

# Modulo 2 - Chesani

<b>Introduzione alla Rappresentazione della Conoscenza.....</b>	<b>5</b>
<b>Logica Proposizionale (PL).....</b>	<b>5</b>
Inferenza Logica.....	6
<b>Logica del Primo Ordine (FOL).....</b>	<b>6</b>
Estensione della PL.....	6
Inferenza nella FOL.....	7
<b>Inferenza e Deduzione.....</b>	<b>7</b>
<b>Risoluzione per Refutazione.....</b>	<b>8</b>
<b>Proprietà dei Metodi di Ragionamento.....</b>	<b>9</b>
in sintesi:.....	9
<b>Prolog.....</b>	<b>10</b>
Unificazione e Inferenza.....	11
Strutture Dati e Liste.....	12
Ricorsione in Prolog.....	12
Controllo del Flusso: Il Predicato Cut (!).....	13
Negazione in Prolog.....	13
Meta-Predicati.....	14
Meta-Interpreters.....	15
Vanilla Meta-Interpreter.....	16
Estensioni dei Meta-Interpreters.....	17
in sintesi:.....	18
<b>Introduzione alle Ontologie.....</b>	<b>19</b>
Caratteristiche fondamentali delle ontologie:.....	19
Ontologie e Semantic Web.....	20
Upper Ontologies.....	21
in sintesi:.....	22
<b>Frames and Semantic Networks.....</b>	<b>23</b>
Frames.....	23

Caratteristiche.....	23
Semantic Networks.....	24
Caratteristiche.....	24
Vantaggi e Limiti.....	25
in sintesi:.....	26
<b>Description Logics.....</b>	<b>27</b>
Struttura della Description Logic.....	27
Operatori.....	27
Varianti della Description Logic.....	28
ALC Description Logics.....	28
Operatori Supportati.....	28
SHOIN(D).....	29
SROIQ(D).....	30
Knowledge Bases in DL.....	30
Ragionamento nella Description Logic.....	31
Implementazioni e Reasoners.....	31
in sintesi:.....	32
<b>Protégé.....</b>	<b>33</b>
Componenti principali.....	33
Proprietà e Restrizioni.....	34
Reasoning in Protégé.....	34
Open World Assumption (OWA) vs. Closed World Assumption (CWA).....	35
<b>Semantic web and Knowledge Graphs.....</b>	<b>36</b>
Strumenti e Tecnologie.....	36
Knowledge Graphs.....	37
Caratteristiche dei Knowledge Graphs.....	37
Applicazioni del Semantic Web e dei Knowledge Graphs.....	37
in sintesi:.....	38
<b>Ragionamento Temporale.....</b>	<b>39</b>
Logiche Temporal.....	39

Linear Temporal Logic (LTL).....	39
Allen's Interval Algebra.....	40
Event Calculus e Monitoraggio degli Eventi.....	40
Applicazioni del Ragionamento Temporale.....	41
in sintesi:.....	42
<b>Modal logics and Linear Temporal Logic (LTL).....</b>	<b>43</b>
Introduzione alle Logiche Modali.....	43
Propositional Attitudes e Modal Logic.....	43
Linear Temporal Logic (LTL).....	44
Semantica di LTL.....	45
Relazione tra Logiche Modali e LTL.....	45
Applicazioni di LTL e Logiche Modali.....	45
in sintesi:.....	46
<b>Prolog LPAD.....</b>	<b>47</b>
Introduzione a Prolog LPAD.....	47
Esempio: Il Problema di Monty Hall in LPAD.....	50
in sintesi:.....	51
<b>Forward Reasoning.....</b>	<b>52</b>
Introduzione ai Sistemi Basati su Regole.....	52
Il Meccanismo di Forward Reasoning.....	52
L'Algoritmo RETE.....	53
in sintesi:.....	54
<b>Business process management (BPM).....</b>	<b>55</b>
Ciclo di Vita dei Processi Aziendali.....	55
Modellazione dei Processi Aziendali.....	55
Tipologie di processi:.....	56
Pattern di Workflow e Controllo di Flusso.....	56
Allocazione delle Risorse nei Processi.....	57
Business Process Mining.....	57
in sintesi:.....	58

<b>Procedural languages for process modeling.....</b>	<b>59</b>
Introduzione ai Linguaggi Procedurali per la Modellazione dei Processi.....	59
Petri Nets.....	59
Workflow Nets (WF-nets).....	60
YAWL (Yet Another Workflow Language).....	60
BPMN (Business Process Model and Notation).....	61
in sintesi:.....	62
<b>Formal properties for BPM languages - declarative languages.....</b>	<b>63</b>
Differenze tra Modellazione Procedurale e Dichiarativa.....	63
Proprietà Formali nei Linguaggi Dichiarativi.....	64
Proprietà Strutturali e di Controllo del Flusso.....	64
Conformance Checking.....	64
DECLARE.....	65
in sintesi:.....	65
<b>Process Mining.....</b>	<b>66</b>
Fasi del Process Mining.....	66
Scoperta (Discovery).....	66
Verifica di Conformità (Conformance Checking).....	67
Miglioramento (Enhancement).....	67
Algoritmi di Process Mining.....	67
Sfide e Limiti del Process Mining.....	68
in sintesi:.....	69

# Introduzione alla Rappresentazione della Conoscenza

La rappresentazione della conoscenza è un aspetto fondamentale dell'intelligenza artificiale, poiché consente ai sistemi di elaborare informazioni in modo strutturato, inferire nuove conoscenze e risolvere problemi. L'obiettivo è formalizzare le informazioni in modo che possano essere trattate da algoritmi logici e motori di inferenza.

L'uso di un linguaggio formale consente di eliminare ambiguità e garantire una rappresentazione chiara e coerente della conoscenza. La sfida principale è bilanciare l'espressività della rappresentazione con la sua efficienza computazionale.

## Logica Proposizionale (PL)

La logica proposizionale è un sistema formale che utilizza proposizioni atomiche per rappresentare fatti del mondo e operatori logici per combinarli.

→ **Proposizioni:** entità che possono essere vere o false.

→ **Operatori logici:**

- **Negazione** ( $\neg P$ ): afferma il contrario della proposizione.
- **Congiunzione** ( $P \wedge Q$ ): entrambe le proposizioni devono essere vere.
- **Disgiunzione** ( $P \vee Q$ ): almeno una delle proposizioni deve essere vera.
- **Implicazione** ( $P \rightarrow Q$ ): se  $P$  è vero, allora  $Q$  deve essere vero.
- **Doppia implicazione** ( $P \leftrightarrow Q$ ):  $P$  e  $Q$  devono avere lo stesso valore di verità.

La semantica della logica proposizionale si basa sulle *interpretazioni*, ovvero assegnazioni di valori di verità alle proposizioni.

- **Modello:** un'interpretazione in cui una formula è vera.
- **Tautologia:** formula sempre vera in ogni interpretazione.

- **Contraddizione:** formula sempre falsa.
- **Soddisfacibilità:** esiste almeno un modello in cui la formula è vera.

## Inferenza Logica

L'inferenza logica permette di *dedurre nuove informazioni* da un insieme di premesse.

- **Modus Ponens:** se  $(P \rightarrow Q)$  e  $P$  sono veri, allora anche  $Q$  è vero.
- **Modus Tollens:** se  $(P \rightarrow Q)$  e  $\neg Q$  sono veri, allora anche  $\neg P$  è vero.
- **Doppia Negazione:**  $\neg(\neg P)$  equivale a  $P$ .
- **Sillogismo ipotetico:** se  $(P \rightarrow Q)$  e  $(Q \rightarrow R)$  sono veri, allora  $(P \rightarrow R)$  è vero.
- **Leggi di De Morgan:** permettono di trasformare negazioni di congiunzioni e disgiunzioni.

## Logica del Primo Ordine (FOL)

### Estensione della PL

La logica del primo ordine *estende la logica proposizionale* introducendo variabili, predicati e quantificatori.

- **Predicati:** esprimono proprietà o relazioni tra oggetti.
- **Funzioni:** restituiscono un oggetto in base agli input forniti.
- **Quantificatori:**
  1. **Universale** ( $\forall x$ ): vale per *tutti gli elementi* del dominio.
  2. **Esistenziale** ( $\exists x$ ): esiste almeno un elemento per cui *la formula è vera*.

### Esempi

"Tutti gli uomini sono mortali":  $\forall x (Uomo(x) \rightarrow Mortale(x))$ .

"Esiste almeno un uomo":  $\exists x (Uomo(x))$ .

"Ogni numero naturale ha un successore":  $\forall x(\text{Naturale}(x) \rightarrow \exists y \text{Successore}(x, y))$ .

## Inferenza nella FOL

L'inferenza in FOL si basa su *regole logiche più potenti* rispetto alla logica proposizionale.

- a. **Unificazione:** tecnica per rendere *due termini compatibili*.
- b. **Clausole di Horn:** fondamentali nei sistemi logici come Prolog.
- c. **Universo di Herbrand:** insieme di tutti i termini costruibili in un dominio.

## Inferenza e Deduzione

L'inferenza è il processo di derivazione di nuove affermazioni a partire da un insieme di premesse, con l'obiettivo di arricchire il sistema di conoscenza automatizzata. Si distinguono tre principali metodi di inferenza:

- **Deduzione:** derivazione rigorosa di conclusioni basate su regole logiche applicate a dati già disponibili.
- **Abduzione:** partendo da un'osservazione, si cerca la causa più plausibile per spiegare il fenomeno.
- **Induzione:** generalizzazione di una regola a partire da casi specifici. (comune nei sistemi di apprendimento automatico.)
- **Chaining forward e backward:** strategie per l'inferenza automatica nei sistemi esperti.

## Risoluzione per Refutazione

La risoluzione per refutazione è un metodo utilizzato per dimostrare che un'affermazione è una conseguenza logica di un insieme di premesse. Si basa sul principio del *ragionamento per assurdo*, ovvero si assume che l'affermazione da dimostrare sia falsa e si cerca di ottenere una contraddizione.

### Passaggi del metodo:

1. **Conversione in Forma Normale Congiuntiva (CNF):** Ogni formula logica viene trasformata in un insieme di clausole, ovvero una congiunzione di disgiunzioni.
2. **Negazione della Congettura:** Si aggiunge la negazione della formula da dimostrare all'insieme delle clausole.
3. **Applicazione della Regola di Risoluzione:** Si selezionano due clausole contenenti letterali complementari e si sostituiscono con una nuova clausola che contiene tutti i letterali rimanenti, rimuovendo i letterali opposti.
4. **Ripetizione fino alla Contraddizione:** Il processo continua fino a generare la clausola vuota ( $\perp$ ), che rappresenta una contraddizione e dimostra che l'asserzione originale è vera.

**Esempio:** per dimostrare che "Socrate è mortale" a partire da "Tutti gli uomini sono mortali" e "Socrate è un uomo", si traduce il problema in logica del primo ordine e si applica la risoluzione per refutazione.



# Proprietà dei Metodi di Ragionamento

Un metodo di inferenza è valutato secondo tre proprietà fondamentali:

- a. **Correttezza (Soundness)**: se il metodo deriva una formula, essa è logicamente vera.
- b. **Completezza (Completeness)**: se una formula è vera, il metodo è in grado di derivarla.
- c. **Decidibilità (Decidability)**: esistenza di un algoritmo che garantisca una risposta in tempo finito.

## in sintesi:

Lo studio della logica proposizionale e del primo ordine costituisce la base per la rappresentazione della conoscenza nell'intelligenza artificiale. L'inferenza logica e la risoluzione per refutazione sono strumenti fondamentali per l'automazione del ragionamento. Infine, la correttezza, la completezza e la decidibilità sono aspetti cruciali per valutare l'efficacia di un sistema logico. Questo modulo fornisce una solida base per lo studio di tecniche avanzate di intelligenza artificiale e rappresentazione della conoscenza.

# Prolog

Prolog è un linguaggio di programmazione logica basato su regole e fatti.

Un programma Prolog è composto da **fatti**, **regole** e **query**:

- a. **Fatti**: dichiarano informazioni su oggetti e relazioni.
- b. **Regole**: esprimono relazioni tra fatti e come derivare nuove informazioni.
- c. **Query**: domande poste al sistema per ottenere inferenze basate sui fatti e sulle regole.

Un programma in Prolog è costituito da un insieme di **clausole definite** con la seguente struttura:

- **Fatto**: una dichiarazione di verità.  
`father(leo, lorenzo).`

Qui si afferma che "Leo è il father di Lorenzo".

- **Regola**: una condizione logica che stabilisce una relazione tra più fatti.  
`nonno(X, Y) :- father(X, Z), father(Z, Y).`

Questa regola afferma che "X è nonno di Y se X è father di Z e Z è father di Y".

- **Query**: una domanda posta al sistema.  
`?-nonno(franco, lorenzo).`

Questa query chiede al sistema se Franco è nonno di Lorenzo.

## Esecuzione di un Programma Prolog

L'esecuzione in Prolog avviene tramite il **meccanismo di inferenza** basato sulla **risoluzione logica**.

Quando si pone una query, il sistema cerca di dimostrare la sua validità esplorando le regole e i fatti noti. Se trova un percorso logico valido, restituisce una risposta positiva; altrimenti, restituisce **false**.

Esempio:

?-nonno(franco, lorenzo).

Il sistema cercherà nella *base di conoscenza (KB)* e risponderà **true** se il fatto è definito, altrimenti **false**.

## Unificazione e Inferenza

L'unificazione è il processo con cui Prolog confronta due termini e determina se possono essere resi uguali sostituendo variabili con valori appropriati.

Esempio di unificazione:

father(X, marco) = father(mario, marco).

X = mario.

Qui, il sistema deduce che **X deve essere mario** per rendere le due espressioni equivalenti.

L'unificazione in Prolog è il meccanismo fondamentale per la risoluzione dei predicati, ma presenta alcune particolarità. Un aspetto importante è il cosiddetto **Occur Check**, ovvero il controllo che impedisce a una variabile di unificarsi con un termine che la contiene, evitando ricorsioni infinite

L'inferenza in Prolog avviene attraverso la **risoluzione SLD (Selective Linear Definite clause resolution)**, che applica le regole e i fatti per derivare conclusioni logiche.

## Strutture Dati e Liste

Prolog supporta diverse strutture dati, tra cui **liste**, che sono definite come sequenze di elementi racchiuse tra parentesi quadre.

Esempio:

```
[1, 2, 3, 4].
```

Per accedere agli elementi di una lista, si usa la notazione **[Testa|Coda]**:

```
?- [X|Y] = [1,2,3,4].
```

```
X = 1,
```

```
Y = [2,3,4].
```

Qui, **X** viene unificato con il primo elemento della lista e **Y** con il resto degli elementi.

## Ricorsione in Prolog

Dato che Prolog non ha cicli tradizionali, la ricorsione è il principale metodo per iterare su strutture dati.

Esempio di somma degli elementi di una lista:

```
somma([], 0).
```

```
somma([H|T], S) :- somma(T, ST), S is H + ST.
```

Qui, la funzione somma:

- Ritorna **0** se la lista è vuota.
- Divide la lista in **testa (H)** e **coda (T)** e calcola la somma ricorsivamente.

## Controllo del Flusso: Il Predicato Cut (!)

Il predicato **cut** (!) viene usato per evitare backtracking non necessario, rendendo *più efficiente l'esecuzione* del programma. ma può avere due scopi distinti:

- green cut, viene utilizzato per migliorare l'efficienza senza cambiare il comportamento logico del programma
- red cut, modifica la logica del programma impedendo a prolog di trovare altre soluzioni, anche se sarebbero corrette.



### Regola pratica:

- Usa **Cut Verde** per ottimizzare il codice senza modificarne il significato.
- Evita il **Cut Rosso**, a meno che tu non voglia forzare un comportamento specifico.

Esempio:

```
massimo(X, Y, X) :- X >= Y, !.  
massimo(X, Y, Y).
```

→ Qui, il **cut** (!) assicura che una volta verificata la prima condizione, *il programma non consideri altre alternative*, migliorando l'efficienza.

## Negazione in Prolog

Prolog utilizza la **negazione come fallimento** (*negation as failure*), il che significa che  $\text{\texttt{\textbackslash+ P}}$  è vero solo se **P** non può essere dimostrato vero. Questo è diverso dalla **negazione classica** della logica formale, in cui  $\text{\texttt{-P}}$  è vero se e solo se **P** è dimostrabilmente falso.

Esempio:

```
non_povero(X) :- \+ povero(X).
```

→ Se Prolog non riesce a provare che **povero(X)** è vero, allora assume che **non\_povero(X)** sia vero.

💡 **Problema con variabili non istanziate:** Se la negazione viene applicata a una variabile non ancora istanziata, può portare a risultati inattesi

## Meta-Predicati e Meta-Interpreters in Prolog

Prolog permette di *manipolare predicati* come dati attraverso i **meta-predicati**, consentendo di definire operazioni di ordine superiore e gestire dinamicamente il comportamento del programma. Inoltre, i **meta-interpreters** permettono di creare interpreti personalizzati per l'esecuzione di regole logiche.

### Meta-Predicati

#### Predicati Predefiniti

1. **member/2**: Verifica se un elemento appartiene a una lista.
2. **length/2**: Calcola la lunghezza di una lista.
3. **append/3**: Concatena due liste.
4. **reverse/2**: Calcola la lista invertita.
5. **findall/3**: Raccoglie tutte le soluzioni di un predicato in una lista.
6. **setof/3**: Raccoglie tutte le soluzioni di un predicato senza duplicati.
7. **bagof/3**: Raccoglie tutte le soluzioni di un predicato permettendo duplicati.
8. **is/2**: Esegue calcoli aritmetici.
9. **number/1**: Verifica se un termine è un numero.
10. **ground/1**: Verifica se un termine non contiene variabili libere.
11. **call/1**: Esegue dinamicamente un predicato.
12. **clause/2**: Accede alle clausole definite nel programma.
13. **var/1**: Verifica se un termine è una variabile.
14. **nonvar/1**: Verifica se un termine non è una variabile.
15. **fail/0**: Fallisce sempre, forzando il backtracking.

16. **cut/0 (!)**: Impedisce il backtracking oltre un certo punto.
  17. **not/1**: Implementa la negazione come fallimento.
- 

## Predicati Esemplificativi

1. **parent/2**: Rappresenta relazioni genitoriali.
2. **people/1**: Raccoglie tutte le persone menzionate in **parent/2** senza duplicati.
3. **filter/2**: Modifica una lista in base a regole specifiche.
4. **deleteFirstOccurrence/3**: Rimuove la prima occorrenza di un elemento in una lista.
5. **deleteAllOccurrences/3**: Rimuove tutte le occorrenze di un elemento in una lista.
6. **rightmost\_interpret/1**: Meta-interprete che seleziona il sotto-goal più a destra.
7. **solve/1** e **solve/2**: Meta-interpreti che risolvono query Prolog.
8. **iterate/0**: Itera su elementi che soddisfano un predicato.
9. **verify/1**: Verifica una condizione su una lista.
10. **if\_then\_else/3**: Simula un costrutto condizionale.

## Meta-Interpreters

In Prolog, non esiste una distinzione tra programma e dati, il che permette di trattare i predicati come termini manipolabili da altri programmi. Un meta-interpreter è un programma che esegue altri programmi, permettendo di modificare il comportamento dell'interprete Prolog standard.

Caratteristiche principali:

- Astrazione del comportamento dell'interprete.
- Supporto per l'estensione e la modifica dell'inferenza logica.
- Utilizzo per debugging e controllo avanzato dell'esecuzione.

I meta-interpreters possono essere utilizzati per costruire sistemi avanzati, come motori di inferenza personalizzati o strumenti di debugging sofisticati. Prolog permette di manipolare predicati come dati attraverso i **meta-predicati**. Alcuni esempi:

- **call(P)**: chiama dinamicamente il predicato **P**.
- **fail**: forza il fallimento di un predicato, utile per generare tutte le soluzioni.
- **setof/3**, **bagof/3**, **findall/3**: raccolgono tutte le soluzioni di un predicato in una lista.

Esempio di **findall**:

```
?- findall(X, father(mario, X), Lista).  
Lista = [marco, anna, giovanni].
```

Qui, **findall** raccoglie tutti i figli di Mario in una lista.

## Vanilla Meta-Interpreter

Il Vanilla Meta-Interpreter è un semplice meta-interprete per Pure Prolog, che implementa il comportamento standard dell'interprete Prolog.

Definizione:

```
solve(true) :- !.  
solve((A, B)) :- !, solve(A), solve(B).  
solve(A) :- clause(A, B), solve(B).
```

Proprietà:

- Replica il comportamento dell'interprete standard.
- Ricerca le clausole nel database e prova a soddisfarle ricorsivamente.
- Non gestisce predicati predefiniti (come `write/1` o `read/1`), che devono essere aggiunti separatamente.



## Estensioni dei Meta-Interpreters

Una volta definito il **Vanilla Meta-Interpreter**, è possibile modificarlo per ottenere comportamenti personalizzati.

- Meta-Interpreter con Tracciamento: Aggiungendo istruzioni di stampa, possiamo monitorare il processo di risoluzione.

```
solve(true) :- !.  
solve((A,B)) :- !, solve(A), solve(B).  
solve(A) :- write('Solving: '), write(A), nl,  
            clause(A,B), solve(B), write('Solved: '), write(A), nl.
```

Risultato dell'esecuzione:

```
?- solve(parent(john, mary)).  
Solving: parent(john, mary)  
Solved: parent(john, mary)
```

- Meta-Interpreter con Contatore di Passaggi: Modifica che conta il numero di passi di risoluzione.

```
solve(true, 0) :- !.  
solve((A,B), S) :- !, solve(A, SA), solve(B, SB),  
                    S is SA + SB.  
solve(A, S) :- clause(A, B), solve(B, SB), S is 1 + SB.
```

Esempio:

```
?- solve(a, Steps).  
Steps = 4.
```

- **Meta-Interpreters per Ragionamento Probabilistico:** Possiamo estendere il meta-interprete per **gestire incertezze** attribuendo punteggi di probabilità ai fatti
- **Meta-Interpreters per Sistemi Esperti:** Un meta-interprete può essere utilizzato per implementare **sistemi esperti**, ovvero sistemi in grado di rispondere a domande sulla base di una knowledge base esterna.

### **in sintesi**

I concetti chiave da comprendere includono:

- Il modello basato su fatti, regole e query.
- L'unificazione e la risoluzione SLD.
- L'uso della ricorsione per iterare sulle strutture dati.
- L'importanza del controllo del flusso con il predicato **cut (!)**.
- L'utilizzo della negazione come fallimento e dei meta-predicati per manipolare dinamicamente le regole logiche.

Questi strumenti permettono di creare sistemi basati sulla conoscenza capaci di rispondere a domande complesse e automatizzare il ragionamento logico.

# Introduzione alle Ontologie

Le **ontologie**, le **tassonomie** e le **basi di conoscenza** sono strumenti complementari per rappresentare e organizzare la conoscenza, ma differiscono per struttura e utilizzo:

- **Ontology** → Le ontologie sono una *rappresentazione formale della conoscenza* in un dominio specifico. Una ontologia definisce le entità (classi), le relazioni tra esse e le proprietà che caratterizzano tali entità.
  - *Esempio*: Un'ontologia può definire il concetto di "**Vehicle**" con proprietà come "**hasWheels**" e "**hasEngine**".
- **Taxonomy** → È una gerarchia di concetti organizzati per relazioni di specializzazione (is-a), ma senza una semantica rigorosa. Non supporta inferenza.
  - *Esempio*: Una tassonomia può rappresentare "**Car**" e "**Truck**" come sottocategorie di "**Vehicle**".
- **Knowledge Base (KB)** → È un insieme di fatti e regole inferenziali che possono essere basate su un'ontologia. Può contenere dati espliciti e derivati.
  - *Esempio*: Una KB può contenere istanze di "**Car**" come "**Tesla Model S**" e regole come "**If a vehicle has four wheels, it is a car**".

## Caratteristiche fondamentali delle ontologie:

Le ontologie possono essere classificate in due categorie principali in base al loro livello di espressività:

1. **Lightweight Ontologies** → Sono semplici strutture gerarchiche che definiscono classi e relazioni base senza vincoli complessi.
  - *Esempio*: **RDFS (RDF Schema)** permette di definire classi (**rdfs:Class**) e proprietà (**rdfs:subClassOf**), ma senza supporto per restrizioni logiche avanzate.

2. **Rich Ontologies** → Includono vincoli formali, inferenza automatica e supporto per ragionamento avanzato.

→ *Esempio:* **OWL (Web Ontology Language)** supporta concetti come `owl:someValuesFrom` per restrizioni esistenziali e `owl:inverseOf` per definire proprietà inverse.

Le caratteristiche fondamentali sono:

1. **Formalità:** devono essere espresse in un linguaggio con semantica ben definita.
2. **Esplicitazione:** tutte le informazioni devono essere chiaramente rappresentate.
3. **Descrizione:** specificano categorie, proprietà e relazioni.
4. **Dominio specifico:** legate a un particolare ambito di interesse.

Le ontologie sono utilizzate per:

- **Integrazione dei dati:** permettono di unificare informazioni provenienti da fonti diverse.
- **Inferenza automatica:** permettono a sistemi intelligenti di dedurre nuove conoscenze.
- **Strutturazione della conoscenza:** facilitano la condivisione e la riutilizzabilità delle informazioni.

## Ontologie e Semantic Web

Il **Semantic Web** è un'estensione del World Wide Web che mira a rendere i dati sul web interpretabili dalle macchine attraverso l'uso di standard e tecnologie definite dal W3C.

Principali tecnologie del Semantic Web:

- **RDF (Resource Description Framework):** utilizza triple `<soggetto, predicato, oggetto>` per descrivere risorse.

- **RDFS (RDF Schema)**: estensione di RDF che introduce classi e proprietà per strutturare la conoscenza.
- **OWL (Ontology Web Language)**: linguaggio per definire ontologie formali, con diverse versioni di espressività (OWL Lite, OWL DL, OWL Full).

Le **upper ontologies** sono ontologie di alto livello progettate per essere applicabili a molteplici domini e fornire una struttura comune per integrare ontologie più specifiche.

## Upper Ontologies

Le **upper ontologies** rappresentano *il livello più astratto delle ontologie* e forniscono un framework generale per la rappresentazione della conoscenza. Servono come base *per costruire ontologie più specifiche*.

Caratteristiche delle upper ontologies:

- **Applicabilità generale**: Devono essere utilizzabili in diversi ambiti senza dipendere da un dominio specifico.
- **Coerenza**: Concetti generali come **tempo, spazio, oggetti, eventi** devono essere compatibili tra loro.
- **Generalizzazione**: Consentono di collegare ontologie specifiche a una struttura più ampia.

Esistono diversi approcci alla creazione delle upper ontologies:

1. **Ontologie sviluppate da filosofi, logici e ricercatori di intelligenza artificiale.**
  - **Esempio: Cyc**: Progetto nato per rappresentare la conoscenza comune e il ragionamento basato sul senso comune.
  - Cyc è una delle ontologie più complete per il ragionamento automatico e ha versioni sia commerciali che open-source.
2. **Ontologie estratte automaticamente da dataset strutturati.**

- **Esempio: DBpedia:** Ontologia derivata da Wikipedia, utilizzata per creare un dataset RDF interconnesso.
- DBpedia permette di eseguire interrogazioni SPARQL su informazioni enciclopediche.

### 3. Ontologie estratte automaticamente da documenti di testo.

- **Esempio: TextRunner:** Sistema basato su NLP per estrarre conoscenza da testi non strutturati presenti nel web.
- Utilizza tecniche di Open Information Extraction per generare ontologie.

### 4. Ontologie costruite con approcci basati su crowdsourcing.

- **Esempio: OpenMind Common Sense:** Progetto che ha raccolto conoscenze da utenti per creare un'ontologia di senso comune.
- Ha permesso di creare una base di conoscenza ampia e diversificata con il contributo della comunità.

### 5. Ontologie di riferimento europee.

- **Esempio: DOLCE (Descriptive Ontology for Linguistic and Cognitive Engineering):** Ontologia creata da un progetto finanziato dall'UE per rappresentare il significato concettuale della conoscenza.
- Struttura concetti come **eventi, qualità, oggetti e sostanze**.

Le upper ontologies sono fondamentali per garantire l'integrazione e la coerenza tra sistemi di rappresentazione della conoscenza diversi, facilitando l'interoperabilità tra applicazioni e basi di dati eterogenee.

#### **in sintesi:**

Le upper ontologies forniscono una struttura di riferimento per organizzare la conoscenza in modo sistematico e interoperabile. Strumenti come Cyc, DBpedia e DOLCE mostrano i diversi approcci alla creazione di ontologie di alto livello. L'integrazione di queste ontologie nel Semantic Web permette di migliorare la qualità dell'informazione e le capacità di inferenza automatica dei sistemi intelligenti.

# Frames and Semantic Networks

I **Frames** e le **Semantic Networks** sono due approcci alla rappresentazione della conoscenza che mirano a strutturare le informazioni in modo più intuitivo rispetto alla logica proposizionale o del primo ordine.

- a. **Frames**: rappresentano concetti attraverso *strutture dati gerarchiche*.
- b. **Semantic Networks**: utilizzano una rete di nodi e collegamenti per rappresentare relazioni tra concetti.

Questi modelli sono stati sviluppati *per migliorare l'organizzazione e l'interpretazione delle informazioni*, permettendo inferenze automatiche e facilitando il recupero della conoscenza.

## Frames

I **Frames** sono *strutture dati* per rappresentare situazioni o oggetti attraverso una serie di **attributi** (slot) e relativi **valori** (filler). Ogni frame ha un nome univoco e descrive un oggetto o una situazione tipica.

Esempio di Frame:

```
(tripLeg123  
  <:Instance-of TripLeg>  
  <:Destination Toronto>  
  <:DepartureDate "2024-02-01">  
)
```

In questo caso, il frame `tripLeg123` rappresenta un viaggio con destinazione Toronto e una data di partenza specifica.

Caratteristiche

- **Prototipi e Categorie:** Gli oggetti appartengono a categorie sulla base della somiglianza con un prototipo.
- **Ereditarietà:** Un frame può ereditare proprietà da un frame più generale.
- **Default e Eccezioni:** Gli slot possono avere valori predefiniti, ma è possibile modificarli se necessario.
- **Procedural Attachment:** Gli slot possono contenere procedure attivabili in base a specifici eventi.

Esempio di Procedural Attachment:

(city Toronto

<:Instance-of City>

<:Population [IF-NEEDED QueryPopulationDB]>

)

Qui, il valore della popolazione viene calcolato solo se necessario.

## Semantic Networks

Le **Semantic Networks** sono una rappresentazione grafica della conoscenza in cui gli **oggetti** e le **categorie** sono collegati da **relazioni**.

Esempio:

(Person → is-a → Mammal)

(Mammal → hasProperty → Warm-Blooded)

Questa rete rappresenta il fatto che una **persona** è un **mammifero**, e che i **mammiferi** sono a **sangue caldo**.

Caratteristiche

- **Inheritance:** Le proprietà possono essere ereditate dalle categorie superiori.



- **Direct inheritance:** Un frame può ereditare proprietà da un frame superiore.
- **Overridden Inheritance** → Uno slot può essere sovrascritto da un valore più specifico.
- **Procedural Attachment** → Uno slot può contenere una funzione o un'operazione eseguibile.
- **Collegamenti:** Ogni relazione tra concetti è rappresentata da un collegamento.
- **Risoluzione delle Ambiguità:** Si gestiscono casi di **ereditarietà multipla** e **sovrascrittura delle proprietà**.

Esempio di Problema di Ereditarietà Multipla:

(Person → is-a → Quaker)  
 (Person → is-a → Republican)  
 (Quaker → hasProperty → Pacifist)  
 (Republican → hasProperty → Not Pacifist)

In questo caso, si genera un conflitto nella classificazione di una persona.

Vantaggi e Limiti

- **Vantaggi:**
  - ✓ Rappresentazione intuitiva.
  - ✓ Inferenza diretta sulle relazioni.
- **Limiti:**
  - ✗ Non supportano facilmente relazioni complesse come la **negazione** e la **quantificazione universale**.
  - ✗ Problemi di ambiguità nell'ereditarietà multipla.

## Confronto tra Frames e Semantic Networks

Caratteristica	Frames	Semantic Networks
Struttura dati	Gerarchica	Grafica (nodi e archi)
Rappresentazione	Attributi e valori	Relazioni tra concetti
Inferenza	Basata su ereditarietà e regole	Basata su collegamenti
Gestione delle eccezioni	Default values, override	Difficile in caso di ereditarietà multipla

### in sintesi:

Frames e Semantic Networks offrono approcci complementari alla rappresentazione della conoscenza. I **Frames** sono più strutturati e utilizzano regole predefinite, mentre le **Semantic Networks** forniscono una rappresentazione visiva intuitiva delle relazioni tra concetti. Entrambi i modelli hanno influenzato lo sviluppo di sistemi esperti e la rappresentazione della conoscenza in intelligenza artificiale.

# Description Logics

La **Description Logic (DL)** è una famiglia di logiche formali progettate per rappresentare e ragionare sulle relazioni tra concetti e individui. È ampiamente utilizzata nel **Semantic Web** e nella **rappresentazione della conoscenza**, fornendo strumenti per definire **concetti**, **ruoli** e **individui** in modo formale.

Caratteristiche principali:

- **Expressivity**: Permette di descrivere gerarchie concettuali complesse.
- **Decidability**: È progettata per garantire la possibilità di eseguire inferenze computabili.
- **Automated Reasoning**: Supporta operazioni come **subsumption**, **satisfiability**, **classification** e **instance checking**.

## Struttura della Description Logic

- **Concetti (Concepts)**: Categorie o classi di oggetti.
- **Ruoli (Roles)**: Relazioni tra concetti o individui.
- **Individui (Individuals)**: Istanti specifici di concetti.

Esempio:

**Person**  $\sqsubseteq$  **LivingBeing**  
**Parent**  $\sqsubseteq$  **Person**  
**Mother**  $\sqsubseteq$  **Parent**  $\sqcap$  **Female**

## Operatori

DL utilizza una serie di operatori per costruire concetti complessi:

- **Intersezione** ( $\sqcap$ ): **Genitore**  $\sqcap$  **Impiegato**  $\rightarrow$  individui che sono sia genitori che impiegati.
- **Unione** ( $\sqcup$ ): **Student**  $\sqcup$  **Employee**  $\rightarrow$  individui che sono studenti o lavoratori.

- **Negazione** ( $\neg$ ):  $\neg\text{Parent} \rightarrow$  individui che non sono genitori.
- **Esistenziale** ( $\exists R.C$ ):  $\exists\text{hasChild.Student} \rightarrow$  individui con almeno un figlio studente.
- **Universale** ( $\forall R.C$ ):  $\forall\text{hasChild.Student} \rightarrow$  individui i cui figli sono tutti studenti.
- **Sottoinsieme** ( $\sqsubseteq$ ):  $\text{Professor} \sqsubseteq \text{Person} \rightarrow$  ogni professore è una persona.

## Varianti della Description Logic

La famiglia di Description Logic include diversi linguaggi con livelli di espressività differenti:

- **AL (Attributive Language)**: Versione più semplice con operatori base.
- **ALC (AL + Complemento)**: Aggiunge il complemento dei concetti ( $\neg C$ ).
- **SHOIN (D)**: Linguaggio alla base di **OWL-DL**, con gerarchie di ruoli, inverse properties, numeri cardinali e datatype.
- **SROIQ (D)**: Base di **OWL 2**, aggiunge ulteriore espressività con inclusione complessa di ruoli e nominali.

## ALC Description Logics

**ALC (Attributive Language with Complement)** è una logica descrittiva fondamentale che consente di rappresentare e ragionare su concetti e relazioni tra individui. È una delle basi delle Description Logics, utilizzata per costruire ontologie e modellare domini complessi.

### Operatori Supportati

#### 1. Negazione ( $\neg$ )

- Rappresenta il complemento di un concetto.
- Un individuo appartiene a  $\neg C$  se e solo se non appartiene al concetto  $C$ .

**Esempio:**

- **-Male**: Rappresenta tutti gli individui che non sono maschi

**2. AND ( $\sqcap$ )**

- Descrive l'intersezione di due concetti.
- Un individuo appartiene a **C  $\sqcap$  D** se appartiene sia a **C** che a **D**.

**Esempio:**

- **Student  $\sqcap$  Athlete**: Rappresenta gli individui che sono sia studenti che atleti.

**3. Universal Quantifier ( $\forall$ , ALL)**

- Restringe una relazione ai concetti specificati.
- Un individuo appartiene a  **$\forall R.C$**  se tutti i valori della relazione **R** appartengono al concetto **C**.
- **Esempio:**
  - **$\forall \text{hasChild.Male}$** : Rappresenta tutti gli individui che hanno solo figli maschi.

**4. Existential Quantifier ( $\exists$ , EXISTS)**

- Indica l'esistenza di almeno un valore per una relazione che soddisfa un concetto.
- Un individuo appartiene a  **$\exists R.C$**  se esiste almeno un valore della relazione **R** che appartiene al concetto **C**.

**SHOIN(D)**

è la logica su cui si basa **OWL-DL**, una delle versioni dell'**Ontology Web Language (OWL)**. Questa logica include varie caratteristiche, tra cui:

- **S** per ruoli transitivi,
- **H** per gerarchie di ruoli,
- **O** per nomi nominali (individui specifici),
- **I** per proprietà inverse,
- **N** per restrizioni di cardinalità,
- **(D)** per l'uso di datatype properties e valori numerici

## **SROIQ(D)**

è un'estensione più espressiva delle logiche descrittive, ed è la base di **OWL 2**, una versione migliorata di OWL-DL. Questa logica introduce:

- **R** per inclusione complessa di ruoli,
- **Q** per restrizioni di cardinalità qualificate,
- **I** per proprietà inverse,
- **D** per datatype properties,
- **O** per nominali.

## Knowledge Bases in DL

Le **Knowledge Bases (KB)** in DL sono suddivise in due parti:

1. **TBox (Terminological Box)**: Contiene assiomi sui concetti e sui ruoli.

**Parent**  $\sqsubseteq$  **Person**  
**hasChild**  $\sqsubseteq$  **Relation**

2. **ABox (Assertional Box)**: Contiene asserzioni sui singoli individui.

**Mary** : **Parent**  
**(Mary, John)** : **hasChild**

## Ragionamento nella Description Logic

Il **ragionamento automatico** consente di inferire nuove conoscenze a partire dai dati esistenti.

Operazioni principali:

1. **Satisfiability**: Determina se un concetto può avere almeno un'istanza valida.
2. **Subsumption**: Verifica se un concetto è un sottoinsieme di un altro ( $A \sqsubseteq B$ ).
3. **Classification**: Organizza una gerarchia concettuale ordinando le relazioni di subsumption.
4. **Instance Checking**: Determina se un individuo appartiene a un dato concetto.
5. **Concept Equivalence**: Determina se due concetti sono semanticamente equivalenti ( $A \equiv B$ ).
6. **Consistency Checking**: Determina se l'insieme di asserzioni e assiomi nella KB è privo di contraddizioni.

**Esempio di inferenza**: Se sappiamo che  $\text{Professor} \sqsubseteq \text{Person}$  e  $\text{Mary} : \text{Professor}$ , possiamo inferire automaticamente che  $\text{Mary} : \text{Person}$ .

$\text{Professor} \sqsubseteq \text{Person}$

$\text{Mary} : \text{Professor}$

→ **Inferenza**:  $\text{Mary} : \text{Person}$

## Implementazioni e Reasoners

I **reasoners** sono strumenti software che *eseguono il ragionamento* sulla knowledge base DL. Alcuni dei più noti:

- **FaCT++**: Reasoner altamente ottimizzato per DL.
- **Pellet**: Supporta OWL-DL e reasoning con dati numerici.
- **HermiT**: Algoritmi avanzati per reasoning efficiente.

**in sintesi:**

La **Description Logic** offre un quadro formale potente per la rappresentazione della conoscenza, combinando espressività e decidibilità. Viene utilizzata in ontologie, sistemi esperti e applicazioni del **Semantic Web**, facilitando il ragionamento automatico e l'inferenza di nuove informazioni a partire da dati strutturati.



# Protégé

**Protégé** è un editor di ontologie open-source sviluppato dalla Stanford University. È utilizzato per creare, modificare e gestire ontologie per il **Semantic Web** e altre applicazioni di **Intelligenza Artificiale**.

Caratteristiche principali:

- **Basato su Java**, supporta plug-in per estendere le funzionalità.
- **Supporta OWL (Ontology Web Language)**, RDF(S) e altri formati.
- **Interfaccia modulare**, con sched

Il processo di sviluppo di un'ontologia in Protégé segue diversi passi fondamentali:

1. **Analisi del dominio e degli obiettivi dell'ontologia.**
2. **Riutilizzo di ontologie esistenti**, se disponibili.
3. **Definizione dei concetti chiave del dominio.**
4. **Organizzazione dei concetti in classi e gerarchie.**
5. **Definizione delle proprietà delle classi.**
6. **Aggiunta di vincoli sulle proprietà** (range, dominio, cardinalità).
7. **Creazione delle istanze.**
8. **Assegnazione dei valori alle proprietà per ogni istanza.**

## Componenti principali

- **Tab “Classes”**: Gestione delle classi e della loro gerarchia.
- **Tab “Object Properties”**: Definizione delle relazioni tra le classi.
- **Tab “Data Properties”**: Proprietà con valori primitivi (es. numeri, stringhe).
- **Tab “Individuals”**: Creazione e gestione delle istanze.
- **Tab “Forms”**: Definizione di moduli per l'inserimento dei dati.

Food  $\sqsubseteq$  Thing  
Pizza  $\sqsubseteq$  Food  
Margherita  $\sqsubseteq$  Pizza

## Proprietà e Restrizioni

Le proprietà in Protégé si dividono in:

- **Object Properties:** Relazioni tra due individui.
- **Data Properties:** Collegano un individuo a un valore primitivo.
- **Annotation Properties:** Utilizzate per aggiungere metadati.

**ObjectProperty:** hasTopping  
**Domain:** Pizza  
**Range:** Topping

Le proprietà possono avere diverse caratteristiche:

- **Inverse properties:** Se **hasParent** è definita, possiamo inferire automaticamente **hasChild**.
- **Functional properties:** Un individuo può essere collegato a massimo un altro individuo tramite questa proprietà.
- **Transitive properties:** Se **A hasAncestor B** e **B hasAncestor C**, allora **A hasAncestor C**.
- **Symmetric properties:** Se **A hasBrother B**, allora **B hasBrother A**.
- **Reflexive properties:** Ogni individuo è in relazione con sé stesso

## Reasoning in Protégé

Protégé utilizza diversi reasoner per eseguire **reasoning** con motori di inferenza come:

- FaCT++ → Ottimizzato per SHOIN(D), molto veloce per grandi ontologie.
- Pellet → Supporta reasoning su datatype properties (es. numeri, date).
- HermiT → Basato su un tableaux algorithm ottimizzato, ideale per OWL 2.

Funzioni principali del reasoner:

1. **Inferenza di una gerarchia ontologica.**
2. **Verifica della consistenza di un'ontologia.**
3. **Inferenza di nuove relazioni tra individui.**

Esempio di inferenza:

**Person**  $\sqsubseteq$  **LivingBeing**  
**Parent**  $\sqsubseteq$  **Person**  
**hasChild**  $\sqsubseteq$  **ObjectProperty**  
**Mary: Parent**  
**John: Person**  
**(Mary, John): hasChild**  
→ **Inferenza: John è un individuo di Person**

Open World Assumption (OWA) vs. Closed World Assumption (CWA)

Protégé e OWL operano secondo l'**Open World Assumption (OWA)**, che differisce dal **Closed World Assumption (CWA)** tipico di database e sistemi logici come Prolog.

Le **Description Logics (DL)** e OWL sono basate su **Open World Assumption (OWA)**, mentre linguaggi come **Prolog** e i database relazionali utilizzano **Closed World Assumption (CWA)**

- **OWA (Open World Assumption)** → Il fatto che un'asserzione non sia dichiarata **non significa che sia falsa**.
- **CWA (Closed World Assumption)** → Se un fatto non è presente nel database, **si assume che sia falso**.

#### **Implicazioni nel Reasoning:**

- OWA consente di modellare conoscenze **incomplete** ed è quindi utile in scenari con conoscenza distribuita o parziale (es. Semantic Web).
- CWA è più adatto a sistemi con dati completamente noti, come database relazionali.

# Semantic web and Knowledge Graphs

Il **Semantic Web** è un'estensione del Web tradizionale che mira *a rendere i dati interpretabili e utilizzabili dalle macchine attraverso l'uso di ontologie e metadati strutturati*. La sua finalità è consentire un'integrazione e un'interoperabilità più avanzata tra le informazioni.

## Caratteristiche principali:

- **Structured Metadata:** Utilizza standard come RDF, OWL e SPARQL per rappresentare dati in modo significativo.
- **Interoperability:** Consente la condivisione e il riutilizzo della conoscenza tra diversi sistemi.
- **Automated Reasoning:** Permette ai motori di inferenza di dedurre nuove informazioni da dati esistenti.
- **Basato su URIs:** L'uso di identificatori univoci garantisce la consistenza e il collegamento dei dati.

## Strumenti e Tecnologie

### 1. RDF (Resource Description Framework)

- Rappresenta le informazioni attraverso **triple** nella forma (**soggetto, predicato, oggetto**).
- Può essere integrato con database relazionali.
- **RDFS (RDF Schema)** → Aggiunge gerarchie

### 2. OWL (Web Ontology Language)

- Permette la definizione di ontologie con relazioni gerarchiche, vincoli e proprietà avanzate.
- Supporta ragionamento automatico per derivare conoscenze implicite.
- OWL Lite, OWL DL e OWL Full offrono diversi livelli di espressività.

### 3. SPARQL (SPARQL Protocol and RDF Query Language)

- Linguaggio per interrogare basi di dati RDF.
- Può restituire risultati in vari formati come JSON, XML e RDF.

# Knowledge Graphs

I **Knowledge Graphs** rappresentano un metodo per organizzare e collegare informazioni strutturate in una rete semantica.

Ricordiamoci che:

**Ontology** → Modello formale che definisce concetti e relazioni. Il suo obiettivo è quello di definire regole e vincoli logici. (es. OWL Ontology)

**Knowledge Graph** → Collezione di dati **interconnessi** con semantica basata su un'ontologia con l'obiettivo di organizzare ed estrarre conoscenza da dati collegati. (es. **Google Knowledge Graph** collega **Albert Einstein** con **Relativity**.)

## Caratteristiche dei Knowledge Graphs

- **Interconnected Entities:** Strutturano i dati tramite nodi e relazioni.
- **Integration of Multiple Data Sources:** Raccolgono informazioni da molteplici dataset.
- **Semantic Inference:** Permettono deduzioni automatiche basate sulle relazioni tra entità.
- **Google Knowledge Graph:** Un esempio di implementazione su larga scala.

## Applicazioni del Semantic Web e dei Knowledge Graphs

- **Motori di ricerca intelligenti** (es. Google Knowledge Graph, Wikidata)
- **Sistemi di raccomandazione** (Netflix, Amazon)
- **Interoperabilità nei Big Data** (Linked Data: I **Linked Data** permettono di connettere dati strutturati attraverso il **Web Semantico**. **SPARQL** è il linguaggio di interrogazione per RDF, simile a SQL per database relazionali.)

- **Medicina e biotecnologie** (ontologie cliniche)
- **Integrazione con Prolog e logiche descrittive.**

**in sintesi:**

Il **Semantic Web** e i **Knowledge Graphs** offrono un'infrastruttura per la rappresentazione e l'inferenza della conoscenza in modo strutturato e interpretabile dalle macchine. Le tecnologie come RDF, OWL e SPARQL consentono una gestione avanzata delle informazioni, facilitando applicazioni in ambiti disparati come il Web, la ricerca scientifica e l'industria.

# Ragionamento Temporale

Il ragionamento temporale riguarda *la rappresentazione e il trattamento della conoscenza nel tempo*. Questo è fondamentale per molte applicazioni, tra cui:

- Pianificazione automatica
- Verifica di sistemi software
- Modellazione di processi aziendali
- Intelligenza artificiale e logica del tempo.

## Logiche Temporal

Le **logiche temporali** sono estensioni della logica proposizionale e del primo ordine che includono operatori per esprimere il tempo. Le principali categorie di logiche temporali sono:

1. **Linear Temporal Logic (LTL)**: assume una progressione lineare del tempo ( $\Box, \Diamond, \bigcirc, U$ ).
2. **Computational Tree Logic (CTL)**: permette di modellare ramificazioni temporali. ( $A, E, X, F, G, U$ ).
3. **Allen's Interval Algebra**: consente di descrivere relazioni tra intervalli temporali. (*Before, Meets, Overlaps*).
4. **Event Calculus (EC)**: basato sugli eventi e sulla loro influenza nel tempo. (*Happens, Initiates, Terminates*).

### Linear Temporal Logic (LTL)

LTL è una logica modale che permette di esprimere proprietà che si verificano lungo una linea temporale discreta e infinita.

Operatori principali di LTL:

- $\bigcirc$  (**Next**): la proprietà è vera nel prossimo stato.
- $\Diamond$  (**Eventually**): la proprietà sarà vera in qualche momento futuro.

- $\square$  (**Always**): la proprietà è sempre vera.
- **U (Until)**: una formula deve rimanere vera fino a quando un'altra diventa vera.

Esempio di formula LTL:

$\square (p \rightarrow \bigcirc q)$  // Se p è vero ora, allora q sarà vero nel prossimo stato.

## Allen's Interval Algebra

James Allen ha sviluppato un insieme di operatori per descrivere relazioni tra intervalli temporali, tra cui:

- **Before**: Un intervallo termina prima che un altro inizi.
- **Meets**: Un intervallo termina esattamente quando un altro inizia.
- **Overlaps**: Due intervalli si sovrappongono.
- **During**: Un intervallo è contenuto all'interno di un altro.

Esempio:

**Evento A: [10:00 - 11:00]**

**Evento B: [10:30 - 11:30]**

**Relazione: A Overlaps B**

## Event Calculus e Monitoraggio degli Eventi

L'**Event Calculus** (EC) è un formalismo logico *per rappresentare e ragionare su eventi e il loro effetto nel tempo.*

Principali concetti:

- **Happens(E, T)**: L'evento E accade al tempo T.
- **HoldsAt(F, T)**: Il fluent F è vero al tempo T.
- **Initiates(E, F, T)**: L'evento E rende vero il fluent F a partire dal tempo T.



- **Terminates(E, F, T)**: L'evento **E** rende falso il fluent **F** a partire dal tempo **T**.

Esempio:

**Happens(OrderCoffee, 10:00)**

**Initiates(OrderCoffee, WaitingForCoffee, 10:00)**

Questo significa che l'evento **OrderCoffee** accade alle 10:00 e porta lo stato **WaitingForCoffee** a essere vero.

## Applicazioni del Ragionamento Temporale

Il **Model Checking** è una tecnica usata per verificare proprietà di sistemi software e hardware utilizzando **logiche temporali**.

Metodo	Descrizione
<b>Model Checking con LTL</b>	Controlla proprietà su <b>una singola sequenza temporale</b> .
<b>Model Checking con CTL</b>	Controlla proprietà su <b>tutte le possibili evoluzioni</b> di un sistema.

Il ragionamento temporale è ampiamente utilizzato in diversi ambiti:

1. **Sicurezza informatica**: Monitoraggio di intrusioni basato su eventi temporali.
2. **Process Mining**: Analisi delle sequenze di eventi nei processi aziendali.
3. **Sistemi di controllo industriale**: Supervisione di eventi in impianti automatizzati.
4. **Sistemi sanitari**: Monitoraggio delle condizioni dei pazienti nel tempo.

**in sintesi:**

Il ragionamento temporale è un elemento chiave nell'intelligenza artificiale e nell'analisi dei processi. Le logiche temporali, l'Event Calculus e l'algebra degli intervalli di Allen offrono strumenti potenti per modellare e analizzare fenomeni nel tempo. Grazie all'uso di formalismi precisi, è possibile garantire l'affidabilità e l'efficacia di sistemi complessi basati sul tempo.

# Modal logics and Linear Temporal Logic (LTL)

## Introduzione alle Logiche Modali

Le **logiche modali** estendono la logica proposizionale introducendo operatori che esprimono possibilità e necessità. Sono ampiamente utilizzate per modellare conoscenza, credenze, e fenomeni temporali.

Caratteristiche principali:

- **Operatori modali:**  $\Diamond$  (possibile) e  $\Box$  (necessario).
- **Accessibilità tra mondi possibili:** Definisce quali stati sono raggiungibili da un dato stato.
- **Axiomi distintivi per diversi sistemi modali:** K, T, S4, S5.

Esempio:

$\Box p \rightarrow p$  // Se  $p$  è necessariamente vero, allora è vero.

$\Diamond p \rightarrow \Box \Diamond p$  // Se  $p$  è possibile, allora è necessariamente possibile.

## Propositional Attitudes e Modal Logic

Le logiche modali sono utilizzate per modellare **atteggiamenti proposizionali** come conoscenza e credenze.

Esempi:

**Federico knows that there is a pizza in the fridge.**

**Federico believes that there is a pizza in the fridge.**

Queste affermazioni possono essere rappresentate tramite l'operatore di conoscenza K:

**K\_federico (location(pizza, fridge))**

Logica	Descrizione
<b>LTL (Linear Temporal Logic)</b>	Assume che il tempo sia <b>lineare</b> ( $\bigcirc, \square, \Diamond, U$ ).
<b>CTL (Computational Tree Logic)</b>	Permette <b>ramificazioni</b> temporali con operatori <b>A</b> ( <i>for all paths</i> ), <b>E</b> ( <i>exists a path</i> ).
<b>CTL*</b>	Unisce la flessibilità di LTL e CTL, permettendo formule miste

## Linear Temporal Logic (LTL)

**LTL** è una logica temporale che esprime proprietà su sequenze di stati lungo una linea temporale infinita. È usata per la verifica di modelli e la specifica di sistemi reattivi.

### Operatori temporali principali:

- **X (Next)**: La proprietà è vera nel prossimo stato.
- **F (Eventually)**: La proprietà sarà vera in qualche momento futuro.
- **G (Globally)**: La proprietà è sempre vera.
- **U (Until)**: Una formula deve rimanere vera fino a quando un'altra diventa vera.

Esempio di formula LTL:

$G(\text{request} \rightarrow F(\text{response}))$  // Ogni richiesta deve essere seguita eventualmente da una risposta.

## Semantica di LTL

Un modello LTL è una struttura temporale  $(K, <)$  in cui:

- $K$  è un insieme totalmente ordinato.
- Gli eventi sono valutati nei punti della linea temporale.

Esempio di **trace LTL**:

p q r p q r ... // Stati di un sistema in sequenza temporale.

## Relazione tra Logiche Modali e LTL

LTL può essere vista come una logica modale specializzata per il tempo, in cui l'accessibilità tra mondi rappresenta la progressione temporale.

- **Modelli di Kripke**: Stati con transizioni temporali definite.
- **Verifica di modelli (Model Checking)**: Determina se una specifica LTL è soddisfatta da un sistema. È una tecnica per verificare proprietà su sistemi dinamici.

Ricordiamoci che:

**Model Checking per Modal Logics** → Verifica formule su mondi possibili

**Model Checking per LTL** → Verifica proprietà su sequenze temporali

## Applicazioni di LTL e Logiche Modali

- **Verifica di protocolli di comunicazione**: Assicurare che le proprietà temporali siano soddisfatte.
- **Sicurezza informatica**: Specificare vincoli di accesso e propagazione della conoscenza.

- **Process Mining e BPM:** Modellazione e analisi dei processi aziendali.
- **Atteggiamenti proposizionali in agenti intelligenti:** Modellare conoscenza e credenze degli agenti.

**in sintesi:**

Le logiche modali e LTL sono strumenti fondamentali per la modellazione e la verifica di sistemi dinamici. Mentre le logiche modali descrivono possibilità e necessità tra mondi possibili, LTL è particolarmente efficace nel ragionamento su eventi temporali e sequenze di stati nei sistemi reattivi e nei processi aziendali.

# Prolog LPAD

## Logic Programs with Annotated Disjunctions

### Introduzione a Prolog LPAD

**Prolog LPAD** *estende Prolog classico* con il supporto alla **logica probabilistica**, consentendo di modellare incertezze (e gestirle) e inferenze probabilistiche. È utile per applicazioni come **data mining, inferenza statistica e AI basata su regole**.

Differenze chiave tra **Prolog Classico** e **LPAD**:

Caratteristica	Prolog Classico	LPAD
<b>Inferenza</b>	Deterministica	Probabilistica
<b>Tipo di regole</b>	Regole logiche standard	Regole con probabilità
<b>Modello del mondo</b>	Chiuso (Closed World Assumption)	Probabilistico

Esempio in **Prolog classico**:

```
piove.  
bagnato :- piove.
```

Qui, **bagnato** è sempre vero se **piove** è definito.

Esempio equivalente in **LPAD**:

```
piove:0.7; non_piove:0.3.  
bagnato :- piove.
```

In questo caso, **piove** ha una probabilità del 70%, quindi **bagnato** non è sempre certo.

Caratteristiche principali:

- **Estende Prolog con probabilità associate a regole.**
- **Supporta l'inferenza probabilistica** per modelli incerti. (L'inferenza in **LPAD** e **ProbLog** permette di calcolare la probabilità di una query data una conoscenza incerta.)
- **Basato sulla semantica della distribuzione**, che genera mondi possibili e calcola la probabilità di una query.

## Motivazioni e Applicazioni

Prolog classico utilizza logica booleana, che **non consente di esprimere incertezze**. LPAD combina il ragionamento logico con probabilità, utile per:

- **Diagnosi mediche probabilistiche.**
- **Analisi di rischio** (es. valutazione probabilistica di cadute negli anziani).
- **Modellazione della conoscenza incerta.**

Esempio:

```
sneezing(X):0.7 ; null:0.3 :- flu(X).  
sneezing(X):0.8 ; null:0.2 :- hay_fever(X).
```

Questa regola indica che una persona con influenza ha il 70% di probabilità di starnutire, mentre una persona con allergia ha l'80%.



## Distribuzione della Probabilità nei Mondi Possibili

LPAD calcola le probabilità di una query generando **tutti i mondi possibili**, selezionando per ogni clausola una delle alternative disponibili.

Esempio di probabilità aggregata:

$$\begin{aligned} P(\text{sneezing}(\text{bob})) &= P(w1) + P(w2) + P(w3) \\ &= (0.7 \times 0.8) + (0.3 \times 0.8) + (0.7 \times 0.2) \\ &= 0.94 \end{aligned}$$

Questa equazione somma le probabilità in tutti i mondi in cui `sneezing(bob)` è vero.

## Semantica della Distribuzione

In LPAD, una **scelta atomica** è la selezione di un valore dalla testa di una clausola con probabilità associata. Un **mondo possibile** è una combinazione di scelte atomiche.

Passaggi:

1. **Grounding** delle regole di Prolog.
2. **Scelta di un valore per ogni regola probabilistica.**
3. **Calcolo delle probabilità totali aggregando tutti i mondi possibili.**

Esempio:

```
sneezing(bob):0.7 ; null:0.3 :- flu(bob).  
sneezing(bob):0.8 ; null:0.2 :- hay_fever(bob).
```

Risultato: Bob ha **il 94% di probabilità di starnutire**, considerando entrambi i fattori.

## Strumenti e Frameworks per LPAD

Due principali strumenti software supportano LPAD:

**1. cplint**

- ✓ Basato su SWI-Prolog, implementa inferenza, apprendimento dei parametri e apprendimento della struttura.

**2. ProbLog**


- ✓ Supporta inferenza probabilistica su larga scala, implementa l'algoritmo LFI-ProbLog per il learning.

**Esempio: Il Problema di Monty Hall in LPAD**

LPAD può essere utilizzato per modellare giochi probabilistici come il **problema di Monty Hall**:

```
prize(1):1/3; prize(2):1/3; prize(3):1/3.  
open_door(2):0.5 ; open_door(3):0.5 :- prize(1).  
open_door(2):- prize(3).  
open_door(3):- prize(2).  
win_keep :- prize(1).  
win_switch :- prize(2), open_door(3).  
win_switch :- prize(3), open_door(2).
```

Risultato: **Cambiare scelta aumenta la probabilità di vittoria.**

 **ProbLog:** Un'alternativa a LPAD è **ProbLog**, che permette inferenza probabilistica su larga scala ed è utilizzato per applicazioni di *machine learning* e *data mining*.

**in sintesi:**

**LPAD** consente di combinare il ragionamento logico con l'incertezza, permettendo di esprimere conoscenza probabilistica in Prolog. È utile per applicazioni in AI, data mining e analisi del rischio. Strumenti come **cplint** e **ProbLog** supportano l'inferenza probabilistica su larga scala.

# Forward Reasoning

## Introduzione ai Sistemi Basati su Regole

I **Rule-Based Systems** sono un modello di rappresentazione della conoscenza in cui *il ragionamento viene effettuato attraverso un insieme di regole di produzione*. Le regole sono tipicamente rappresentate nella forma:

**Se (Condizione) Allora (Azione)**

Questi sistemi sono **particolarmente utilizzati** nei sistemi esperti e nelle applicazioni di intelligenza artificiale *per la risoluzione di problemi complessi in modo strutturato e scalabile*.

**Forward Chaining** → Parte dai fatti e applica regole per inferire nuove conoscenze (**data-driven**).

**Backward Chaining** → Parte da un obiettivo e cerca di dimostrarlo (**goal-driven**).

## Il Meccanismo di Forward Reasoning

Il **forward reasoning** o **data-driven reasoning** *parte dai fatti noti e applica le regole per derivare nuove informazioni*. Il processo segue questi passi:

1. **Riconoscimento (Matching)**: Identificare le regole che corrispondono ai dati disponibili.
2. **Conflict Resolution**: Se più regole possono essere applicate, viene scelto l'ordine di esecuzione.
3. **Esecuzione delle regole**: Si eseguono le azioni corrispondenti alle regole attivate.
4. **Ripetizione**: Il processo continua fino a quando non vengono più applicate nuove regole o si raggiunge un obiettivo.

Esempio di forward reasoning:

1. Se una persona ha febbre e tosse → Potrebbe avere l'influenza.
2. Se una persona ha l'influenza → Dovrebbe riposare e assumere liquidi.

## L'Approccio delle Regole di Produzione

Le **production rules** sono il cuore dei sistemi basati su regole. Queste regole si basano su condizioni di antecedenti e conseguenti:

**Regola 1:** Se X è un mammifero e X allatta i piccoli, allora X è un animale viviparo.

**Regola 2:** Se X è un animale viviparo e ha il pelo, allora X è un mammifero.

Le regole di produzione vengono utilizzate in vari ambiti, tra cui:

- Sistemi esperti in medicina
- Automazione industriale
- Sistemi di supporto alle decisioni.

## L'Algoritmo RETE

L'**algoritmo RETE** è un metodo efficiente per il matching delle regole nei sistemi basati su regole. Funziona memorizzando i pattern delle regole in una rete di nodi per ottimizzare la ricerca e l'applicazione delle regole.

Principali caratteristiche dell'algoritmo RETE:

- a. **Ottimizzazione del matching:** Riduce il numero di confronti necessari tra regole e fatti.
- b. **Struttura gerarchica:** Utilizza una rete di nodi per rappresentare e confrontare le regole in modo efficiente.
  - **Alpha Nodes** → Filtrano le condizioni di ogni regola.

- **Beta Nodes** → Combinano più condizioni per attivare regole
- c. **Miglioramento delle performance:** Particolarmente utile nei sistemi con un grande numero di regole e fatti.

💡 - **Matching delle Regole:** Confronta regole con i dati della **Working Memory**.

- **Ottimizzazione:** Evita confronti ridondanti utilizzando una rete di nodi (Alpha e Beta).

## **Applicazioni dei Sistemi Basati su Regole**

I sistemi basati su regole sono utilizzati in diversi settori:

- ✓ **Diagnosi medica:** Sistemi esperti per l'analisi di sintomi e patologie.
- ✓ **Automazione industriale:** Controllo di processi produttivi basato su regole predefinite.
- ✓ **Filtraggio delle informazioni:** Sistemi di raccomandazione e analisi dei dati.
- ✓ **Intelligenza Artificiale:** Implementazione di agenti intelligenti con ragionamento automatico.

### **in sintesi:**

I sistemi basati su regole rappresentano una metodologia efficace per la rappresentazione della conoscenza e il ragionamento automatico. Il forward reasoning, combinato con algoritmi come RETE, permette di ottimizzare il processo di inferenza, rendendo questi sistemi adatti a una vasta gamma di applicazioni nell'**intelligenza artificiale**, nell'**automazione** e nei **sistemi esperti**.

# Business process management (BPM)

Il **Business Process Management (BPM)** è un insieme di **concetti, metodi e tecniche** per la progettazione, amministrazione, configurazione, esecuzione e analisi dei processi aziendali. Il BPM aiuta a ottimizzare le operazioni aziendali, migliorando **efficienza, qualità e conformità ai requisiti normativi**

## Obiettivi principali:

- Migliorare la comprensione delle operazioni aziendali e delle loro relazioni.
- Ottenere maggiore flessibilità e capacità di adattamento ai cambiamenti.
- Automatizzare e ottimizzare i processi aziendali per ridurre costi e tempi.
- Supportare la presa di decisioni con informazioni basate su dati.

## Ciclo di Vita dei Processi Aziendali

Il BPM segue un ciclo di vita iterativo, suddiviso in quattro fasi:

1. **Design & Analysis** – Modellazione e verifica dei processi.
2. **Configuration** – Implementazione e personalizzazione del processo.
3. **Enactment** – Esecuzione del processo e raccolta di dati.
4. **Evaluation** – Analisi delle prestazioni e ottimizzazione.

## Modellazione dei Processi Aziendali

La modellazione dei processi aziendali consente di descrivere graficamente il flusso delle attività, migliorando la **comprensione e l'ottimizzazione**.

- **Process Modeling** → Rappresentazione grafica del flusso di lavoro.
- **Process Execution** → Implementazione ed esecuzione del processo in un sistema BPM.
- **Process Mining** → Analisi dei log per migliorare i processi.

Tipologie di processi:

a. **Organizational vs. Operational**

- **Organizational:** Descrivono il contesto generale e gli obiettivi aziendali.
- **Operational:** Specificano le attività dettagliate per l'implementazione.

b. **Intra vs. Inter-organizational**

- **Intra-organizational:** Tutte le attività sono svolte all'interno della stessa organizzazione.
- **Inter-organizational:** Coinvolgono più aziende o partner esterni.

**Approcci alla modellazione:**

1. **Procedurale** (Closed)

- Specifica l'ordine preciso delle attività.
- Facilmente implementabile nei workflow automatizzati.

2. **Dichiarativo** (Open)

- Definisce solo le **regole** e i **vincoli** del processo.
- Maggiore flessibilità nell'esecuzione.

Pattern di Workflow e Controllo di Flusso

I workflow definiscono come le attività vengono coordinate ed eseguite. Alcuni pattern comuni:

- **Sequence:** Una attività segue un'altra.
- **Parallel Split (AND-Split):** Due o più attività vengono eseguite contemporaneamente.
- **Exclusive Choice (XOR-Split):** Una scelta tra due alternative esclusive.
- **Synchronization (AND-Join):** Unisce flussi paralleli in un unico percorso.
- **Multiple Instances:** Un'attività viene eseguita più volte simultaneamente.
- **Arbitrary Cycles:** Permette ripetizioni illimitate di un'attività.
-



Esempio:

**A → B → C // Sequenza**

**A → (B | C) → D // Scelta esclusiva (XOR-Split)**

**A → [B & C] → D // Esecuzione parallela (AND-Split)**

## **Allocazione delle Risorse nei Processi**

La gestione delle risorse è fondamentale nei processi aziendali. Alcuni schemi di allocazione delle risorse:

- **Direct Allocation:** Un'attività è assegnata a un individuo specifico.
- **Role-Based Allocation:** Un'attività è assegnata a un ruolo specifico.
- **Authorization Allocation:** L'allocazione dipende da autorizzazioni e ruoli aziendali.
- **History-Based Allocation:** L'allocazione dipende dalle attività precedentemente eseguite dall'utente.

## **Business Process Mining**

Il **Process Mining** analizza i log dei processi aziendali per estrarre modelli e individuare inefficienze.

Principali tecniche di Process Mining:

- **Discovery:** Estrazione automatica di modelli di processo dai dati.
- **Conformance Checking:** Confronto tra il modello estratto e il modello atteso.
- **Enhancement:** Ottimizzazione dei processi sulla base dei dati analizzati.

Esempio:

Evento: **Ordine ricevuto → Pagamento effettuato → Spedizione**

Process Mining: **Identifica colli di bottiglia nel tempo di pagamento.**

**in sintesi:**

Il **Business Process Management** è un approccio sistematico per modellare, analizzare e ottimizzare i processi aziendali. Strumenti come **workflow patterns**, **allocazione delle risorse** e **process mining** aiutano le aziende a migliorare l'efficienza e la qualità delle operazioni.

**Approcci futuri includono:**

- ✓ Maggiore integrazione con **AI e machine learning**.
- ✓ Automazione avanzata tramite **robotic process automation (RPA)**.
- ✓ Ottimizzazione continua attraverso **big data e analytics**.

# Procedural languages for process modeling

## Introduzione ai Linguaggi Procedurali per la Modellazione dei Processi

I **linguaggi procedurali** per la modellazione dei processi aziendali *descrivono il flusso di lavoro come una sequenza ben definita di passi*. Sono caratterizzati da un'**esecuzione deterministica** e una struttura **chiusa**, dove ogni attività ha una posizione ben definita all'interno del processo.

### Caratteristiche principali:

- **Struttura chiusa**: l'ordine delle attività è prefissato e ben definito.
- **Sequenzialità**: i processi seguono un flusso deterministico.
- **Automatizzabilità**: ideali per workflow gestiti da macchine.
- **Espressione chiara del controllo di flusso**: tramite costrutti come sequenza, scelta, cicli e sincronizzazione.

Nel **Process Modeling**, vengono utilizzati diversi linguaggi:

- **BPMN (Business Process Model and Notation)** → Standard visivo per rappresentare processi aziendali.
- **Petri Nets** → Formalismo matematico per modellare processi con analisi rigorosa.
- **YAWL (Yet Another Workflow Language)** → Linguaggio basato su **Workflow Nets**, più espressivo di BPMN.

### Petri Nets

Uno dei principali formalismi per rappresentare i processi procedurali è la **Rete di Petri**, che rappresenta il flusso di esecuzione con **nodi e archi**, consentendo un'analisi rigorosa delle proprietà del processo.

### Elementi fondamentali delle Petri Nets:

- **Places (Stati)**: rappresentano condizioni o risorse.
- **Transitions (Eventi/Azioni)**: rappresentano attività o eventi che cambiano lo stato del sistema.
- **Archi direzionati**: collegano stati e transizioni, determinando il flusso del processo.
- **Token**: rappresentano lo stato attuale del processo e si spostano attraverso la rete durante l'esecuzione.

Esempio di una Rete di Petri:

(Pagamento) → [Verifica] → (Spedizione)

Dove (Pagamento) e (Spedizione) sono stati, e [Verifica] è una transizione.

Workflow Nets (WF-nets)

Le **Workflow Nets (WF-nets)** sono una classe specifica di **Reti di Petri** utilizzate per modellare i workflow aziendali.

- Supportano la rappresentazione gerarchica dei processi.
- Gestiscono il controllo di flusso con vincoli ben definiti.
- Permettono la verifica della correttezza formale del modello..

Esempio:

(Ordine ricevuto) → [Elaborazione] → (Spedizione) → (Fine processo)

YAWL (Yet Another Workflow Language)

YAWL è un linguaggio di modellazione basato su **Workflow Nets**, progettato per superare le limitazioni dei modelli tradizionali di BPMN e Petri Nets.

**Caratteristiche principali di YAWL:**

- **Gestione avanzata delle istanze multiple.**
- **Espressione esplicita di comportamenti split e join.**
- **Supporto per workflow complessi con cicli e decisioni.**

\*\*Possibilità di modellare **cancellazioni non locali** e interazioni dinamiche tra attività.

## BPMN (Business Process Model and Notation)

**BPMN** è uno standard per la modellazione dei processi aziendali che combina semplicità grafica con una semantica formale chiara.

### Differenze tra BPMN e i linguaggi procedurali classici:

- ✓ BPMN è più **espressivo** e può modellare sia workflow rigidi che flessibili.
- ✓ Supporta **interazioni tra più partecipanti** attraverso messaggi.
- ✓ Permette di gestire **eventi e eccezioni**, cosa che non è immediata nei linguaggi procedurali puri.

### Confronto tra Linguaggi Procedurali e Dichiarativi

Caratteristica	Procedurale	Dichiarativo
<b>Definizione del flusso</b>	Specificato rigidamente	Basato su regole e vincoli
<b>Flessibilità</b>	Bassa ✗	Alta ✓
<b>Esecuzione</b>	Deterministica	Più adattabile

<b>Espressività</b>	Limitata alla sequenza definita	Supporta comportamenti imprevisti
---------------------	------------------------------------	--------------------------------------

**in sintesi:**

I **linguaggi procedurali per la modellazione dei processi** sono strumenti potenti per la gestione strutturata dei workflow aziendali. Strumenti come **Petri Nets**, **YAWL** e **BPMN** permettono di modellare e analizzare i processi con precisione, garantendo l'ottimizzazione delle operazioni aziendali.

I linguaggi procedurali sono particolarmente utili in **scenari in cui il controllo del flusso è ben definito**, mentre per processi più flessibili si preferiscono approcci **dichiarativi**.

# Formal properties for BPM languages - declarative languages

I **linguaggi dichiarativi** per la modellazione dei processi aziendali *si concentrano sulla definizione di regole e vincoli* che devono essere rispettati piuttosto che sulla sequenza rigida di attività. Questo approccio consente una maggiore flessibilità, adattabilità e gestione di situazioni impreviste.

## Caratteristiche principali:

- **Flessibilità:** Permette di specificare solo i vincoli, lasciando libertà sull'ordine delle attività.
- **Espressività:** Supporta vincoli logici e temporali senza imporre una struttura rigida.
- **Adattabilità:** Consente ai processi di evolversi dinamicamente in base alle esigenze del contesto.

## Differenze tra Modellazione Procedurale e Dichiarativa

Caratteristica	Procedurale	Dichiarativa
<b>Definizione del flusso</b>	Sequenza rigida	Vincoli sulle attività
<b>Flessibilità</b>	Bassa	Alta
<b>Esecuzione</b>	Deterministica	Più adattabile
<b>Supporto per eccezioni</b>	Limitato	Forte

### **Esempio:**

**Procedurale:** “L’attività A deve essere seguita dall’attività B.”

**Dichiarativo:** “L’attività B deve avvenire dopo A, ma senza specificare un ordine rigido con altre attività.”

## Proprietà Formali nei Linguaggi Dichiarativi

### Proprietà Strutturali e di Controllo del Flusso

I linguaggi dichiarativi garantiscono proprietà formali come:

- **Soundness:** Assicura che ogni istanza del processo possa terminare correttamente.
- **Weak Soundness:** Permette percorsi che potrebbero non essere eseguiti in tutte le istanze, ma il processo nel suo insieme è eseguibile.
- **Lazy Soundness:** Alcune attività possono essere eseguite dopo che il processo è formalmente terminato.

### Conformance Checking

Verifica se un’istanza del processo rispetta i vincoli dichiarativi, individua violazioni e identifica deviazioni dai vincoli definiti.

**Esempio:** “Un ordine deve essere spedito solo dopo che è stato confermato dal cliente.”

- **Token Replay:** Simula l’esecuzione del processo e verifica la correttezza.
- **Alignment-Based Checking:** Confronta l’istanza con il modello atteso e misura le deviazioni.



## DECLARE

**DECLARE** è uno dei principali linguaggi dichiarativi per la modellazione dei processi aziendali.

- **Approccio basato su vincoli:** definisce solo ciò che è obbligatorio o proibito.
- **Basato su LTL (Linear Temporal Logic):** utilizza logica temporale per specificare le proprietà dei processi.
- ✓ **Supporta conformance checking** e monitoraggio dei processi.

**Esempio di vincoli in DECLARE:**

**Response(A, B) // Se A accade, allora B deve accadere dopo.**

**Precedence(A, B) // B può accadere solo se A è già accaduto.**

## Applicazioni e Benefici dei Linguaggi Dichiarativi

I linguaggi dichiarativi sono utilizzati in diversi settori:

- ✓ **Sanità:** Gestione di percorsi terapeutici flessibili.
- ✓ **Compliance aziendale:** Garanzia del rispetto delle normative.
- ✓ **Sistemi dinamici:** Adattamento automatico dei processi in base a cambiamenti di contesto.

**in sintesi:**

I linguaggi dichiarativi per BPM offrono una flessibilità superiore rispetto ai linguaggi procedurali, permettendo una modellazione più adattabile e dinamica. DECLARE e altre soluzioni basate su vincoli consentono di specificare **proprietà formali** garantendo la conformità ai requisiti aziendali senza imporre un flusso rigido di esecuzione.

# Process Mining

Il **Process Mining** è una tecnica che permette di **scoprire, monitorare e migliorare i processi aziendali** a partire dai dati contenuti nei **registri degli eventi (event logs)**. Questo approccio è fondamentale per identificare inefficienze, colli di bottiglia e deviazioni dai processi attesi.

## Principali attività del Process Mining:

1. **Scoperta (Discovery)**: Creazione di un modello di processo a partire dai dati senza alcun modello predefinito.
2. **Verifica di conformità (Conformance Checking)**: Confronto tra un modello di riferimento e i dati reali per individuare deviazioni.
3. **Miglioramento (Enhancement)**: Ottimizzazione dei processi aziendali grazie ai dati estratti.

## Componenti Chiave del Process Mining

- **Event Logs**: Contengono informazioni sui processi aziendali, inclusi timestamp, attori e attività.
- **Modelli di Processo**: Strutture formali (ad esempio, **Reti di Petri**) che rappresentano il flusso di attività.
- **Algoritmi di Mining**: Tecniche come  **$\alpha$ -algorithm**, **Heuristic Mining** e **Inductive Mining** per estrarre modelli dai dati.

## Fasi del Process Mining

### Scoperta (Discovery)

L'obiettivo della **scoperta** è costruire un modello basato sui dati raccolti nei log degli eventi. Gli algoritmi analizzano le sequenze di eventi e generano un modello rappresentativo del processo aziendale.

## Verifica di Conformità (Conformance Checking)

Questa fase confronta il modello scoperto con un **modello di riferimento**, per identificare discrepanze tra il comportamento atteso e quello reale.

Metodi:

1. **Token Replay**: Simulazione dell'esecuzione del modello per verificare la correttezza.
2. **Alignment-based Checking**: Determina le deviazioni tra il modello atteso e il log.

## Miglioramento (Enhancement)

L'analisi dei dati permette di:

1. **Identificare inefficienze** (tempi di attesa elevati, colli di bottiglia).
2. **Ottimizzare il flusso di lavoro**.
3. **Migliorare il supporto decisionale** basato su dati empirici.

## Algoritmi di Process Mining

Algoritmo	Caratteristiche
<b><math>\alpha</math>-algorithm</b>	Basato su relazioni di precedenza tra attività
<b>Heuristic Mining</b>	Identifica dipendenze e frequenze tra eventi
<b>Inductive Mining</b>	Crea modelli flessibili con alta precisione

Esempio di scoperta tramite  **$\alpha$ -algorithm**:

1. Identifica la relazione tra eventi.
2. Crea una rete di transizioni.
3. Genera un **modello di processo** con **Reti di Petri**.

## Applicazioni del Process Mining

Il Process Mining viene utilizzato in diversi settori:

- ✓ **Sanità**: Analisi dei percorsi dei pazienti per migliorare la gestione ospedaliera.
- ✓ **Produzione**: Ottimizzazione delle catene di approvvigionamento.
- ✓ **Servizi Finanziari**: Identificazione di frodi attraverso pattern anomali.

## Sfide e Limiti del Process Mining

### 1. Qualità dei Dati

- **Dati mancanti o inconsistenti** nei log possono compromettere l'analisi.
- **Eventi duplicati o errati** rendono difficile la costruzione del modello.

### 2. Complessità Computazionale

- **Modelli troppo dettagliati** possono risultare difficili da interpretare.
- **Processi con alta variabilità** richiedono tecniche avanzate di analisi.

### 3. Sfide aggiuntive

- **Concept Drift**: I processi aziendali possono evolversi nel tempo, rendendo obsoleti i modelli scoperti.
  - **Sudden Drift**: il processo cambia *improvvisamente*
  - **Gradual Drift**: La *transizione* tra due versioni del processo avviene *lentamente*.
  - **Recurring Drift**: Il processo *alterna* tra diverse varianti.
- **Rappresentazione del Modello**: La scelta del giusto livello di dettaglio è fondamentale per ottenere insight utili.

### **in sintesi:**

Il **Process Mining** rappresenta un potente strumento per migliorare l'efficienza aziendale, fornendo **analisi dettagliate sui processi basate sui dati reali**. Tuttavia, la qualità dei dati e la complessità computazionale rimangono sfide critiche. Con l'evoluzione dell'**intelligenza artificiale e del machine learning**, il Process Mining continuerà a crescere, offrendo nuove opportunità per l'ottimizzazione dei processi aziendali.