarativo della programmazione logica

- Programma come insieme di formule
- Grande potere espressivo
- Computazione come costruzione di una dimostrazione di una affermazione (goal)

- Base formale
  - Calcolo dei predicati del primo ordine ma con limitazione nel tipo di formule (clausole di Horn)
  - Utilizzo di particolari tecniche per la dimostrazione di teoremi (meccanismo di Risoluzione)

Prolog

- Acronimo per **PRO**gramming in **LOG**ic. Nato nel 1973.
- Basato su una restrizione della logica del primo ordine
- Stile dichiarativo
- Prolog è usato per determinare se una certa affermazione è vera o no e, se è vera, quali vincoli sui valori attribuibili alle variabili hanno generato la risposta
  - Nuovi sviluppi
    - Constraint Logic Programming (CLP)

Formule ben formate e forma normale "a clausole"

- Ogni formula ben formata (*fbf*, o well-formed formula, *wff*) di un linguaggio logico del primo ordine può essere riscritta in forma normale a clausole
- Vi sono due forme normali a clausole
  - **Forma normale congiunta** (conjunctive normal form - CNF)
    - La formula è una ***congiunzione*** di ***disgiunzioni*** di predicati o di negazioni di predicati (**letterali** positivi e **letterali** negativi)
  - **Forma normale disgiunta** (disjunctive normal form - DNF)
    - La formula è una ***disgiunzione*** di ***congiunzioni*** di predicati o di negazioni di predicati (**letterali** positivi e **letterali** negativi)

Formule ben formate e forma normale "a clausole"

- **Forma normale congiunta** (conjunctive normal form - CNF)

$$\bigwedge_i \left(\bigvee_j L_{ij} \right)$$

- **Forma normale disgiunta** (disjunctive normal form - DNF)

$$\bigvee_j \left(\bigwedge_i L_{ij} \right)$$

dove

$$L_{ij} \; ..P_{ij}(x,y,\cdots,z) \text{ o } L_{ij} \; ..\downarrow Q_{ij}(x,y,\cdots,z)$$

Forma Normale Congiuntiva

- Consideriamo una wff in CNF $\underbrace{\bigwedge_i \left(\bigvee_j L_{ij} \right)}_{\text{clausole}}$
- Esempi

$$\underbrace{(p(x) \vee q(x,y) \vee \downarrow t(z))}_{\text{clausola 1}} \wedge \underbrace{(p(w) \vee \downarrow s(u) \vee \downarrow r(v))}_{\text{clausola 2}}$$

$$\underbrace{(\downarrow t(z))}_{\text{clausola 1}} \wedge \underbrace{(p(w) \vee \downarrow s(u))}_{\text{clausola 2}} \wedge \underbrace{(p(x) \vee s(x) \vee q(y))}_{\text{clausola 3}}$$

se scartiamo il simbolo di congiunzione, rimaniamo con solo le clausole disgiuntive (primo esempio):

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

Forma Normale Congiuntiva

- Le clausole relative al primo esempio sono anche riscrivibili come

$$t(z) \leftarrow p(x) \wedge q(x,y)$$
$$s(u) \mid r(v) \leftarrow p(w)$$

  ovvero, un insieme di formule in CNF è riscrivibile come un insieme (congiunzione) di implicazioni

- Le clausole che hanno al più un solo letterale positivo (con o senza letterali negativi) si chiamano **clausole di Horn**
  - Non tutte le fbf possono essere trasformate in un insieme di clausole di Horn
  - **I programmi Prolog sono collezioni di clausole di Horn**
    - Grazie a questa restrizione, Kowalski, nel 1974, produsse una interpretazione procedurale di un insieme di clausole (di Horn) che è alla base della semantica di tutti i sistemi Prolog (Kowalski, R., *Predicate Logic as Programming Language*, in Proceedings IFIP Congress, Stockholm, North Holland Publishing Co., 1974, pp. 569-574)

Prolog – Linguaggio dichiarativo

- Non contiene istruzioni (quasi)
- Contiene solo fatti e regole (clausole di Horn)
  - Fatti
    - Asserzioni vere nel contesto che stiamo descrivendo
  - Regole
    - Ci danno gli strumenti per dedurre nuovi fatti da quelli esistenti

- Un programma Prolog ci dà informazioni su un sistema ed è normalmente chiamato **base di conoscenza** (**knowledge base**)
- Un programma Prolog non si «esegue» ma si «**interroga**» (**queried**)
  - Ci si chiede: questi fatti sono veri?
  - Il Programma risponderà solo **SI** o **NO** alla domanda

Programmi Prolog

- **Sintassi**: un programma Prolog è costituito da un insieme di clausole della forma

```
a.                          % FATTO o ASSERZIONE
c :- b1, b2, …, bn.         % REGOLA

:- q1, q2, …, qm.           % GOAL
?- q1, q2, …, qm.           % QUERY
```

- In cui `a`, b_i, `c`, e q_i sono **termini** (composti)
- Notare che in molte implementazioni il prompt Prolog è anche un operatore che chiede al sistema di valutare il **goal**, in questo caso una congiunzione di termini

Termini

- Le espressioni del Prolog sono chiamate **TERMINI**. Esistono diversi tipi di termini:
  - Atomi
    - Numeri
    - …
  - Variabili
  - Una composizione di termini ⇒ termine composto (simbolo di *funtore* + uno o più argomenti)

Atomi

- Un **atomo** è:
  - Una sequenza di caratteri alfanumerici, che inizia con un carattere **minuscolo** (può contenere il carattere "_" underscore)
  - Qualsiasi cosa racchiusa tra apici (' ')
  - Un numero
  - Una stringa (SWI Prolog)
  - …

Variabili (logiche)

- Una **variabile** (logica) è una sequenza alfanumerica che inizia con un carattere **maiuscolo** o con il carattere _ (underscore)
- Le variabili (notare il plurale) composte solo dal simbolo _ sono chiamate **indifferenza** (*don't care* o *any*) o **anonime**
- Le variabili vengono **istanziate** (legate a un valore) con il procedere del programma (nella dimostrazione del teorema)

Termini composti

- Una composizione di termini consiste in:
  - Un **funtore** (simbolo di funzione o di predicato definito come atomo)
  - Una sequenza di termini racchiusi tra parentesi tonde e separati da virgole. Questi sono chiamati argomenti del funtore
    - non ci deve mai essere uno spazio tra il funtore e la parentesi di sinistra; questo per via di caratteristiche molto sofisticate del sistema di parsing di Prolog (cfr., **operatori**)

Esempi di termini

- Validi

```
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, []))))
```

- 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)
```

Prolog: i fatti o predicati

- Un fatto (predicato) consiste in:
  - Un nome di predicato, ad esempio **`fattoriale`**, **`genitore`**, **`uomo`** o **`animale`**; deve iniziare con una lettera minuscola.
  - Zero o più argomenti come **`maria`**, **`42`** o **`cane`**.
    - Da notare che i fatti (e le regole e le domande) devono essere terminati da un punto "**`.`**" .

Fatti: esempi

- Esempi

```
libro(kowalski, prolog).
libro(yoshimoto, kitchen).
donna(yoshimoto).
uomo(kowalski).
animale(cane).
animale(trota).
ha_le_squame(trota).
la_risposta(42).
```

Le regole

- In Prolog si usano le **regole** quando si vuole esprimere che un certo fatto dipende da un insieme di altri fatti.

- Per esprimere in linguaggio naturale questa dipendenza usiamo la parola «**se**» («**if**» in inglese)
  - *Uso l'ombrello **se** piove.*
  - *Gino compra il vino **se** è meno caro della birra.*

Le regole

- Le regole sono anche usate per esprimere definizioni:
  - X è un pesce **se**:
    - *X è un animale*
    - *X ha le squame*
- oppure
  - X è sorella di Y **se**:
    - *X è di sesso femminile*
    - *X e Y hanno gli stessi genitori*

Le regole

- In Prolog una regola consiste di una **testa** (head) e di un **corpo** (body)
  - Testa e corpo sono collegati dall'operatore `:-`
  - La testa di una regola corrisponde al **conseguente** di un'implicazione logica
  - Il corpo di una regola corrisponde all'**antecedente** di un'implicazione logica
  - Le regole Prolog corrispondono alle **clausole di Horn**, ovvero hanno un solo termine (predicato) come conseguente
    - L' operatore Prolog '`:-`' esprime il "se" (*implicazione*)
    - L'operatore Prolog '`,`' equivale a "e" (*and*, o *congiunzione*)

Regole

- Esempi
  - La frase "*un pesce è un animale che ha le squame*" in Prolog diventa:

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

  - La frase *"Gigi ama chiunque ami il vino"* in Prolog diventa:

```
ama(gigi, X) :- ama(X, vino).
```

  - La frase *"Gigi ama le donne a cui piace il vino"*, in Prolog sarà:

```
ama(gigi, X) :- donna(X), ama(X, vino).
```

Prolog e relazioni definite da più regole

- Una relazione (ad esempio **`genitore`**) può essere definita da più regole (ovvero da più clausole) aventi lo stesso predicato come conclusione.

- Esempio

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

- Le regole (ed i fatti) sono implicitamente connesse dall'operatore logico di congiunzione («**and**»); se non si sono commessi errori – per l'appunto – **logici**, **entrambe** le implicazioni qua sopra sono da ritenersi **vere**

Regole – La ricorsione

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

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

Operatori logici

- L'operatore logico **AND** inserito in una regola viene espresso mediante la virgola '`,`'
- L'operatore logico **OR** inserito in una regola viene invece espresso mediante il punto e virgola '`;`'

Sintassi

- Ricordatevi che
  - ogni fatto o regola o funzione DEVE terminare con un punto '.'
  - ogni variabile DEVE iniziare con una maiuscola
  - I commenti si inseriscono dopo un «`%`» (commento di linea) o tra «`/*`» e «`*/`» (come in C/C++ e Java)

```
%%% Questa è una linea di commento
f(a,b). % Solo questa parte di riga è un commento.

%%% Ecco un commento
%%% su più righe.
%%% Anche questo è un commento.

/* Questo commento può essere
 * scritto su più righe
 */
```

L'interprete Prolog: interrogazioni (queries o goals)

- Una volta che le regole ed i fatti sono scritte e caricate nell'interprete (vedremo come), eseguire un programma Prolog significa **interrogare** l'**interprete**
- Una volta fatto partire, di solito, l'interprete Prolog ci presenta il prompt

```
?-
```

- Un esempio di query diventa quindi

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

- Dati i fatti e le regole che abbiamo visto precedentemente, il Prolog risponde «Yes» o «true»
  - *Interrogare il programma equivale a richiedere la dimostrazione di un teorema*

Le variabili delle interrogazioni

- Le interrogazioni possono contenere variabili interpretate come variabili *esistenziali*
- Le variabili sono *istanziate* quando il Prolog prova a rispondere alla domanda
- Tutte le variabili istanziate vengono mostrate nella risposta.

- Esempi

```
?- libro(kowalski, LINGUAGGIO).
Yes
   LINGUAGGIO = prolog

?- libro(AUTORE, prolog).
Yes
   AUTORE = kowalski

?- libro(AUTORE, LINGUAGGIO).
Yes
   AUTORE = kowalski
   LINGUAGGIO = prolog
```

Unificazione: introduzione

- L'operazione di istanziazione 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
- Questo insieme di sostituzioni (brevemente: sostituzione o assegnamento) permette di rendere «**uguali**» i due termini

Unificazione: introduzione

- Tradizionalmente la procedura di unificazione costruisce un insieme di sostituzioni chiamato «most general unifier» ed indicato con *Mgu*
- Una sostituzione è indicata come una sequenza (ordinata) di coppie *variabile/valore*
- **Esempi**

Mgu(`42, 42`)	⇒ {}
Mgu(`42, X`)	⇒ {`X`/`42`}
Mgu(`X, 42`)	⇒ {`X`/`42`}
Mgu(`foo(bar, 42), foo(bar, X)`)	⇒ {`X`/`42`}
Mgu(`foo(Y, 42), foo(bar, X)`)	⇒ {`Y`/`bar`, `X`/`42`}
Mgu(`foo(bar(42), baz), foo(X, Y)`)	⇒ {`X`/`bar(42)`, `Y`/`baz`}
Mgu(`foo(X), foo(bar(Y))`)	⇒ {`X`/`bar(Y)`, `Y`/`_G001)`}

- Notare l'ultimo esempio con **ridenominazione** di variabili

Unificazione: introduzione

- Il "most general unifier" non è nient'altro che il risultato finale della procedura di valutazione - ovvero di prova - del Prolog
- Il modo più semplice per vedere come la procedura di unificazione funziona è di usare l'operatore Prolog = (detto, per l'appunto, *operatore di unificazione*)

- **Esempi**

```
?- 42 = 42.
   Yes

?- 42 = X.
   X = 42
   Yes

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

Unificazione: introduzione

- **Esempi**

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

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

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

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

Diverse rappresentazioni di dati ed interrogazioni

- Consideriamo il seguente esempio: vogliamo descrivere un insieme di fatti riguardanti i corsi offerti dal dipartimento (esempio dal capitolo 2 di “Art of Prolog”)

- **Possibilità 1**
  - Tutte le informazioni sono concentrate in una relazione con 6 campi

```
corso(linguaggi, lunedì, '9:30', 'U4', 3, antoniotti).
corso(biologia_computazionale, lunedì, '14:30', 'U14', t023, antoniotti).
```

  - A partire da questa definizione possiamo poi costruire altri predicati

```
aula(Corso, Edificio, Aula) :-
            corso(Corso, _, _, Edificio, Aula, _).
docente(Corso, Docente) :-
            corso(Corso, _, _, _, _, Docente).
```

Diverse rappresentazioni di dati ed interrogazioni (cont…)

- **Possibilità 2**
  - Tutte le informazioni sono concentrate in una relazione con 4 campi; le informazioni sono concentrate in termini funzionali che rappresentano informazioni raggruppate logicamente

```
corso(linguaggi, orario(lunedì, '9:30'), aula('U4', 3), antoniotti).
corso(biologia_computazionale,
      orario(lunedì, '14:30'),
      aula('U4', 3),
      antoniotti).
```

  - A partire da questa definizione possiamo poi costruire altri predicati

```
aula(Corso, Edificio, Aula) :- corso(Corso, _, aula(Edificio, Aula), _).
docente(Corso, Docente) :- corso(Corso, _, _, Docente).


%%%% oppure…

aula(Corso, Luogo) :- corso(Corso, _, Luogo, _).
```

Diverse rappresentazioni di dati ed interrogazioni (cont…)

- **Possibilità 3**
  - I predicati che abbiamo definito dalle relazioni con 6 o 4 campi possono essere ricodificate con predicati **binari** (cfr., le **triple** XML usate negli strumenti e nella ricerca sulla «semantic web»)

```
giorno(linguaggi, martedì).
orario(linguaggi, '9:30').
edificio(linguaggi, 'U4').
aula(linguaggi, 3).
docente(linguaggi, antoniotti).
```

  - La relazioni a 6 o 4 argomenti possono essere ricostruite in a partire da queste relazioni binarie
    - La costruzione di schemi RDF/XML (e, a volte, SQL) corrisponde a questa operazione di ri-rappresentazione con **triple**

NB, la rappresentazione qui sopra è insufficiente per rappresentare completamente un orario; perché?

Le liste in Prolog

- Si definisce una lista in Prolog racchiudendo gli elementi (termini e/o variabili logiche) della lista tra parentesi quadre '`[`' e '`]`' e separandoli da virgole.
- Gli elementi di una lista in Prolog possono essere termini qualsiasi o liste
- `[]` indica la lista vuota.

Le liste: testa e coda

- Ogni lista non vuota può essere divisa in due parti: una testa ed una coda.
  - La testa è il primo elemento della lista
  - La coda rappresenta tutto il resto ed **è sempre una lista**

- **Esempi**

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

L'operatore |

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

- **Esempi** (notare la convenzione `Ys`, `Zs`)

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

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

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

La lista vuota `[]`

- La lista vuota `[]` in prolog viene gestita come una lista speciale.

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

L'interprete Prolog: `consult`

- La base di conoscenza è *nascosta* ed è accessibile solo tramite opportuni comandi o tramite ambiente di programmazione)
- Ovviamente è necessario poter **inizializzare** o **caricare** un insieme di fatti e regole nell'ambiente Prolog
- Il comando principale che assolve questa funzione è **`consult`**
  - Il comando **`consult`** appare come un predicato da valutare (un goal) e prende almeno un termine che denota un file come argomento
  - Il file deve contenere un insieme di fatti e regole

- **Esempi**

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

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

L'interprete Prolog: `consult`

- Il predicato **`consult`** può essere usato anche per inserire fatti e regole direttamente alla console usando il termine speciale **`user`**

- **Esempi**

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

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

(*) *La semantica di* **`consult`** *e* **`reconsult`** *è più complicata secondo lo standard ISO-Prolog*

L'interprete Prolog: `reconsult`

- Il predicato **`reconsult`** deve invece essere usato quando si vuole ***ri***caricare un file (ovvero un data o knowledge base) nell'ambiente Prolog
- L'effetto è di prendere i predicati presenti nel file, rimuoverli completamente dal data base interno e di reinstallarli utilizzando le nuove definizioni (*)

- **Esempi**

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

%%% A questo punto la base di dati Prolog contiene i due
%%% fatti inseriti manualmente.

?- friends(zaphod, W).
W = trillian
Yes
```

Sommario

- Introduzione al Prolog
- **Termini**
  - Fatti
  - Regole
- **Unificazione**
- Rappresentare dati in Prolog
- **Liste**
- Minima introduzione all'ambiente Prolog

Linguaggi di Programmazione 2021-2022

Prolog e Programmazione Logica III

Marco Antoniotti
Gabriella Pasi
Rafael Peñaloza

Modello di esecuzione Prolog

```
p :- q, r.
```

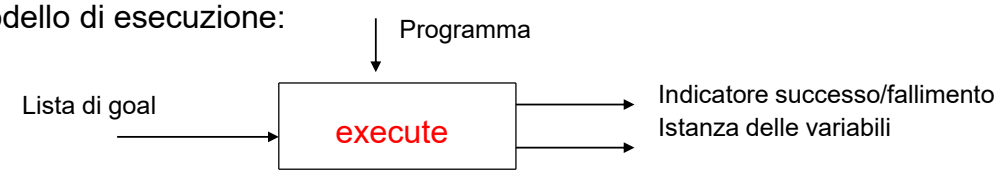
Interpretazione dichiarativa:

`p` è vero se sono veri `q` e `r`.

Interpretazione procedurale:

Il problema `p` può essere scomposto nei sottoproblemi `q` e `r`.

Modello di esecuzione:



Un goal può essere visto come una chiamata ad una procedura.

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

Estensioni

- Per rendere il Prolog un linguaggio effettivamente utilizzabile vengono aggiunti
  - Già introdotti
    - Notazione per le liste
    - Meccanismi per il caricamento del codice Prolog
  - Da vedere
    - Meccanismi di controllo del backtracking
    - Operazioni aritmetiche
    - Trattamento della negazione
    - Possibilità di manipolare e confrontare le strutture dei termini
    - Predicati meta-logici ed extra-logici
    - Predicati di input/output
    - Meccanismi per modificare/accedere alla base di conoscenza

Il controllo di esecuzione di un programma

- Come abbiamo intuito, le clausole nel data base di un programma Prolog vengono considerate “da sinistra, verso destra” e “dall’alto al basso”

- Se un (sotto)goal fallisce, allora il dimostratore Prolog sceglie un’alternativa, scandendo “dall’alto” verso “il basso” la lista delle clausole

- Il Prolog mette a disposizione un predicato speciale, chiamato **cut** (*taglio*, scritto con il solo simbolo esclamativo `!`) per controllare questa sequenza di scelte
  - Il cut è molto complesso da interpretare (non ha un’interpretazione “logica” ma solo procedurale)
  - La sua importanza per il Prolog non può essere sottovalutata
  - Per capire come funziona è necessario avere un’idea più approfondita del funzionamento del dimostratore Prolog (ovvero della sua “macchina virtuale”)

Il predicato cut ‘!’

- Consideriamo la seguente clausola generica con cut

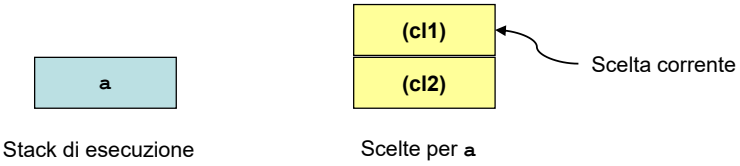
  C = `a :-` b_1`,` b_2`, …,` b_k`, !,` b_{k+1}`, …,` b_n`.`

- L’effetto del cut è il seguente
  - Se il goal corrente `G` unifica con `a` e b_1, …, b_k hanno successo, allora il dimostratore si impegna inderogabilmente alla scelta di C per dimostrare `G`
  - Ogni clausola alternativa (successiva, in basso) per `a` che unifica con `G` viene ignorata
  - Se un qualche b_j con `j > k` fallisse, il backtracking si fermerebbe al cut `!`
    - Le altre scelte per i b_i con `i` $\leq$ `k` sono di conseguenza rimosse dall’albero di derivazioni
  - Quando il backtracking raggiunge il cut, allora il cut fallisce e la ricerca procede dall’ultimo punto di scelta prima che `G` scegliesse C

Modello di Esecuzione di un Programma Prolog

- Considerate il seguente programma Prolog
  (cl1) `a :- p, b.`
  (cl2) `a :- p, c.`
  (cl3) `p.`

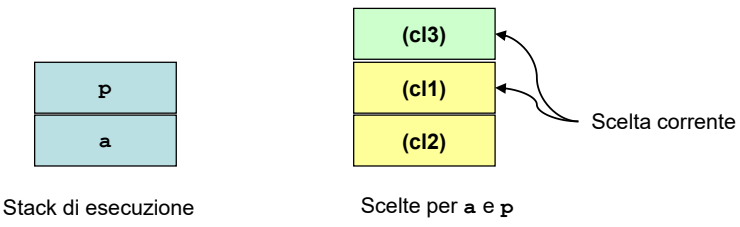
- Considerate la valutazione della query
  `?- a.`
- Lo stato interno del sistema Prolog diventa il seguente



Modello di Esecuzione di un Programma Prolog

- Considerate il seguente programma Prolog
  (cl1) `a :- p, b.`
  (cl2) `a :- p, c.`
  (cl3) `p.`

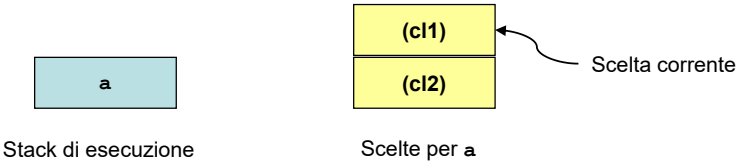
- Si mette `p` in cima allo stack



Modello di Esecuzione di un Programma Prolog

- Considerate il seguente programma Prolog
  (cl1) `a :- p, b.`
  (cl2) `a :- p, c.`
  (cl3) `p.`

- La valutazione di `p` ha successo

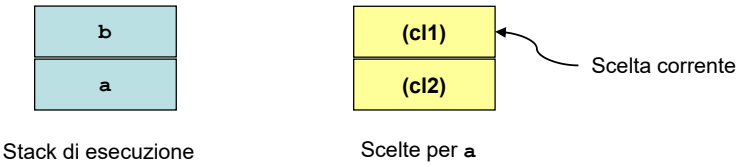


Modello di Esecuzione di un Programma Prolog

- Considerate il seguente programma Prolog
  (cl1) `a :- p, b.`
  (cl2) `a :- p, c.`
  (cl3) `p.`

- Quindi si inserisce `b` in cima allo stack

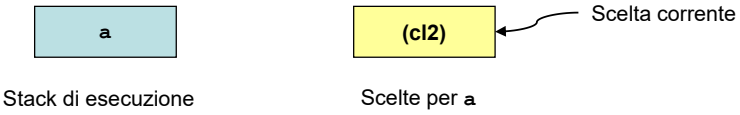
La valutazione di `b` fallisce; quindi viene attivato il meccanismo di **backtracking**



Modello di Esecuzione di un Programma Prolog

- Considerate il seguente programma Prolog
  (cl1) `a :- p, b.`
  (cl2) `a :- p, c.`
  (cl3) `p.`

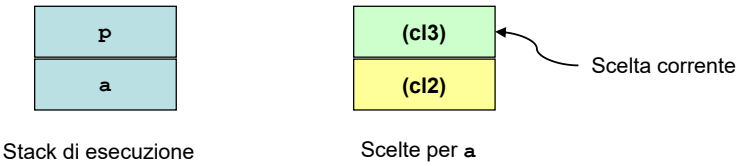
- Per proseguire con la valutazione di `?- a.` si passa a considerare la seconda clausola
- Lo stato interno del sistema Prolog diventa il seguente



Modello di Esecuzione di un Programma Prolog

- Considerate il seguente programma Prolog
  (cl1) `a :- p, b.`
  (cl2) `a :- p, c.`
  (cl3) `p.`

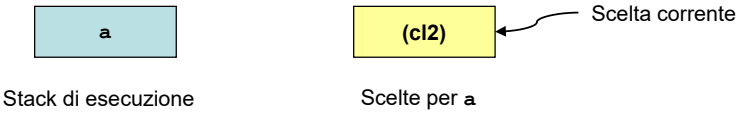
- Si mette `p` in cima allo stack



Modello di Esecuzione di un Programma Prolog

- Considerate il seguente programma Prolog
  (cl1) `a :- p, b.`
  (cl2) `a :- p, c.`
  (cl3) `p.`

- La valutazione di `p` ha successo

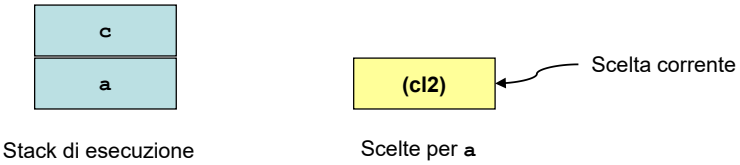


Modello di Esecuzione di un Programma Prolog

- Considerate il seguente programma Prolog
  (cl1) `a :- p, b.`
  (cl2) `a :- p, c.`
  (cl3) `p.`

- Quindi si inserisce `c` in cima allo stack

La valutazione di `c` fallisce; quindi viene attivato il meccanismo di **backtracking**; ma visto che non ci sono più clausole anche `a` fallisce e quindi lo stack di esecuzione si svuota



Modello di Esecuzione di un Programma Prolog

- Due pile (stacks):

  1. Pila di esecuzione che contiene i record di attivazione delle varie “procedure” (ovvero le sostituzioni per l’unificazione delle varie regole)

  2. Pila di **backtracking** che contiene l’insieme dei “punti di scelta”; ad ogni fase della valutazione questa pila contiene dei “puntatori” alle scelte *aperte* nelle fasi precedenti della dimostrazione

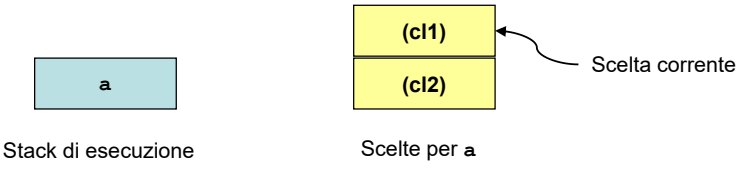
Modello di Esecuzione di un Programma Prolog

- Considerate il seguente programma Prolog

```
(cl1)  a :- p, b.
(cl2)  a :- r.
(cl3)  p :- q.
(cl4)  p :- r.
(cl5)  r.
```

- Considerate ancora la valutazione della query

```
?- a.
```

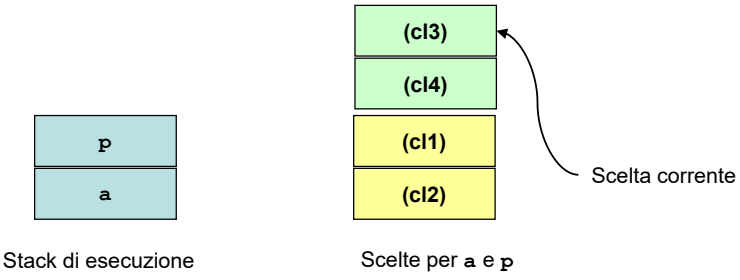


Modello di Esecuzione di un Programma Prolog

- Considerate il seguente programma Prolog

```
(cl1)  a :- p, b.
(cl2)  a :- r.
(cl3)  p :- q.
(cl4)  p :- r.
(cl5)  r.
```

- Si mette `p` in cima allo stack e lo stato interno del sistema Prolog diventa il seguente



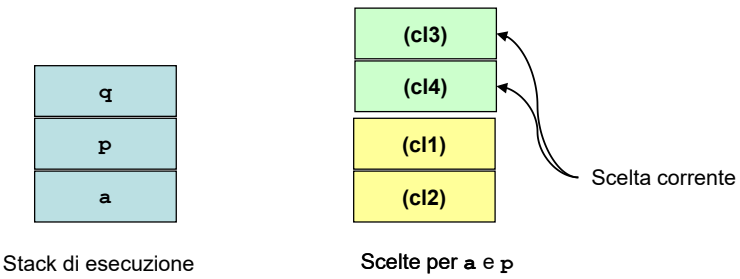
Modello di Esecuzione di un Programma Prolog

- Considerate il seguente programma Prolog

```
(cl1)  a :- p, b.
(cl2)  a :- r.
(cl3)  p :- q.
(cl4)  p :- r.
(cl5)  r.
```

- Si prosegue inserendo `q` in cima allo stack di esecuzione

La valutazione di `q` fallisce ⇒ **backtracking**

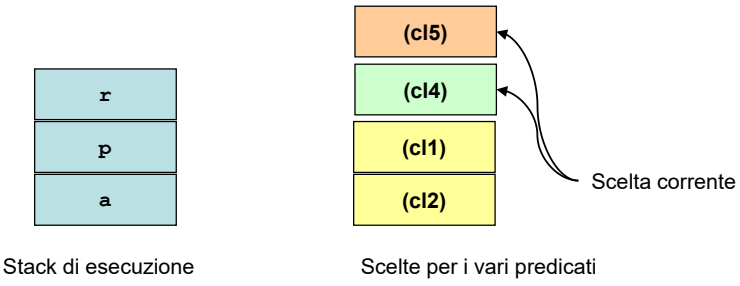


Modello di Esecuzione di un Programma Prolog

- Considerate il seguente programma Prolog

```
(cl1)  a :- p, b.
(cl2)  a :- r.
(cl3)  p :- q.
(cl4)  p :- r.
(cl5)  r.
```

- Si prosegue inserendo `r` in cima allo stack di esecuzione



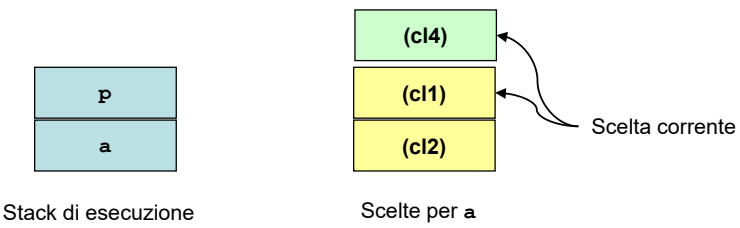
Modello di Esecuzione di un Programma Prolog

- Considerate il seguente programma Prolog

```
(cl1)  a :- p, b.
(cl2)  a :- r.
(cl3)  p :- q.
(cl4)  p :- r.
(cl5)  r.
```

- Si prosegue inserendo `r` in cima allo stack di esecuzione

La valutazione di `r` ha successo, e quindi anche `p`



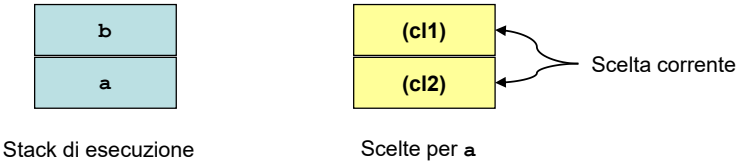
Modello di Esecuzione di un Programma Prolog

- Considerate il seguente programma Prolog

```
(cl1)  a :- p, b.
(cl2)  a :- r.
(cl3)  p :- q.
(cl4)  p :- r.
(cl5)  r.
```

- Si inserisce `b` in cima allo stack di esecuzione

La valutazione di `b` fallisce; si cambia clausola



Modello di Esecuzione di un Programma Prolog

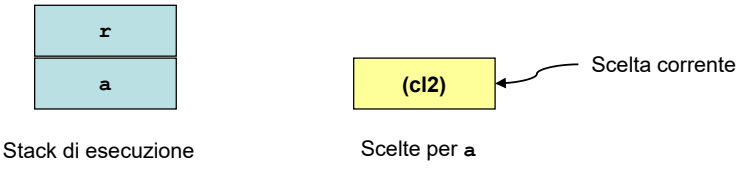
- Considerate il seguente programma Prolog

```
(cl1)  a :- p, b.
(cl2)  a :- r.
(cl3)  p :- q.
(cl4)  p :- r.
(cl5)  r.
```

- Si inserisce `r` in cima allo stack di esecuzione

`r` ha successo

Gli stack sono ora vuoti e la query iniziale **`?- a`** ha successo



Modello di Esecuzione di un Programma Prolog

Cosa succede se abbiamo delle queries con il **cut** (**!**)?

Modello di esecuzione: il trattamento del cut

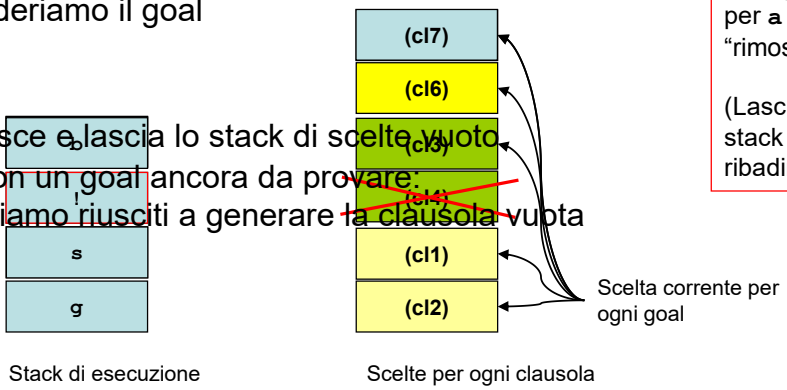
- Considerate il seguente programma Prolog

```
(cl1)  g :- a.
(cl2)  g :- s.
(cl3)  a :- p, !, b.
(cl4)  a :- r.
(cl5)  p :- q.
(cl6)  p :- r.
(cl7)  r.
```

- Consideriamo il goal

```
:- g.
```

`s` fallisce e lascia lo stack di scelte vuoto, ma con un goal ancora da provare: non siamo riusciti a generare la clausola vuota



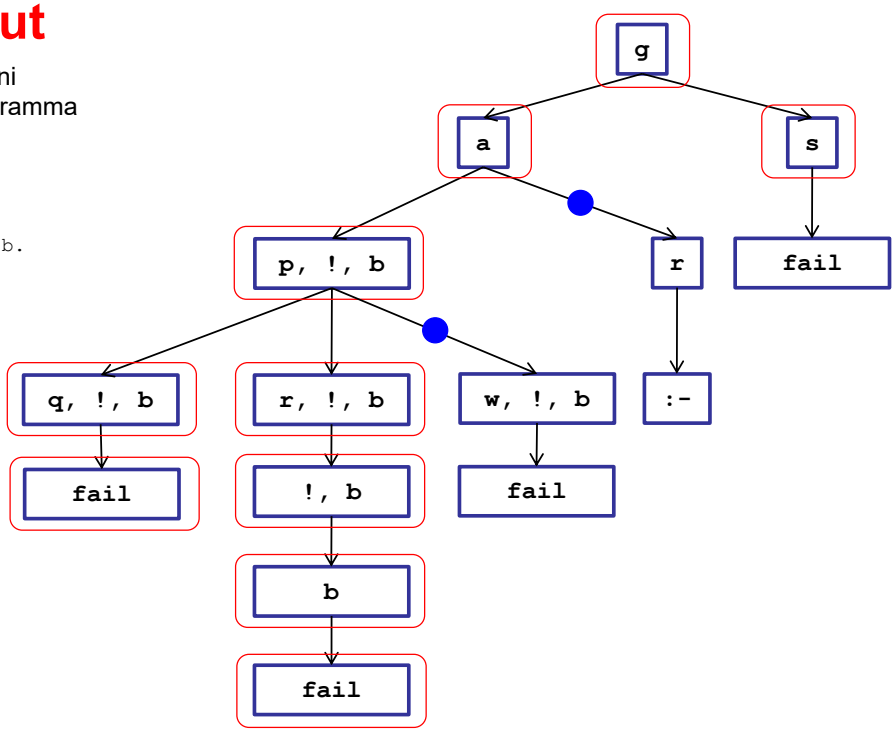
Effetto del **cut** (`!`): tutti i punti di scelta per `a` (e per `p`) sono "rimossi" dallo stack

(Lasciamo il `!` sullo stack giusto per ribadire il suo effetto).

Modello di esecuzione: il trattamento del cut

- Albero di derivazioni SLD/LM per il programma con goal `:- g.`

```
(cl1)   g :- a.
(cl2)   g :- s.
(cl3)   a :- p, !, b.
(cl4)   a :- r.
(cl5)   p :- q.
(cl6)   p :- r.
(cl7)   p :- w.
(cl8)   r.
```



Due tipi di cut

- Si possono distinguere due tipi di cut (ovvero due usi del predicato cut)
- Green Cuts
  utili per esprimere “determinismo” (e quindi per rendere più efficiente il programma)
- Red Cuts
  usati per soli scopi di efficienza, hanno per caratteristica principale quella di omettere alcune condizioni esplicite in un programma e, *soprattutto*, quella di modificare la semantica del programma equivalente senza cuts
  - Quindi sono tendenzialmente **indesiderabili** (anche se, a volte, utili)

Esempio 1

Consideriamo il seguente programma che serve a fare il "merge" di due liste ordinate

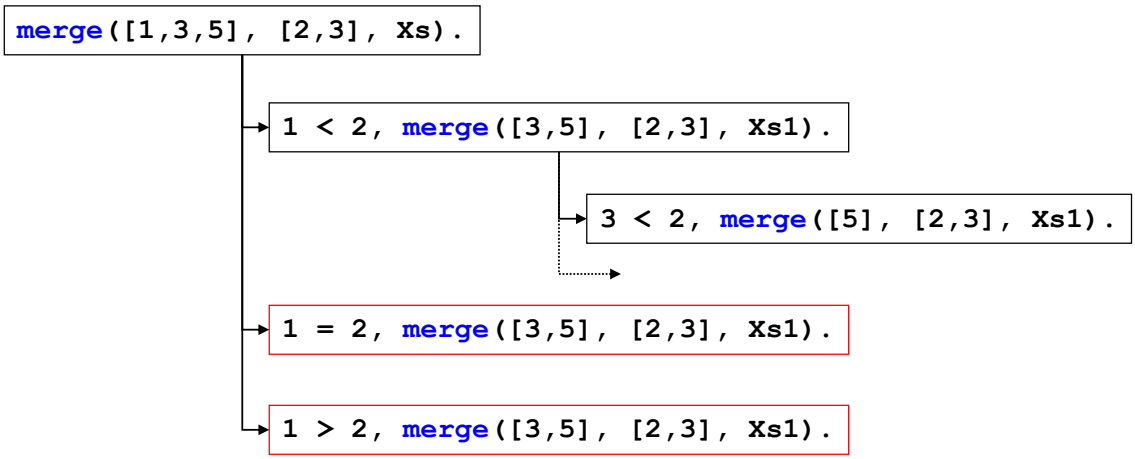
```
merge([X | Xs], [Y | Ys], [X | Zs]) :-
  X < Y,
  merge(Xs, [Y | Ys], Zs).
merge([X | Xs], [Y | Ys], [X, Y | Zs]) :-
  X = Y,
  merge(Xs, Ys, Zs).
merge([X | Xs], [Y | Ys], [Y | Zs]) :-
  X > Y,
  merge([X | Xs], Ys, Zs).
merge([], Ys, Ys).
merge(Xs, [], Xs).
```

Esempio 2

- Consideriamo anche il seguente programma che serve a controllare se quale è il minimo tra due numeri

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

Esecuzione di merge



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

Esecuzione di merge

- Considerate anche la query seguente

```
?- merge([], [], Xs).
Xs = [];
Xs = [];
No
```

- Abbiamo una soluzione di troppo!

Determinismo e green cuts

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

- Come già visto i cuts che servono per esplicitare questo determinismo vengono detti **green cuts**

Esempio 1: merge con green cuts

Consideriamo il seguente programma che serve a fare il "merge" di due liste ordinate

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

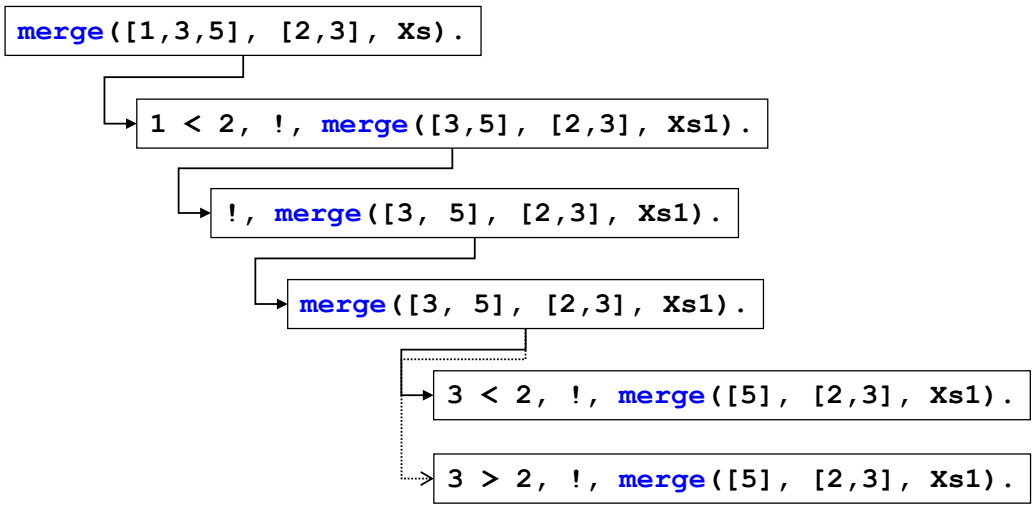
Esecuzione di merge con green cuts

- La query con troppe soluzioni ora diventa

```
?- merge([], [], Xs).
Xs = [];
No
```

- Ovvero abbiamo esattamente il numero di soluzioni che ci interessa!

Esecuzione di merge



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

Esempio 2: minimum con green cuts

- Il programma diventa

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

- Notate come il secondo cut sia in realtà ridondante
- Viene comunque messo nel programma per motivi di simmetria

Esempio 2: minimum con red cuts

- Riconsideriamo il programma minimum

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

- Non solo il secondo cut è ridondante
- Una volta che il programma ha fallito la prima clausola (ovvero il test `X =< Y`) al sistema Prolog non rimane che controllare la clausola seguente

- Premature optimization is the root of all evil (cit. Dijkstra)
- Some optimizations, even if not premature, are still evil

Esempio 2: minimum con red cuts

- Il programma potrebbe essere riscritto in maniera non simmetrica nel seguente modo

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

- In questo caso il cut è red, dato che serve solo (?!?) a tagliare delle soluzioni
- Non solo, il goal `minimum(2, 5, 5)` viene verificato
  - Quindi il programma è scorretto

- **I red cuts vanno usati con estrema cura**

Sommario

- Modello di esecuzione Prolog
  - Attraversamento di un albero di derivazioni **SLD/left-most**
    - In profondità ("**depth-first**")
    - Con "**backtracking**"

- Controllo dell'attraversamento
  - "Predicato" **taglio** ("**cut**"), indicato con '`!`'
    - Rimozione di rami dall'albero di derivazioni

Linguaggi di Programmazione 2021-2022

Prolog e Programmazione Logica II

Marco Antoniotti
Gabriella Pasi
Rafael Peñaloza

Prolog

- Lavora su strutture ad **albero**
  - *anche i programmi sono strutture dati manipolabili* (mediante predicati “extra-logici”)
- Utilizza ricorsione (non c’è una nozione semplice di assegnamento)

- Metodologia di programmazione
  - concentrarsi sulla specifica del problema rispetto alla strategia di soluzione
- Un programma Prolog è un insieme di **clausole di Horn** che rappresentano
  - **fatti** riguardanti gli oggetti in esame e le relazioni che intercorrono tra di loro
  - **regole** sugli oggetti e sulle relazioni (*se* … *allora* …)
  - **interrogazioni** (**goals** o **queries**; clausole «senza testa»), sulla base della conoscenza definita

Clausole

- **Implicazione**
  - Sappiamo che $\neg B \vee A$ corrisponde a $B \rightarrow A$, oppure «A implicato da B» (che si legge “A se B”), scritto anche come $A \leftarrow B$
  - Data una clausola

    $A_1 \vee A_2 \vee \ldots \vee A_n \vee \neg B_1 \vee \neg B_2 \vee \ldots \vee \neg B_m$

    usando la nota **formula di riscrittura** (teorema di De Morgan)

    $(A_1 \vee A_2 \vee \ldots \vee A_n) \vee \neg(B_1 \wedge B_2 \wedge \ldots \wedge B_m)$

    da cui otteniamo

    $(A_1 \vee A_2 \vee \ldots \vee A_n) \leftarrow (B_1 \wedge B_2 \wedge \ldots \wedge B_m)$
  - Ovviamente anche il percorso inverso è valido
  - **Definizione:** le A e le B nelle formule precedenti si dicono **letterali**, **negativi** quelli negati e **positivi** gli altri

Clausole di Horn

- Una **clausola** di **Horn** ha – al più – un solo letterale positivo

  $A \leftarrow B_1 \wedge B_2 \wedge \ldots \wedge B_m$

- In particolare possiamo classificare le clausole di Horn nel modo seguente
  - Fatti $A \leftarrow$
  - Regole $A \leftarrow B_1 \wedge B_2 \wedge \ldots \wedge B_m$
  - Goals $\leftarrow B_1 \wedge B_2 \wedge \ldots \wedge B_m$
  - Contraddizione $\leftarrow$
- E, come già abbiamo visto, in Prolog questi diventano
  - Fatti `A.`
  - Regole `A :- B1, B2, …, Bm.`
  - Goals `:- B1, B2, …, Bm.`
  - Contraddizione `fail`

Un programma logico

- Un programma logico che manipola la rappresentazione «unaria» dei numeri naturali (l'insieme $\mathbb{N}$)

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

- Possiamo interpretare **`s(N)`** come il *successore* del numero **`N`**
- Quindi **`0`**, **`s(0)`**, **`s(s(0))`**, **`s(s(s(0)))`**… rappresentano 0, 1, 2, 3…
- Questo programma **definisce** la somma fra due numeri naturali (rappresentati in "unario")

Un programma logico

- Possiamo interrogare il «programma» nel modo seguente

```
∃X sum(s(0), 0, X)              {X / s(0)}
∃W sum(s(s(0)), s(0), W)        {W / s(s(s(0)))}
```

- Mediante il procedimento di negazione e di trasformazione in sintassi Prolog otteniamo

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

Sostituzioni

- Nell'esempio precedente abbiamo visto le **sostituzioni** `{W / s(s(s(0)))}` e `{N / s(0)}`
- Una sostituzione ci dice con che «valori» (che possono essere altre variabili) possiamo sostituire le variabili in un termine
- Di solito si denota una sostituzione nel modo seguente

$$\sigma = \{X_1/v_1, X_2/v_2, ..., X_k/v_k\}$$

- Una sostituzione può essere considerata come una *funzione*, applicabile ad un termine (T è l'insieme dei termini)

$$\sigma : T \rightarrow T$$

- Esempio (data la sostituzione σ = `{X/42, Y/foo(s(0))}`)

$$\sigma(\texttt{bar(X, Y)}) = \texttt{bar(42, foo(s(0)))}$$

Esecuzione di un programma

- Una computazione corrisponde al tentativo di dimostrare, tramite la regola di risoluzione, che una formula segue logicamente da un programma (è un teorema).
- Inoltre, si deve determinare una sostituzione per le variabili del goal (detto anche **query**) per cui la query segue logicamente dal programma.
- Dato un programma **P** e la query

  `:- p(t1, t2, … , tm).`

  se `X1`, `X2`,…, `Xn` sono le variabili che compaiono in `t1`, `t2`,…, `tm`, il significato della query è

  ∃ `X1`, `X2`,…, `Xn . p(t1, t2, … , tm)`

  e l'obiettivo è quello di trovare una sostituzione

  s = {`X1`/s1, `X2`/s2, …, `Xn`/sn}

  dove gli s_i sono termini tali per cui **P** ⊢ **s**`[p(t1, t2, … , tm)]`

Esecuzione di un programma

- Dato un insieme di clausole di Horn è possibile derivare la clausola vuota solo se c`è n'è almeno senza testa – ovvero se abbiamo almeno una «query» (un «goal») G_0 da provare

- Si deve dimostrare che da $\mathbf{P} \cup \{G_0\}$ è possibile derivare la clausola vuota $\Rightarrow$ dimostrazione per assurdo mediante applicazione del *Principio di Risoluzione*.

- **Come?**
  - **Problema**: se si tentassero ad ogni passo tutte le risoluzioni possibili e si aggiungessero le clausole inferite all'insieme di partenza si avrebbe una **esplosione combinatoria**
  - Si deve adottare una strategia di soluzione opportuna (una variante più vincolata del Principio di Risoluzione).

Risoluzione ad Input Lineare (SLD)

- Il sistema Prolog dimostra la veridicità o meno di un'interrogazione (un goal) eseguendo *una sequenza di passi di risoluzione*
  - **L'ordine complessivo con cui questi passi vengono eseguiti rende i sistemi di prova di teoremi basati su risoluzione più o meno "efficienti"**

- In Prolog la risoluzione avviene sempre fra l'ultimo goal derivato in ciascun passo e una «clausola di programma»; mai fra due clausole di programma o fra una clausola di programma ed un goal derivato in precedenza
  - Questa particolare forma di risoluzione viene detta **Risoluzione-SLD** (*Selection* function for *Linear* and *Definite* sentences *Resolution*; dove le "frasi lineari" sono essenzialmente le clausole di Horn).

Risoluzione ad input lineare (SLD)

- A partire dal goal G_i

  $G_i \equiv$ `?-` $\mathtt{A}_{i,1}$`,` $\mathtt{A}_{i,2}$`,` $\mathtt{A}_{i,3}$`, …,` $\mathtt{A}_{i,m}$`.`

  e dalla regola

  $\mathtt{A}_r$ `:-` $\mathtt{B}_{r,1}$`,` $\mathtt{B}_{r,2}$`, …,` $\mathtt{B}_{r,k}$`.`

  se esiste un unificatore σ tale che $\sigma[\mathtt{A}_r] = \sigma[\mathtt{A}_{i,1}]$,
  allora si ottiene un nuovo goal G_{i+1}

  $G_{i+1} \equiv$ `?-` $\mathtt{B'}_{r,1}$`,` $\mathtt{B'}_{r,2}$`, …,` $\mathtt{B'}_{r,k}$`,` $\mathtt{A'}_{i,2}$`,` $\mathtt{A'}_{i,3}$`, …,` $\mathtt{A'}_{i,m}$`.`

  Questo è **un** passo di risoluzione eseguito dal sistema Prolog (dove gli $\mathtt{A'}_{\ldots}$ e i $\mathtt{B'}_{\ldots}$ sono i risultati $\sigma[\mathtt{A}_{\ldots}] = \mathtt{A'}_{\ldots}$ e $\sigma[\mathtt{B}_{\ldots}] = \mathtt{B'}_{\ldots}$
- **Nota:** la scelta di unificare il **primo** sottogoal di G_i è **arbitraria** (anche se *comoda*); scegliere $\mathtt{A}_{i,m}$ o $\mathtt{A}_{i,c}$ con $\mathtt{c} \in \mathtt{[1,m]}$ casuale sarebbe ugualmente *lecito*

Risoluzione ad input lineare (SLD)

- Altro caso: a partire dal goal

  $G_i \equiv$ `?-` $\mathtt{A}_{i,1}$`,` $\mathtt{A}_{i,2}$`,` $\mathtt{A}_{i,3}$`, …,` $\mathtt{A}_{i,m}$`.`

  e dalla regola (ovvero dal fatto):

  $\mathtt{A}_r$`.`

  se esiste un unificatore σ tale che $\sigma[\mathtt{A}_r] = \sigma[\mathtt{A}_{i,1}]$,
  allora si ottiene un nuovo goal

  $G_{i+1} \equiv$ `?-` $\mathtt{A'}_{i,2}$`,` $\mathtt{A'}_{i,3}$`, …,` $\mathtt{A'}_{i,m}$`.`

  Ovvero, il goal G_{i+1} ha dimensioni minori rispetto a G_i avendo m - 1 sotto-goals.

Risoluzione ad input lineare (SLD)

- Ripetiamo: nella risoluzione SLD, il passo di risoluzione avviene sempre fra l'ultimo goal e una clausola di programma.

- Il risultato finale può essere
  - **Successo**
    viene generata la clausola vuota, ovvero se per *n* finito G_n è uguale alla clausola vuota $G_n \equiv$ `:-`
  - **Insuccesso finito**
    se per *n* finito, G_n non è uguale a `:-` e non è più possibile derivare un nuovo risolvente da G_n ed una clausola d programma
  - **Insuccesso infinito**
    se è sempre possibile derivare nuovi risolventi tutti diversi dalla clausola vuota

- La sostituzione di risposta è la sequenza di unificatori usati; applicata alle variabili nei termini del goal iniziale dà la risposta finale

Risoluzione ad input lineare (SLD)

- Durante il processo di generazione di goal intermedi si costruiscono delle varianti dei letterali e delle clausole coinvolti mediante la rinominazione di variabili
- Una variante per una clausola *C* é la clausola *C'* ottenuta da *C* rinominando le sue variabili (renaming)

- **Esempio**

```
p(X) :- q(X, g(Z)).
```

è equivalente alla clausola con variabili rinominate

```
p(X1) :- q(X1, g(FooFrobboz)).
```

Strategia di selezione di un sotto-goal

- Possono esserci più clausole di programma utilizzabili per applicare la risoluzione con il goal corrente.
- Si possono adottare diverse strategie di ricerca per queste clausole
  - **In profondità** (**Depth First**)
    si sceglie una clausola e si mantiene fissa questa scelta, finchè non si arriva alla clausola vuota o alla impossibilità di fare nuove risoluzioni; in questo ultimo caso si riconsiderano le scelte fatte precedentemente
  - **In ampiezza** (**Breadth First**)
    si considerano in parallelo tutte le possibili alternative

- *Il **Prolog** adotta una strategia di risoluzione in profondità con **backtracking***
  - Permette di risparmiare memoria
  - Non è completa per le clausole di Horn

Alberi di derivazione SLD

- Dato un programma logico **P**, un goal G_0 e una regola di calcolo ***R***, un **albero SLD** per $\mathbf{P} \cup \{G_0\}$ via ***R*** è definito sulla base del processo di prova visto precedentemente
  - Ciascun **nodo** dell'albero è un goal (possibilmente vuoto)
  - La **radice** dell'albero SLD è il goal G_0
  - dato il nodo

    $:- A_1, \ldots, A_{m-1}, A_m, A_{m+1}, \ldots, A_k$

    se A_m è il sottogoal selezionato dalla regola di calcolo ***R***, allora questo nodo (genitore) ha un nodo figlio per ciascuna clausola del tipo

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

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

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

    e il ramo dal nodo padre al figlio è etichettato dalla sostituzione σ e dalla clausola selezionata C_i (o C_k)
  - Il nodo vuoto (indicato con "`:-`") non ha figli

Alberi di derivazione SLD

- **Ripetiamo**

- La regola ***R*** è variabile
  - Può essere "scelta del sottogoal più a **sinistra**" (se c'è)
    Detta regola **Left-most**
  - Può essere "scelta del sottogoal più a **destra**" (se c'è)
    Detta regola **Right-most**
  - Oppure "scelta di un sottogoal **a caso**"
  - O "scelta del sottogoal **migliore**" (data un'opportuna definizione di "migliore")

- Il Prolog adotta una regola **Left-most**

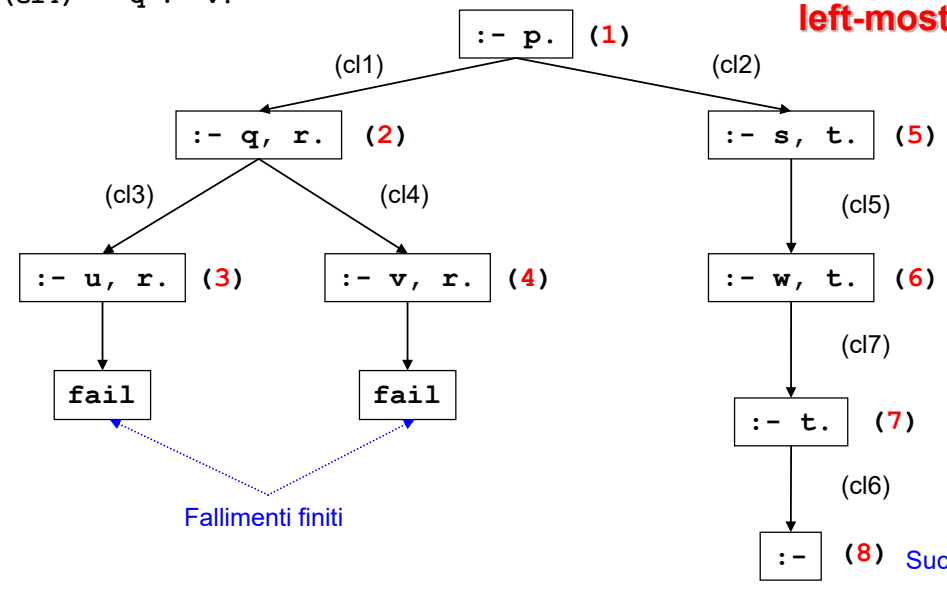
- L'albero SLD (implicito!) generato dal sistema Prolog ordina i figli di un nodo secondo l'ordine dall'alto verso il basso delle regole e dei fatti del programma **P**

Esempio di albero di derivazione

```
(cl1)   p :- q, r.        (cl5)   s:- w.
(cl2)   p :- s, t.        (cl6)   t.
(cl3)   q :- u.           (cl7)   w.
(cl4)   q :- v.
```

goal `:- p.`

Albero di risoluzione left-most



Il Prolog adotta una strategia di attraversamento dell'albero (implicito) SLD in profondità (**Depth First**) con backtracking. Il numero in rosso rappresenta l'ordine di attraversamento usato dal sistema Prolog

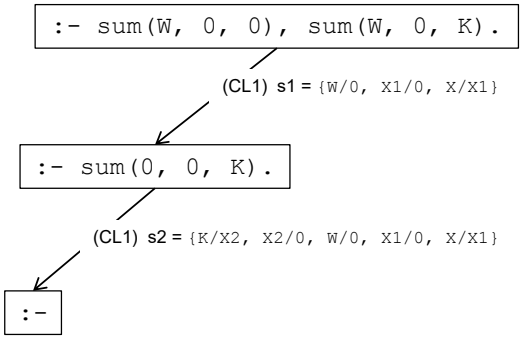
Regola di calcolo

- Ad ogni ramo di un albero SLD corrisponde una derivazione SLD
  - Ogni ramo che termina con il nodo vuoto (“`:-`”) rappresenta una derivazione SLD di successo
- La regola di calcolo influisce sulla struttura dell’albero per quanto riguarda sia l’ampiezza sia la profondità
- Tuttavia non influisce su *correttezza* e *completezza*; quindi, qualunque sia ***R***, il *numero di cammini di successo* (se in numero finito) è lo stesso in tutti gli alberi SLD costruibili per $\mathbf{P} \cup \{G_0\}$
- ***R*** influenza solo il numero di cammini di fallimento (finiti ed infiniti).

Un altro esempio

```
sum(0,X,X).                                   (CL1)
sum(s(X), Y, s(Z)):- sum(X, Y ,Z).            (CL2)
```
G_0 = `:- sum(W, 0, 0), sum(W, 0, K).`

Albero SLD con regola di calcolo “left-most”



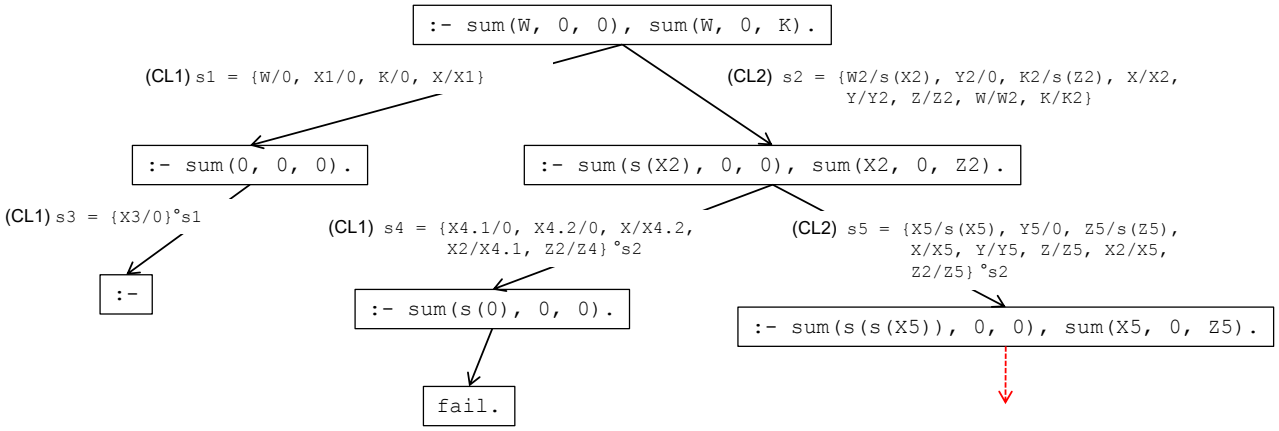
Le variabili `X1` e `X2` sono il risultato dell’operazione di ridenominazione (renaming) della variabile `X` in CL1; appena una clausola viene presa in considerazione le sue variabili sono ridenominate.

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

Stesso esempio, regola diversa

```
sum(0,X,X).                                   (CL1)
sum(s(X), Y, s(Z)):- sum(X, Y ,Z).            (CL2)
```
G_0 = `:- sum(W, 0, 0), sum(W, 0, K).`

Albero SLD con regola di calcolo "right-most"

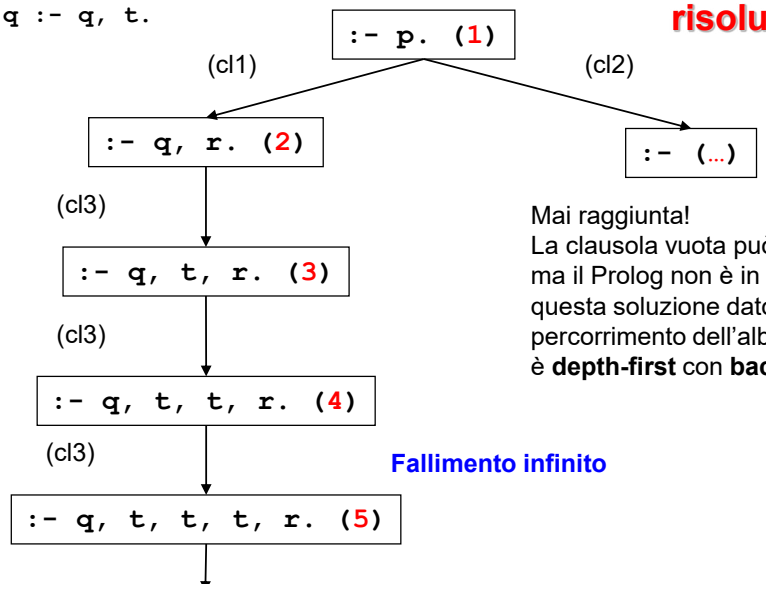


Esempio: fallimento infinito

goal **:- p.**

```
(cl1)   p :- q, r.
(cl2)   p.
(cl3)   q :- q, t.
```

Albero di risoluzione left-most



Mai raggiunta!
La clausola vuota può essere generata ma il Prolog non è in grado di trovare questa soluzione dato che la sua strategia di percorrimento dell'albero (implicito) di soluzioni è **depth-first** con **backtracking**

Fallimento infinito

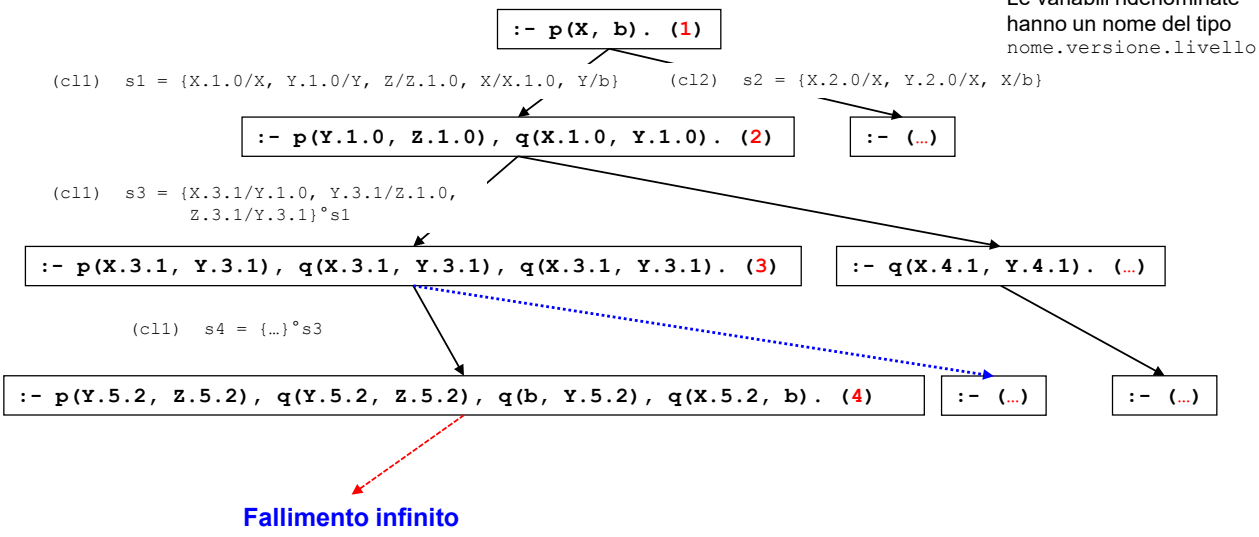
Esempio: fallimento infinito

```
(cl 1)   p(X, Y) :- p(Y, Z), q(X, Y).
(cl 2)   p(X, X).
(cl 3)   q(a, b).
```

goal `:- p(X, b).`

Albero di risoluzione left most

Le variabili ridenominate hanno un nome del tipo `nome.versione.livello`



Fallimento infinito

Sommario

- Relazione tra Calcoli Logici (del Primo Ordine) e Prolog
  - Forma normale
  - Clausole di Horn
- Metodo di *computazione* mediante **prova di teoremi** (*goals*)
  - Regola di inferenza per **Risoluzione**
- Alberi di derivazione (SLD)
  - Un albero SLD rappresenta l'intera computazione potenziale di un sistema di prova di teoremi (date le restrizioni iniziali)
  - Un sistema Prolog attraversa un albero di derivazione **SLD/left-most** in maniera **depth-first** con **backtracking**.

Linguaggi di Programmazione 2020-2021

Prolog e Programmazione Logica IV

Marco Antoniotti
Gabriella Pasi
Rafael Peñaloza

Altri elementi di Prolog

- Predicati meta-logici
- Ispezione di termini
- Predicati di ordine superiore

PREDICATI META-LOGICI

Predicati meta-logici: motivazioni

- Considerate questo predicato:

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

- Il predicato non è *invertibile* (provare per credere) e l'introduzione della seconda clausola

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

non aiuta, dato che il sistema si blocca (con un errore) sulla prima clausola

- Il problema è che dobbiamo decidere quale è l'*input* e quale è l'*output* del nostro calcolo
- Per risolvere questo problema abbiamo bisogno di predicati **meta-logici**

Predicati meta-logici

- Alcuni predicati quindi (ad esempio **`minimum`**, **`celsius_fahrenheit`**) non hanno la tipica *invertibilità* dei risultati di varie queries

- La ragione di questo effetto sta nell'uso che abbiamo fatto di vari predicati aritmetici nel corpo dei predicati (**`>`**, **`<`**, **`=<`**, **`is`**, etc)

- Ovvero, per poter usare i predicati aritmetici che usano direttamente l'hardware abbiamo sacrificato la semantica dei nostri programmi

Predicati meta-logici

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

- I predicati meta-logici più importanti sono
  `var(X)`: vero se **`X`** è una variabile logica
  `nonvar(X)`: l'opposto di **`var(X)`**

- **Esempi**

```
?- var(foo).
false
?- var(X).
tre
?- nonvar(42).
true
```

Predicati meta-logici: esempio

- Il risultato per `celsius_fahrenheit` è il seguente

```
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)` ci permette di decidere quale clausola utilizzare
- Quindi l'uso di questi predicati ci permette di scrivere dei programmi efficienti ed allo stesso tempo semanticamente "corretti"
  - Domanda: cosa succede se C ed F sono entrambe variabili? Ovvero se la query è del tipo `?- celsius_fahrenheit(X, Y).` ?
  - Possiamo usare dei cuts per scrivere un programma ancora più completo?

ISPEZIONE DI TERMINI

Ispezione di termini

- Finora abbiamo visto come si usano dei termini per rappresentare diverse strutture dati in Prolog
- In particolare abbiamo anche intuito che esistono termini **atomici** (corrispondenti alle costanti in un linguaggio logico del primo ordine) e termini **composti** (funzioni e predicati)
- In Prolog abbiamo a disposizione i predicati
  `atomic(X)`: vero se `X` è un numero od una costante
  `compound(X)`: vero se non `atomic(X)`

Ispezione di termini

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

Ispezione di termini

- **Esempi**

```
?- functor(foo(24), foo, 1).
true

?- functor(node(x, _, [], []), F, 4).
F = node

?- functor(Term, bar, 2).
Term = bar(_0,_1)
```

Ispezione di termini

- **Esempi**

```
?- arg(3, node(x, _, [], []), X).
X = []

?- arg(1, father(X, lot), haran).
X = haran
```

Ispezione di termini

- **Esempi**

```
?- father(haran, lot) =.. Ts.
Ts = [father, haran, lot]

?- father(X, lot) =.. [father, haran, lot].
X = haran
```

PROGRAMMAZIONE DI ORDINE SUPERIORE

Programmazione di ordine superiore

- Quando si formula una domanda per il sistema Prolog, ci si aspetta una risposta che è un'istanza (individuale) derivabile dalla knowledge base
- Il meccanismo di backtracking ci permette di estrarre tutte le istanze che possono essere derivate, una alla volta
- ***Cosa succede se vogliamo come risultato l'insieme di tutte le istanze (soluzioni) che soddisfano una certa query?***

Programmazione di ordine superiore

- Questa richiesta non è una richiesta formulabile direttamente al primo ordine (ovvero non è una richiesta formulabile in un linguaggio logico del primo ordine)
- La richiesta è al secondo ordine, dato che richiede un insieme di elementi che soddisfano una certa proprietà
- Il Prolog mette a disposizione dell'utente una serie di **predicati su insiemi** che *estendono il modello computazionale* del linguaggio di base

Predicati su insiemi

- I predicati su insiemi più importanti sono tre

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

Esempi

- Assumiamo di avere a disposizione il solito database di alberi genealogici

```
?- findall(C, father(X, C), Kids).

Kids = [abraham, nachor, haran, isaac, lot, milcah, yiscah]
```

Esempi

- Un esempio con **bagof**

```
?- bagof(C, father(X, C), Kids).

X = terach
KIDS = [abraham, haran, nachor];

X = haran
KIDS = [lot, yiscah, milcah];

X = abraham
KIDS = [isaac];

false
```

Esempi

- **bagof** con variabile esistenziale

```
?- bagof(C, X^father(X, C), Kids).

Kids = [abraham, haran, lot, yiscah, nachor, isaac, milcah];

false
```

Predicati di ordine superiore e meta variabili

- Il Prolog mette a disposizione dell'utente anche altri predicati di ordine superiore
- Buona parte di questi predicati funziona grazie al meccanismo delle meta-variabili, ovvero variabili interpretabili come goals

- Un esempio tipico è il predicato **chiama** che si può pensare essere definito come

```
chiama(G) :- G.
```

  - Il predicato standard SWI-Prolog si chiama **call**.

Predicati di ordine superiore e meta variabili

- Grazie alle 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
```

MANIPOLAZIONE DELLA BASE DATI

Cose da fare con molta attenzione

- Un programma Prolog è costituito da una *base di dati* (o *knowledge base*) che contiene *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 predicati che servono a manipolare direttamente la base di dati sono
  - `listing`
  - `assert`, `asserta`, `assertz`
  - `retract`
  - `abolish`

Manipolazione base dati

- Immaginiamo di partire con una knowledge base vuota. Se si lancia il comando:

```
?- listing.
Yes
```

- La risposta sarà semplicemente `Yes`; il “listing” é ovviamente vuoto, ovvero la base dati corrente è vuota.

Manipolazione base dati

- Consideriamo invece il comand seguente.

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

- Il comando ha successo (assert ha sempre successo). Tuttavia l’importanza di questo comando non è il suo successo, ma il suo effetto collaterale sullo stato del database. Se proseguiamo con la query listing, ottieniamo il seguente effetto:

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

- Ovvero, la base dati non è più vuota: ora contiene il fatto che abbiamo asserito (con `assert`).

Manipolazione base dati

- Eseguiamo quattro nuove assert:

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

?- assert(happy(marcellus)).
true

?- assert(happy(butch)).
true

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

Manipolazione base dati

- A questo punto chiediamo il listing:

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

- Tutti I fatti che abbiamo asserito si trovano nella knowledge base. Notate che `happy(vincent)` si trova due volte nella knowledge base. Ciò non stupisce dato che lo abbiamo asserito due volte.

Manipolazione base dati

- Finora abbiamo solo asserito (aggiunto) dei *fatti* nella base di dati, ma possiamo anche asserire delle regole. Ad esempio, possiamo asserire la regola - molto ottimista - che chiunque è felice è ingenuo (*naïve*). Ovvero, vorremmo asserire

```
naive(X) :- happy(X).
```

- Per far ciò usiamo il comando:

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

- Notate la sintassi di questo comando: la regola che stiamo asserendo è racchiusa tra parentesi. Se ora chiediamo il listato della knowledge base otteniamo:

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

Manipolazione base dati

- Ora che sappiamo come possiamo asserire fatti e regole, ovvero nuove informazioni, nella base di dati, possiamo chiederci come possiamo fare l'operazione inversa. Ovvero come possiamo rimuovere dalla knowledge base fatti e regole che non ci interessano più. Il predicato inverso di `assert` è `retract`. Ad esempio, possiamo dare direttamente il comando seguente:

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

e chiedere il listato del programma; ciò che otteniamo è:

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

Ovvero, il fatto `happy(marcellus)` è stato rimosso.

Manipolazione base dati

- Supponiamo di procedere oltre lanciando in comando seguente:

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

- Il listato che si ottiene è:

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

- Notate che *solo la prima occorrenza* di `happy(vincent)` è stata rimossa.

Manipolazione base dati

- Per rimuovere tutte le nostre asserzioni possiamo usare una variabile.

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

- La richiesta di listare il programma risulta quindi in:

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

Manipolazione base dati

- Per avere più controllo su dove vengono aggiunti fatti e regole possiamo usare le due varianti di `assert`, ovvero:
  - **`assertz`**
    Inserisce l'asserzione alla fine della knowledge base.
  - **`asserta`**
    inserisce l'asserzione all'inizio della knowledge base.
- Ad esempio, se partiamo con una base di dati vuota, e diamo il seguente comando:

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

- Il risultato del comando listing sarà:

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

Manipolazione base dati

- La manipolazione del database Prolog è una cosa molto utile
- Ad esempio, può essere usata per memorizzare i risultati intermedi di varie computazioni, in modo da non dover rifare delle queries dispendiose in futuro: semplicemente si ricerca direttamente il fatto appena asserito
- Questa tecnica si chiama *memoization* o *caching* (in Inglese)

Manipolazione base dati

- Ecco un esempio. Creiamo una tavola di addizioni manipolando la knowledge base. Consideriamo il programma seguente:

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

In questo esempio `member/2` è il predicato standard che controlla l'appartenenza di un elemento in una lista

Manipolazione base dati

- Consideriamo il seguente esempio

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

- La risposta è *false*; ma non è la risposta che ci interessa, bensì l'effetto (collaterale) che l'interrogazione ha sulla knowledge base
- Se ora chiediamo un listato della base dati otteremo

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

Manipolazione base dati

- Domanda: come possiamo rimuovere tutti questi fatti quando non li vogliamo più nel data base? È vero che potremmo semplicemente dare il comando:

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

- Ma in questo casi il Prolog ci chiederebbe se vogliamo rimuovere tutti i fatti uno per uno!
- In realtà c'è un modo più semplice e diretto: usiamo il comando:

```
?- retract(sum(_, _, _)), fail.
No
```

- Ancora una volta, lo scopo del `fail` è di forzare il backtracking. Il Prolog rimuove il primo fatto con funtore sum dalla base di dati e poi fallisce. Quindi fa bactrack e rimuove il fatto successivo e così via. Alla fine, dopo aver rimosso tutti i fatti con funtore `sum`, la query fallirà completamente ed il Prolog risponderà (correttamente) con un No. Ma anche in questo caso a noi interessa unicamente l'effetto - *collaterale* - sulla knowledge base.

Sommario

- A conclusione, si ripetono gli argomenti trattati in questa serie di slides.
  - Predicati meta-logici
  - Ispezione di termini
  - Predicati di ordine superiore

Linguaggi di Programmazione 2021-2022

Prolog e Programmazione Logica V

Marco Antoniotti
Gabriella Pasi
Rafael Peñaloza

I/O IN PROLOG

Input e Output in Prolog

- I predicati primitivi principali per la gestione dell'I/O sono essenzialmente due, **`read`** e **`write`**, a cui si aggiungono i vari predicati per la gestione dei files e degli streams: **`open`**, **`close`**, **`seek`**, etc.

- **`read`** e **`write`** sono peculiari: leggono e scrivono ***termini*** Prolog
  - **`write`** è equivalente all'invocazione di un metodo `toString` Java su un oggetto di classe "termine"
  - **`read`** di fatto invoca il parser Prolog

Input e Output in Prolog

- **Esempi**

```
?- 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).
```

Input e Output in Prolog

- **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)
```

- **open** e **close** servono per leggere e scrivere files; la versione più semplice di **open** ha con tre argomenti: un atomo che rappresenta il nome del file, una "modalità" con cui si apre il file ed un terzo argomento a cui si associa l'identificatore del file.
- Vi sono naturalmente molti altri predicati per I/O in Prolog
  - Qui sopra avete visto il predicato **put/2**, che emette un carattere sullo stream ed il predicato **nl/2** che mette un 'newline' sullo stream
  - Inoltre avete visto che il Prolog usa la notazione **0'c** per rappresentare i caratteri come termini

INTERPRETERS IN PROLOG

“Interpreti” in Prolog

- Il Prolog si presta benissimo alla costruzione di **interpreti** per la manipolazione di linguaggi specializzati (**Domain Specific Languages** – DSLs)

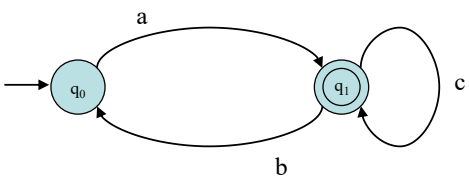
- Esempi tipici sono
  - Intepreti per **Automi** (a Stati Finiti, a Pila, Macchine di Turing)
  - Sistemi per la **deduzione automatica**
  - Sistemi per la manipolazione del **Linguaggio Naturale** (**Natural Language Processing** – NLP)

Interpreti in Prolog: Automi

- Come costruiamo un “interprete” per riconoscere nondeterministicamente dei linguaggi regolari?

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

- Consideriamo l’automa



Interpreti in Prolog: Automi

- L'automa viene codificato con

```
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).
```

- Esempi

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

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

Interpreti in Prolog: Automi a Pila

- Come costruiamo un "interprete" per riconoscere nondeterministicamente dei linguaggi liberi da contesto (context-free languages)?

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

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

- Consideriamo il linguaggio

$$\boldsymbol{L} = \{wrw^R \mid w \in \{a, b, c\}^n \text{ AND } n \geq 0\}$$

il linguaggio non è regolare (applicate il "Pumping Lemma")

- Scriviamo le regole necessarie per la codifica della funzione di transizione

Interpreti in Prolog: Automi a Pila

- L'automa viene codificato con

```
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, []).
```

- Esempi

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

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

Meta-interpreti

- Il predicato **`call/1`** che abbiamo visto precedentemente è il più semplice **meta-interprete** Prolog
- Possiamo scrivere degli interpreti più complicati e/o specializzati se accettiamo di rappresentare i programmi con una sintassi leggermente diversa
- Considerate la base di dati seguente

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

Meta-interpreti: variazioni 1

- Considerate ora il seguente programma

```
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).
```

- Il programma `solve` è un meta-interprete per i predicati `rule` che compongono il nostro sistema (o programma)

Meta-interpreti: variazioni 2

- Ragioniamo con **incertezza**

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

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

Meta-interpreti: variazioni 3

- Ragioniamo con incertezza, ma con una *soglia* (indicata dalla variabile `T`, per "threshold")

```
solve_cf(true, 1, T) :- !.

solve_cf((A, B), C, T) :-
     !,
     solve_cf(A, CA, T),
     solve_cf(B, CB, T),
     minimum(CA, CB, C).

solve_cf(A, 1, T) :- builtin(A), !, call(A).

solve_cf(A, C, T) :-
     rule_cf(A, B, CR),
     CR > T,
     T1 is T/CR,
     solve_cf(B, CB, T1),
     C is CR * CB.
```

Conclusioni

- Il Prolog è un linguaggio di altissimo livello che ci permette di esprimere problemi, conoscenze e soluzioni in un modo conciso e naturale
- Lo stile di programmazione che meglio si adatta al linguaggio è completamente dichiarativo
- L'uso del Prolog è particolarmente efficace nella programmazione di sistemi di deduzione di vario tipo
- 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)

Conclusioni

- Noi abbiamo visto alcuni elementi del **Prolog**, anche sofisticati

- Non abbiamo visto
  - Moduli
  - Uso di Prolog come strumento di *analisi sintattica* (“*parsing*”)
  - Uso della nozione di operatore per modificare il linguaggio
  - Esplorazione più approfondita dell’equivalenza tra programmi e dati