Alma Mater Studiorum · Università di Bologna

FACOLTÀ DI SCIENZE MATEMATICHE, FISICHE E NATURALI

Corso di Laurea Magistrale in Ingegneria Informatica

Progettazione e sviluppo di un sistema esperto per il *"Menu Planning"*

Tesi di Laurea in Intelligenza Artificiale

Relatore:
Chiar.mo Prof. Andrea Roli
Correlatore:
Egr. Dott. Primo Vercilli

Presentata da:
Davide Dusi

Sessione Terza
Anno Accademico 2010/2011

"Applications programming is a race between software engineers, who strive to produce idiot-proof programs, and the Universe which strives to produce bigger idiots.
So far, the Universe is winning."

- ***Rich Cook***, *The Wizardry Compiled.*

Introduzione

La dieta, nell'antica medicina greca, rappresentava il complesso delle norme di vita, come l'alimentazione, l'attività fisica, il riposo, atte a mantenere lo stato di salute di una persona. Al giorno d'oggi le si attribuisce un significato fortemente legato all'alimentazione, può riferirsi al complesso di cibi che una persona mangia abitualmente oppure, con un messaggio un po più moderno, ad una prescrizione di un regime alimentare da parte di un medico. Ogni essere umano mangia almeno tre volte al giorno, ognuno in base al proprio stile di vita, cultura, età, etc. possiede differenti abitudini alimentari che si ripercuotono sul proprio stato di salute. Inconsciamente tutti tengono traccia degli alimenti mangiati nei giorni precedenti, chi più chi meno, cercando di creare quindi una *pianificazione* di cosa mangiare nei giorni successivi, in modo da variare i pasti o semplicemente perchè si segue un regime alimentare particolare per un certo periodo. Diventa quindi fondamentale tracciare questa pianificazione, in tal modo si può tenere sotto controllo la propria alimentazione, che è in stretta relazione con il proprio stato di salute e stress, e si possono applicare una serie di aggiustamenti dove necessario. Questo è quello che cerca di fare il *"Menu Planning"*, offrire una sorta di guida all'alimentazione, permettendo così di aver sotto controllo tutti gli aspetti legati ad essa. Si pensi, ad esempio, ai prezzi degli alimenti, chiunque vorrebbe minimizzare la spesa, mangiare quello che gli piace senza dover per forza rinunciare a quale piccolo vizio quotidiano. Con le tecniche di *"Menu Planning"* è possibile avere una visione di insieme della propria alimentazione.

La prima formulazione matematica del *"Menu Planning"* (allora chiamato *diet problem*) nacque durante gli anni '40, l'esercito Americano allora impegnano nella Seconda Guerra Mondiale voleva abbassare i costi degli alimenti ai soldati mantenendo però inalterata la loro dieta. George Stingler, economista americano, trovò una soluzione, formulando un problema di ottimizzazione e vincendo il premio Nobel in Economia nel 1982.

Questo elaborato tratta dell'automatizzazione di questo problema e di come esso possa essere risolto con un calcolatore, facendo soprattutto riferimento a particolari tecniche di Intelligenza Artificiale e di rappresentazione della conoscenza, nello specifico il lavoro si è concentrato sulla progettazione e sviluppo di un ***ES*** *case-based* per risolvere il problema del *"Menu Planning"*. Verranno mostrate varie tecniche per la rappresentazione della conoscenza e come esse possano essere utilizzate per fornire supporto ad un programma per elaboratore, partendo dalla Logica Proposizionale e del Primo Ordine, fino ad arrivare ai linguaggi di *Description Logic* e Programmazione Logica. Inoltre si illustrerà come è possibile raccogliere una serie di informazioni mediante procedimenti di *Knowledge Engineering.*
A livello concettuale è stata introdotta un'architettura che mette in comunicazione l'***ES*** e un ***Ontologia*** di alimenti con l'utilizzo di opportuni *framework* di sviluppo. L'idea è quella di offrire all'utente la possibilità di vedere la propria pianificazione settimanale di pasti e dare dei suggerimenti su che cibi possa mangiare durante l'arco della giornata. Si mostreranno quindi le potenzialità di tale architettura e come essa, tramite ***Java***, riesca far interagire ***ES*** *case-based* e ***Ontologia*** degli alimenti.

Organizzazione della tesi

Nel Primo capitolo sono racchiusi tutti i concetti preliminari per la corretta comprensione dell'elaborato, le Logiche fondamentali per rappresentare la conoscenza, che cosa è la *Knowledge Engineering*, vari tecniche per descrivere la conoscenza, linguaggi di Programmazione per interpretarla ed usarla ed infine i sistemi basati su di essa.

Il Secondo capitolo tratta del problema del *"Menu Planning"* nel dettaglio fornendo cenni storici e come le tecniche per risolverlo si siano evolute da problema di ottimizzazione a sistemi di Intelligenza Artificiale.

Nel Terzo capitolo viene mostrata l'analisi dell'architettura affrontata e le problematiche in relazione ai sistemi basati sulla conoscenza.

Infine nel capitolo conclusivo verranno mostrati esempi pratici di come questa architettura possa essere sviluppata con *framework* di sviluppo *opensource*.

Indice

Introduzione i

Organizzazione della tesi iii

1 *"Can a machine think?"* 1

1.1 La Logica . . . 2
1.2 Logica Proposizionale . . . 3
1.2.1 Sintassi . . . 3
1.2.2 Semantica . . . 5
1.2.2.1 Implicazione . . . 6
1.2.2.2 Modelli . . . 6
1.2.3 Inferenza . . . 7
1.2.4 Validità . . . 8
1.2.5 Soddisfacibilità . . . 8
1.2.6 Modus Pones . . . 9
1.2.7 Algoritmi di concatenazione . . . 9
1.2.7.1 Concatenazione in avanti . . . 10
1.2.7.2 Concatenazione all'indietro . . . 12
1.3 Risoluzione . . . 13
1.3.0.3 Forma Normale Congiuntiva . . . 13
1.4 Logica del Primo Ordine (FOL) . . . 14
1.4.1 Sintassi e Semantica . . . 15
1.4.1.1 Modelli . . . 17
1.4.1.2 Quantificatori . . . 18

1.4.2 Modus Pones Generalizzato 20
1.4.3 Algoritmi di concatenazione in ***FOL*** 22
1.4.3.1 Forward Chaining 22
1.4.3.2 Backward Chaining 24
1.4.4 Risoluzione in ***FOL*** 25
1.5 Knowledge Engineering 27
1.6 Rappresentazione della Conoscenza 28
1.6.1 Ontologia 29
1.6.2 Description Logic 31
1.6.3 OWL 31
1.7 Programmazione Logica 35
1.7.1 Prolog 38
1.8 Sistemi Basati sulla Conoscenza 40
1.8.1 I Sistemi Esperti 41
1.8.2 Controllo 45

2 Approccio al problema del *“Menu Planning”* **47**
2.1 Computer e *“Menu Planning”* 50
2.2 Il *Common Sense* per il *“Menu Planning”* 52

3 Progettazione di un Sistema Esperto per il *“Menu Planning”* **55**
3.1 Dominio e Attori 56
3.2 Requisiti del Sistema 56
3.3 Prima Analisi e Struttura del Sistema 57
3.4 Risoluzione del problema del *Common Sense* 59
3.5 Expertise Modeling 61
3.5.1 Ontologia del Dominio 61
3.5.2 Knowledge Base 62
3.5.3 Ontologia degli Alimenti 64
3.6 Meta-Engine 65

4 Sviluppo del Sistema Esperto **69**
4.1 Sviluppo della Conoscenza 70
4.1.1 La Knowledge Base e il Motore Inferenziale 70
4.1.2 Sviluppo dell'Ontologia degli Alimenti 72
4.2 Sviluppo del Meta-Engine 73
4.2.1 Generazione di un menú 74
4.2.2 ***Menu Sharpening*** 74
4.2.3 ***Menu Filling*** 76
4.2.4 Utilizzo della Piramide Alimentare 77

Conclusioni **81**

A Appendice **83**
A.1 Resolution Propositional Logic 83
A.2 Backward Chaining ***FOL*** 84
A.3 Algoritmo di Ricerca Completo 84
A.4 Euristica 84
A.5 Herbert Simon 84
A.6 Bottle-Neck 85
A.7 Fuzzy 85
A.8 Licenza LGPL 85
A.9 Web Semantico 85

Bibliografia **87**

Elenco delle figure

1.1 Albero di concatenazione [12] 11
1.2 Pseudocodice della concatenazione in avanti. 23
1.3 Albero di dimostrazione forward. 23
1.4 Albero di dimostrazione backward 25
1.5 Punti di vista differenti per uno *scambiatore di calore* 30
1.6 Costruttori OWL 33
1.7 Assiomi OWL 33
1.8 Architettura di un sistema basato sulla rappresentazione della conoscenza in ***Description Logic*** [18]. 34
1.9 Architettura di un sistema esperto.[14] 43

2.1 Gerarchida di database di PRISM.[14] 53

3.1 Piramide alimentare. 57
3.2 Diagramma delle attività. 59
3.3 Architettura del sistema. 60
3.4 Ontologia del dominio. 63
3.5 Ontologia degli alimenti. 64
3.6 Analisi del *Meta-Engine*. 65

4.1 Progettazione del *Meta-Engine*. 69

A.1 Pseudocodice dell'algoritmo di risoluzione. 83
A.2 Pseudocodice della concatenazione all'indietro. 84

Capitolo 1

“Can a machine think?”

Fin dall’antichità l’uomo si è sempre interrogato su come funzionasse la propria mente, i filosofi hanno cercato di dare risposte su cosa sia l’intelletto e il pensiero umano che tutt’oggi sono ancora rimaste irrisolte. Ma ammettendo anche che si riescano a carpire tutti i processi cognitivi che si mettono in atto durante un pensiero, poi si potranno riprodurre tali processi? E lo si potrà poi incorporare in una macchina? O tale abilità è e rimarrà una prerogativa dell’essere umano?

Intelligenza: *Complesso di facoltà psichiche e mentali che consentono all’uomo di pensare, . . . e lo rendono insieme capace di adattarsi a situazioni nuove e di modificare la situazione stessa quando questa presenta ostacoli all’adattamento.*

(Treccani, Ist. Enciclopedia Italiana, 2010 [1])

Dalla definizione si può facilmente intuire che le facoltà che compongono l’intelligenza sono prettamente umane, queste rappresentano le risorse della nostra psiche e vengono utilizzate durante l’attività del pensiero.

Pensare: *Esercitare l’attività del pensiero, cioè l’attività psichica per cui l’uomo acquista coscienza di sé e del mondo in cui vive.*

(Treccani, Ist. Enciclopedia Italiana, 2010[1])

[1]http://www.treccani.it/

Il principio fondamentale su cui si basava la filosofia di Cartesio era *cogito, ergo sum* (cit. dal latino - *penso, dunque sono*, tradur. dell'espressione). Quello che cerca di fare l'Intelligenza Artificiale (*Artificiale Intelligence - AI*) è riprodurre, se pur parzialmente le attività intellettuali proprie dell'uomo, in particolare tutti i processi di apprendimento, di risoluzione di problemi e di scelta. Una macchina, o un programma, viene ritenuta intelligente quando riesce a riprodurre o imitare i comportamenti meno complessi ritenuti peculiari dell'intelligenza umana[1], in particolare solitamente sono in grado di risolvere piccoli problemi *domain specific*, come ad esempio giocare a scacchi; si distinguono dai canonici programmi in quanto son in grado di dedurre e, in alcuni casi, possono anche non eseguire nessuna tipologia di calcolo.
Quindi diventa importante capire quando una macchina può essere ritenuta intelligente. A questo proposito Alan Turing (1912 - 1954) inventò il cosiddetto *"Turing imitation game"*. Il gioco è volto a capire se una macchina possa passare un test di comportamento intelligente, la metodologia proposta da Turing si articola in due fasi. Per prima cosa un'interlocutore, un uomo e una donna vengono divisi in tre stanze separate, ognuno di essi dispone di un terminale con cui possono comunicare, lo scopo dell'interlocutore è capire chi è l'uomo e chi la donna ponendo domande di qualsiasi entità, ovviamente ognuno di essi cercherà di sviarlo sulla propria reale identità. Nella seconda fase l'uomo viene sostituito con un computer programmato a illudere l'interlocutore come faceva la persona nella fase precedente. Se il computer riesce nel proprio intento allora avrá passato il test. Ovviamente il computer può essere programmato per fare errori o rispondere in logica fuzzy, l'interlocutore inoltre può porre qualsiasi tipo di domanda, perfino calcoli matematici molto complessi per misurare il tempo di calcolo.

1.1 La Logica

La logica è lo studio dei principi alla base del corretto ragionamento. Grazie ai linguaggi logici è possibile rappresentare informazioni e, tramite

apposite regole, effettuare ragionamenti e trarre conclusioni in particolari contesti. Un linguaggio logico è formalmente composto da una *sintassi*, con la quale è possibile definire delle frasi, e da una *semantica*, ovvero il significa astratto che gli si attribuisce.

1.2 Logica Proposizionale

1.2.1 Sintassi

La logica proposizionale[2] è la più semplice delle logiche, la sua sintassi presenta due elementi principali:

- *Simbolo Proposizionale* (A,B,C,P,Q,R, ...)
- *Connettivo Logico* ($\neg, \wedge, \vee, \Rightarrow, \Leftrightarrow$)

Questi due elementi basilari permettono di comporre formule, dette *Atomiche* se composte da un solo simbolo proposizionale o dette *Complesse* se mettono insieme simboli proposizionali e connettivi logici.
Un simbolo identifica una proposizione e può assumere due valori quello di verità (*true*) o quello di non verità (*false*). Si possono usare differenti nomenclature per i simboli che variano a seconda del contesto in cui si usa la logica, per una più facile comprensione della loro semantica, esistono comunque due simboli proposizionali che rimangono invariati di significato, il simbolo *True* che rappresenta una proposizione sempre vera e il simbolo *False* che al contrario indica una proposizione sempre falsa. Per formalità i simboli vengono rappresentati con una lettera maiuscola (A,B,C,P, ...), mentre i valori che questi simboli possono assumere vengono scritti con la prima lettera minuscola, come ad esempio *true* e *false*.

[2]Conosciuta anche come logica di **Boole** o *zeroth-order logic.*

I connettivi logici permetto la costruzione di formule complesse, ne esistono in particolare cinque di uso comune:

- **Negazione** $\neg$ *(not)*: dichiara l'affermazione negativa di un simbolo. Per esempio $\neg$A è la negazione di A.

- **Congiunzione** $\wedge$ *(and)*: congiunge due simboli, come A $\wedge$ B (A e B sono detti *congiunti*).

- **Disgiunzione** $\vee$ *(or)*: disgiunge due simboli, come C $\vee$ D (C e D sono detti *disgiunti*).

- **Implicazione** $\Rightarrow$: (A $\wedge$ B) $\Rightarrow$ C è composta da due parti, la parte a sinistra detta *premessa* (A $\wedge$ B) e la parte a destra detta *conclusione* (C), chiamate anche regole *if-then.*

- **Equivalenza** $\Leftrightarrow$: chiamate anche regola *bicondizionale*, A $\Leftrightarrow$ B può essere letta come *..if and only if...*

La concatenazione di questi simboli nelle formule complesse può comportare ambiguità su quale operazione prevale su un'altra, come in matematica esiste la priorità degli operatori, lo stesso principio è applicabile ai connettivi logici, che sono stati presentati in ordine di priorità dal maggiore al minore nell'elenco precedente. Quindi se per esempio abbiamo una formula del tipo $A \wedge B \vee \neg C$, deve essere letta come $(A \wedge B) \vee (\neg C)$.
La priorità degli operatori non risolve completamente il problema delle ambiguità, di conseguenza in formule che hanno connettivi logici di egual priorità diventa necessario l'uso delle parentesi, come ad esempio $A \vee B \vee C$ che può essere letta come $(A \vee B) \vee C$ oppure $A \vee (B \vee C)$.
Inoltre si utilizzano lettere greche $(\alpha, \beta, \ldots)$ per fare rifermento espressioni arbitrarie in logica proposizionale, di per se non rappresentano simboli, ma vengono utilizzate per definire un meta-linguaggio che esprime formule generiche. Un'espressione del tipo $\neg\alpha$, il valore di α non è detto che sia una singolo simbolo proposizionale, come ad esempio A, ma può anche riferirsi

ad una formula più complessa come $(D \wedge E) \vee (\neg E)$.
Una formula perché possa essere interpretata logicamente deve essere *ben formata*[3], che in logica proposizionale è possibile definire con il seguente ragionamento ricorsivo:

1. Qualsiasi simbolo proposizionale è una *wff*.
2. Se α è una *wff*, allora anche $\neg\alpha$ è una *wff*.
3. Se α e β sono *wff*, allora anche $\alpha \wedge \beta$ è una *wff*.
4. Se α e β sono *wff*, allora anche $\alpha \vee \beta$ è una *wff*.
5. Se α e β sono *wff*, allora anche $\alpha \Rightarrow \beta$ è una *wff*.
6. Se α e β sono *wff*, allora anche $\alpha \Leftrightarrow \beta$ è una *wff*.
7. Tutto ciò che può essere costruito ricorsivamente con i punti (1) e (6) è una *wff*.

Questo ragionamento può essere semplicemente interpretato come la costruzione di frasi corrette nel linguaggio comune, il buon senso ci dice che la parola *cane* debba essere scritta così perché possa essere interpretata, lo stesso ragionamento lo si deve applicare alle formule proposizionali. Ad esempio è facilmente intuibile che la formula $(A \wedge B) \vee (\neg C)$ sia *wff*, mentre non lo è se viene scritta $(AB\wedge)\vee)C\neg($.

1.2.2 Semantica

Come precedentemente accennato una logica non è composta solamente dalla sintassi, bisogna anche attribuire un significato a ciò che si vuole rappresentare ed è questo che si intende per *semantica*.
Il modello della logica proposizionale presenta solamente due valori che possono essere assunti da un simbolo, ovvero *true* oppure *false*. Di conseguenza

[3] *well-formed formula*, d'ora in poi abbreviata con *wff*

il valore, verità o falsità, di formule più complesse viene determinato dal valore che i singoli simboli hanno. Come per esempio:

modello: $A = true, B = true, C = false$
data la formula: $(A \wedge B) \vee (\neg C)$
$(true \wedge true) \vee (\neg false) = (true \wedge true) \vee (true) = true \vee true = true$

Di seguito una tabella di verità dei connettivi proposizionali:

A	B	$\neg A$	$A \wedge B$	$A \vee B$	$A \Rightarrow B$	$A \Leftrightarrow B$
false	false	true	false	false	true	true
false	true	true	false	true	true	false
true	false	false	false	true	false	false
true	true	false	true	true	true	true

1.2.2.1 Implicazione

L'implicazione in logica (rappresentata dal simbolo $\models$) è una relazione che coinvolge un gruppo di frasi ben formate (che possiamo definire *Knowledge Base*[4]) e una singola frase. Quindi data una KB, questa *implica logicamente* α se e solo se ogni interpretazione di KB soddisfa anche α.
Per esempio prendiamo la frase γ che *implica logicamente* $(\gamma \vee \delta)$, per la semantica della *disgiunzione* basta che o γ o δ siano veri per dire che $\gamma \vee \delta$ è vera, di conseguenza $\gamma \vee \delta$ è vera dove γ è vera. Al contrario se si prende $\gamma \models (\gamma \wedge \delta)$, per la semantica della congiunzione sia γ che δ devono essere vere, quindi un qualsiasi gruppo di frasi che contenga γ o δ non implica che $(\gamma \wedge \delta)$ sia vero.

1.2.2.2 Modelli

Nel calcolo proposizionale un *modello* è un assegnamento dei valori *true* o *false* ad ogni simbolo proposizionale. Quindi si dice che m è un modello di α se α è vero in m. Può rivelarsi necessario dover enumerare più modelli, si

[4] d'ora in poi abbreviata con KB

definisce quindi $\mathbf{M}(\alpha)$ un gruppo di modelli di α.
Riprendendo la definizione di *implicazione* si può dire che KB $\models \alpha$ se e solo se $\mathbf{M}(\alpha)$ è un sottoinsieme dei modelli di KB, ovvero $\mathbf{M}(\mathrm{KB}) \subseteq \mathbf{M}(\alpha)$.
Fortemente legato ai concetti di *implicazione* e *modello* è *l'equivalenza logica*, ovvero quando due formule α *e* β sono vere nello stesso insieme di modelli. Questo si può rappresentare come nel seguente esempio, date due formule α *e* β:

$$\alpha \equiv \beta \textit{ if and only if } \alpha \models \beta \textit{ and } \beta \models \alpha$$

Con questo concetto è possibile definire una serie di proprietà degli operatori logici proposizionali, riassunte nella tabella sottostante:

$(\alpha \wedge \beta)$	$\equiv$	$\beta \ \wedge \alpha$	commutativity of $\wedge$
$(\alpha \vee \beta)$	$\equiv$	$\beta \ \vee \ \alpha$	commutativity of $\vee$
$((\alpha \wedge \beta) \wedge \gamma)$	$\equiv$	$(\alpha \wedge (\beta \wedge \gamma))$	associativity of $\wedge$
$((\alpha \vee \beta) \vee \gamma)$	$\equiv$	$(\alpha \vee (\beta \vee \gamma))$	associativity of $\vee$
$\neg(\neg\alpha)$	$\equiv$	α	double-negation elimination
$(\alpha \Rightarrow \beta)$	$\equiv$	$(\neg\beta \Rightarrow \neg\alpha)$	contraposition
$(\alpha \Rightarrow \beta)$	$\equiv$	$(\neg\alpha \vee \beta)$	implication elimination
$(\alpha \Leftrightarrow \beta)$	$\equiv$	$((\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha))$	biconditional elimination
$\neg(\alpha \wedge \beta)$	$\equiv$	$(\neg\alpha \vee \neg\beta)$	De Morgan
$\neg(\alpha \vee \beta)$	$\equiv$	$(\neg\alpha \wedge \neg\beta)$	De Morgan
$(\alpha \wedge (\beta \vee \gamma))$	$\equiv$	$(\alpha \wedge \beta) \vee (\alpha \wedge \gamma))$	distributivity of $\wedge$ *over* $\vee$
$(\alpha \vee (\beta \wedge \gamma))$	$\equiv$	$(\alpha \vee \beta) \wedge (\alpha \vee \gamma))$	distributivity of $\vee$ *over* $\wedge$

1.2.3 Inferenza

L'*inferenza* logica (rappresentata dal simbolo $\vdash$) è un processo con il quale si possono trarre delle conclusioni da frasi della grammatica. L'esempio più utilizzato è quello riferito al famoso filosofo greco Socrate:

1. Tutti gli uomini sono mortali.

2. Socrate è un uomo.

3. *Quindi* Socrate è un mortale.

È di conseguenza possibile definire una procedura i che derivi la frase "Socrate è mortale" dalla KB contente {"Tutti gli uomini sono mortali", "Socrate è un uomo"}.
In generale si dice che data l'inferenza KB $\vdash_i \alpha$, è possibile derivare α da KB attraverso la procedura i. Questa può avere due differenti proprietà:

- **Solidità**[5]: Si dice che la procedura i è *solida* ogni qual volta prova formule valide rispetto all'implicazione, ovvero se abbiamo KB $\vdash_i \alpha$ allora è anche vero che KB $\models \alpha$.
- **Completezza**[6]: Al contrario, si dice che ogni frase α che è conseguenza di KB può essere derivata dal KB, ovvero ogni qual volta KB $\models \alpha$, è anche vero che KB $\vdash_i \alpha$.

1.2.4 Validità

Un altro concetto molto importante è quello delle *tautologie*. Queste formule *wff* sono anche chiamate *le verità della logica*, sono formule vere per **tutti** i modelli. Per esempio:

$$A \vee \neg A$$

Per ogni assegnamento di A la precedente formula sarà sempre vera infatti, per la semantica della disgiunzione, A o la sua negazione saranno necessariamente vere in ogni modello.

1.2.5 Soddisfacibilità

È raro però che una formula sia vera in tutti i modelli per questo si utilizza un concetto meno vincolante della validità. Quando una formula è vera in **qualche** modello allora questa è *soddisfacibile*, per questo motivo quando m è un modello di α si dice che m *soddisfa* α.

[5] Soundness.
[6] Completeness.

1.2.6 Modus Pones

È possibile applicare a formule della logica proposizionale *regole di inferenza*, con queste si vengono a creare catene di ragionamento con le quali si possono raggiungere particolari obiettivi di dimostrazione. In logica per regole di inferenza si intendono tutti quei *template*[7] che portano alla costruzione di formule valide.
Il più noto tra le regole di inferenza è il ***Modus Ponens***:

$$\frac{\alpha \ \Rightarrow \ \beta, \ \alpha}{\beta}$$

In sostanza significa che se si dispone della formula α e $\alpha \Rightarrow \beta$ *allora* si può dedurre la formula β.
Prendiamo ad esempio una KB = {A,B,C,($A \wedge B \ \ \Rightarrow \ D$)}, la *premessa* dell'implicazione $A \wedge B \ \ \Rightarrow \ D$ è vera in quanto la KB contiene $A \wedge B$, applicando il ***Modus Ponens*** possiamo dimostrare che anche D è vera nella KB.

1.2.7 Algoritmi di concatenazione

Gli algoritmi di concatenazione utilizzano le regole di inferenza per dimostrare gli obiettivi desiderati.
Solitamente non è necessario utilizzare tutta la potenza espressiva della sintassi logica, per questo nella maggior parte dei casi pratici le KB sono composte dalle cosiddette ***clausole di Horn***. Una clausola di Horn è una *congiunzione* di una serie di *implicazioni* o di *simboli proposizionali.* Un esempio molto semplice può essere: $(A \Rightarrow B) \wedge C$, è possibile quindi ridefinire il ***Modus Ponens*** per le clausole di Horn come:

$$\frac{\alpha_1, \ldots, \alpha_n \quad \alpha_1 \wedge \ldots \wedge \alpha_n \Rightarrow \ \beta}{\beta}$$

[7]traduzione letterale: sagoma.

Esistono due principali algoritmi di concatenazione, la ***Concatenazione in avanti***[8] e la ***Concatenazione all'indietro***[9], sono entrambi due algoritmi molto semplici che si risolvono in tempo lineare.

1.2.7.1 Concatenazione in avanti

L'idea della concatenazione in avanti è soddisfare le premesse per raggiungere un determinato obiettivo. Quando si decide di applicare una regola di implicazione, innanzi tutto questa deve aver *premessa* vera (fatti noti) nella KB in esame, poi si aggiungerà la sua *conclusione* alla KB. Riprendendo l'esempio precedente:

$$\text{KB} = \{\text{A,B,C,}(A \wedge B \ \Rightarrow D)\}$$

La premessa $A \wedge B$ è vera nella KB, quindi possiamo applicare il ***Modus Ponens*** per dedurre D. Di conseguenza la KB diverrà:

$$\text{KB} = \{\text{A,B,C,D,}(A \wedge B \ \Rightarrow D)\}$$

Questo processo continua fino a che non si raggiunge l'obiettivo desiderato (il Goal è nella KB) oppure risulta impossibile applicare ulteriori regole, viene solitamente rappresentato con un grafo (Figura[1.1]).

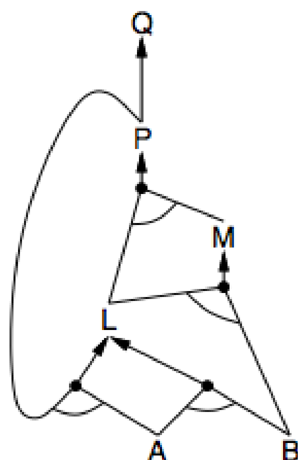
Analizzando il grafico è possibile intuire che si vuole dimostrare il simbolo Q e che inizialmente si ha a disposizione i simboli A e B ed una serie di regole di inferenza, di seguito la KB relativa:

$$P \Rightarrow Q$$
$$L \wedge M \Rightarrow P$$
$$B \wedge L \Rightarrow M$$
$$A \wedge P \Rightarrow L$$
$$A \wedge B \Rightarrow L$$
$$\text{A}$$
$$\text{B}$$
$$Goal : Q$$

[8] Forward Chaining.

[9] Backward Chaining.

Figura 1.1: Albero di concatenazione [12]



Si prenda ora, per esempio, la regola $A \wedge B \Rightarrow L$, la sua premessa $A \wedge B$ è vera nella KB, quindi è possibile applicare tale regola e la KB diventerà:

$$P \Rightarrow Q$$
$$L \wedge M \Rightarrow P$$
$$B \wedge L \Rightarrow M$$
$$A \wedge P \Rightarrow L$$
$$A \wedge B \Rightarrow L$$
$$\text{A}$$
$$\text{B}$$
$$\text{L}$$
$$Goal : Q$$

Questo tipo di ragionamento è anche denominato *data-driven*, è sicuramente quello più spontaneo da applicare ma in alcuni casi può comportare molte esecuzioni che non fanno avvicinare al Goal finale.

1.2.7.2 Concatenazione all'indietro

Un altro tipo di approccio è dato dall'algoritmo di concatenazione all'indietro. In questo caso si lavora con un ragionamento inverso, ovvero dato il Goal finale si cerca di provare che le sue premesse sono vere nella KB attraverso una regola di inferenza. Questo approccio permette di evitare l'affollarsi di inutili conseguenze nella KB che si potrebbe verificare nel caso in cui si utilizzi una concatenazione in avanti. Riprendendo l'esempio precedente, partendo dal Goal finale Q si cerca quindi di dimostrare che la sua premessa P è vera (regola $P \Rightarrow Q$). In questo caso P non è presente nella KB come singolo simbolo, però esiste una premessa per la quale P è vero cioè $L \wedge M$ (regola $L \wedge M \Rightarrow P$), si procede quindi cercando di dimostrare le premesse per L e M. Il ragionamento termina quando tutte le premesse sono state dimostrate, nell'esempio L viene constatato dai fatti A e B che non hanno alcuna premessa, rimane quindi da dimostrare M per il quale si utilizzerà la regola ($B \wedge L \Rightarrow M$), con la quale si avrà il fatto B e L dimostrato con il percorso precedente.
Nella KB sono presenti due regole con cui si può dimostrare l'esistenza di L ovvero $A \wedge P \Rightarrow L$ e $A \wedge B \Rightarrow L$, in questo caso non esiste una scelta precisa con la quale decidere quale tra le due applicare[10], solitamente si scandiscono le regole in ordine, ma può anche essere previsto l'utilizzo di un meta-algoritmo.

Concludendo si possono utilizzare due approcci uno *data-driven*[11] e uno *goal-driven*[12], utilizzando il primo algoritmo è possibile che vengano dimostrati un numero considerevoli di fatti inutili per il raggiungimento del Goal, al contrario l'algoritmo di *backward chaining* risulta essere molto appropriato per il *problem solving* e la sua complessità è molto inferiore a quella lineare nelle dimensioni della KB.

[10]è triviale che nell'esempio si è scelto il percorso che porta alla terminazione dell'algoritmo

[11]concatenazione in avanti

[12]concatenazione all'indietro

1.3 Risoluzione

Il ***Modus Ponens*** è una regola di inferenza corretta e completa finchè si utilizzano le clausole di *Horn*, gli algoritmi di concatenazione sono abbastanza efficienti e utili in molti casi rimangono però inadeguati in alcune circostanze. Per questi motivi si usa la regola di inferenza di *risoluzione* che se unito a qualsiasi algoritmo di ricerca completo dà luogo ad un *algoritmo di inferenza completo.*

1.3.0.3 Forma Normale Congiuntiva

La regola di *risoluzione* però si applica solamente alle disgiunzioni di formule atomiche. Ogni formula della logica proposizionale è logicamente equivalente a una congiunzione di disgiunzioni di letterali, che viene denominata **forma normale congiuntiva**[13]. Si può ottenere una ***CNF*** con le seguenti regole:

1. si elimina $\Leftrightarrow$, si rimpiazzando $\alpha \Leftrightarrow \beta$ con $(\alpha \Rightarrow \beta) \wedge (\alpha \Rightarrow \beta)$
2. si elimina $\Rightarrow$, rimpiazzando $\alpha \Rightarrow \beta$ con $\neg\alpha \vee \beta$
3. il $\neg$ deve essere applicato solo ai letterali quindi si utilizzano le seguenti equivalenze
   - $\neg(\neg\alpha) \equiv \alpha$, si elimina la doppia negazione
   - $\neg(\alpha \wedge \beta) \equiv (\neg\alpha \wedge \neg\beta)$, De Morgan
   - $\neg(\alpha \vee \beta) \equiv (\neg\alpha \vee \neg\beta)$, De Morgan

Una volta ottenuta la formula ***CNF*** è possibile applicare la formula di *risoluzione unitaria*:

$$\frac{l_1 \vee \ldots \vee l_k, \quad m_1 \vee \ldots \vee m_n}{l_1 \vee \ldots \vee l_{i-1} \vee l_{i+1} \vee \ldots \vee l_k \vee m_1 \vee \ldots \vee m_{j-1} \vee m_{j+1} \vee \ldots \vee m_n}$$

[13] in inglese *Conjunctive Normal Form*, d'ora in poi CNF

Gli algoritmi di inferenza basati su questa formula sfruttano il principio di dimostrazione per assurdo, significa che per dimostrare $KB \models \alpha$, si cerca l'insoddisfacibilità di $KB \models \neg\alpha$. L'algoritmo di risoluzione è completo per il *teorema di risoluzione ground*: "Se un insieme di clausole è insoddisfacibile la sua chiusura di della risoluzione contiene la clausola vuota."[12]. La *chiusura della risoluzione* di un insieme di clausole è l'insieme di tutte le clausole derivabili dall'applicazione ripetuta della regola di risoluzione. Per concludere questo algoritmo risulta essere *corretto* e *completo per refutazione*, ovvero che può dire solo se una determinata formula α è o non è implicata nella KB.

La logica proposizionale vista finora offre un punto di partenza per la costruzione di una base di conoscenza, possiede numerosi vantaggi a partire dal fatto che il suo significato è indipendente dal contesto in cui la si usa[14] offrendo così grade scalabilità, è inoltre dichiarativa ovvero un simbolo o una formula si riferiscono ad un'istruzione o fatto. In aggiunta a queste due caratteristiche essa viene definita *compositiva*, ovvero il significato di una formula è derivato dal significato dei singoli simboli all'interno di essa, per esempio data $A \wedge B$ se e A e B hanno valore *true* allora anche la formula avrà il medesimo significato. Il grosso limite però imposto da questa logica è la sua capacità espressiva, che non è sufficiente se si vuole definire una KB di uso comune.

1.4 Logica del Primo Ordine (FOL)

Per sopperire alla bassa capacità espressiva della logica proposizionale è stata introdotta la ***Logica del Primo Ordine***[15], che riprende tutti gli aspetti positivi della logica proposizionale, come la semantica dichiarativa, compositiva, *contex-indipendent* e non ambigua, aumentandone però l'espressività. Contrariamente alla logica precedente dove tutto era rappresentato

[14] context indipendent.

[15] d'ora in poi abbreviata in ***FOL***.

da *fatti*, la ***FOL*** cerca di avvicinarsi al linguaggio di uso comune utilizzando tre elementi base:

- **Oggetti**: come persone, cani, numeri, Rich Cook, colori . . .
- **Relazioni**: tra oggetti, se unitarie vengono anche chiamate *proprietà* (e.g. rosso, giallo, tondo, grande . . .). Ma generalmente sono *n-arie* come più grande di, più alto di, ha colore, . . .
- **Funzioni**: +, un in più di, padre di . . .

Prendiamo, per esempio, la frase "due più due fa tre" può essere scomposta come:

due		due		tre	*Oggetti*
	più				*Funzione*
			fa		*Relazione*

La potenza di questa logica risiede proprio nel fatto che la maggior parte delle attività comuni possono essere pensate come oggetti collegati da relazioni, permettendo così di esprimere in modo molto più naturale idee, pensieri e concetti che con la logica proposizionale non potrebbero essere rappresentati (o sarebbe più complicato farlo).

1.4.1 Sintassi e Semantica

Si prenda ora in considerazione la sintassi del linguaggio che è composta sostanzialmente dai seguenti simboli e che riprende i concetti di oggetto, relazione e funzione:

- **Costanti**: Marco, 2, Luigi, C, . . .
- **Predicati**: Fratello, >, =, . . .
- **Funzioni**: Sqrt, CapelliNeriDi, . . .
- **Variabili**: x, y, a, b, . . .

- **Connettivi**: $\wedge, \vee, \neg, \Rightarrow, \Leftrightarrow$
- **Quantificatori**: $\forall, \exists$

Per i simboli di funzione e predicato solitamente si deve definire il numero di parametri, questi vengono definiti come *arità* del simbolo di funzione o di predicato, per esempio *padre(X, Y)* è un simbolo con arità due.
Infine per determinare se una formula è vera, queste devono essere messe in relazione con i modelli attraverso la semantica. Sostanzialmente si necessita di un'*interpretazione* che aiuti a specificare a quali oggetti, relazione e funzioni fanno riferimento i vari simboli.

Le frasi della ***FOL*** sono composte da due elementi principali, formule atomiche e termini. I *termini* sono un'espressione logica che si riferisce ad un oggetto, quindi un simbolo di costante è un termine. Capita spesso però che non sia necessario assegnare un simbolo ad ogni oggetto, per esempio quando parliamo dei “capelli neri di Marco”, non è necessario assegnare un particolare nome ai suoi capelli, basta infatti utilizzare i simboli di funzione che sostituisce un ipotetico simbolo di costante poco significativo, quindi potremmo dire *CapelliNeriDi(Marco)*. Come detto in precedenza i simboli funzione non hanno un singolo argomento (arità uno), ma possono avere una lista di oggetti che vengono messi in relazione tra loro (arità n).

$$\textbf{Termine} = \left| \begin{array}{l} funzione(termine_1, \ldots, termine_2) \\ \vee\ costante \\ \vee\ variabile \end{array} \right.$$

Il secondo importante elemento da definire sono le *formule atomiche*, queste permettono di mettere in relazione due oggetti per asserire un fatto. Una formula è composta da un simbolo di predicato seguito da una lista di termini tra parentesi (tanti quanti ne dispone la sua arità), per esempio *Fratello(Marco,Luigi)* è una formula che afferma, in base ad una corretta interpretazione, che Marco è fratello di Luigi. Possono anche contenere termini

complessi, come ad esempio *Sposato(Padre(Marco), Madre(Anna))*.

*Una formula atomica è **vera** in un dato modello, sotto una determinata interpretazione, se la relazione a cui fa riferimento il simbolo di predicato è verificata tra gli oggetti a cui fanno riferimento gli argomenti*[12].

Quindi una formula atomica precedente è vera se *Padre(Marco)* e *Madre(Anna)* sono verificati e di conseguenza *Marco* ed *Anna* devono essere oggetti del dominio.

$$\textbf{Formula atomica} = \left|\begin{array}{l} predicato(termine_1, \ldots, termine_2) \\ \lor\ termine_1 = termine_2 \end{array}\right.$$

Infine come nella logica proposizionale i connettivi logici possono essere utilizzati per produrre *formule complesse*. Il significato che gli viene attribuito è identico in ***FOL***, per esempio $\neg Fratello(Marco, Luigi)$ determina il fatto che "Marco **non** è fratello di Luigi", oppure $Fratello(Marco, Luigi) \Rightarrow Fratello(Luigi, Marco)$.

$$\neg S \quad S_1 \land S_2 \quad S_1 \lor S_2 \quad S_1 \Rightarrow S_2 \quad S_1 \Leftrightarrow S_2$$

$$\underbrace{\underbrace{\underbrace{Fratello}_{Predicato}(\underbrace{Marco}_{Termine}, \underbrace{Luigi}_{Termine})}_{Formula\ atomica} \Rightarrow \underbrace{\underbrace{Fratello}_{Predicato}(\underbrace{Luigi}_{Termine}, \underbrace{Marco}_{Termine})}_{Formula\ atomica}}_{Formula\ complessa}$$

ll valore di verità di una formula complessa in un modello è determinato dal valore di verità delle sue sotto formule, come avviene nella logica proposizionale.

1.4.1.1 Modelli

Nella ***FOL*** i modelli assumono un significa più marcato, quí sono composti da due elementi, dominio e interpretazione. Per prima cosa una frase della grammatica si riferisce ad un particolare *dominio* ovvero un **insieme**

di oggetti non vuoto, finito o infinito, gli oggetti al suo interno vengono, appunto, chiamati *elementi del dominio* che possono essere messi in relazione tra di loro. L'*interpretazione* invece assegna ad ogni simbolo un significato nel dominio:

Costante	un elemento del dominio
Predicato	una relazione nel dominio (con appropriata arità)
Funzione	una funzione nel dominio (con appropriata arità)

In generale data un'interpretazione I di una formula α, I viene detto *modello* di α se α è vero in I. Definiti questi due concetti si può ridefinire anche la soddisfacibilità e la validità visti nella logica proposizionale.
Una formula α si dice *soddisfacibile* se è vera in almeno una interpretazione, esiste quindi almeno un modello per α.
Invece una formula α si definisce *tautologia* se è vera in tutte le interpretazioni, cioè tutte le interpretazione sono modelli di α.

1.4.1.2 Quantificatori

Con questa tipologia di logica può risultare necessario esprimere caratteristiche per insiemi di oggetti, senza dover specificarli uno ad uno[16]. Per questo sono stati introdotti i *quantificatori*, in particolare si fa uso del *quantificatore universale* ($\forall$) e del *quantificatore esistenziale* ($\exists$).
Con il *quantificatore universale* si possono esprimere regole generali, per esempio "tutti i fratelli sono persone" oppure "tutti i cani sono animali", si prenda la prima fase che rimane nel dominio della parentela usato in tutti gli esempi, in logica ***FOL*** la si può quindi scrivere come:

$$\forall x\ Fratello(x) \Rightarrow Persona(x)$$

In particolare la formula deve essere letta "*per ogni* x, se x è una fratello, allora x è una persona". Come detto in precedenza la lettera x definisce una *variabile*, che per convenzione viene scritta in minuscolo. Le variabili possono

[16]nota: maggiore potenza espressiva rispetto alla logica proposizionale.

anche essere termini, di conseguenza sono utilizzate anche come argomenti di funzione, un termine che non ha alcuna variabile viene definito **termine ground**.
Un errore molto comune legato a questo quantificatore è l'utilizzo del connettivo $\wedge$ invece di $\Rightarrow$. Questo comporta un'errata interpretazione delle frasi, prendiamo per esempio la frase sottostante:

$$\forall x \; Fratello(x, Luigi) \Rightarrow Alto(x)$$

Questa frase si legge "tutti i fratello di Luigi sono alti" quindi:

$$\begin{gathered} Fratello(Marco, Luigi) \Rightarrow Alto(Marco) \\ \wedge \; Fratello(Andrea, Luigi) \Rightarrow Alto(Andrea) \\ \ldots \end{gathered}$$

È sbagliato definire la frase come segue:

$$\forall x \; Fratello(x, Luigi) \wedge Alto(x)$$

Infatti con questa notazione significa "tutti sono fratelli di Luigi e tutti sono alti", che non coincide con l'idea che si vuole rappresentare.
Il *quantificatore di esistenza* invece permette di esprimere asserzioni riguardanti *alcuni* oggetti del dominio senza però specificarne i nomi. Riprendendo l'esempio precedente:

$$\exists x \; Fratello(x, Luigi) \wedge Alto(x)$$

In questo questo caso si modella "qualche fratello di Luigi è alto", in particolare si afferma che deve esistere *almeno un'interpretazione* che assegna ad x un elemento del dominio.
Con il quantificatore di esistenza viene usato il connettore $\wedge$ per evitare di esprimere una semantica troppo debole, che si creerebbe se si usasse $\Rightarrow$.

$$\exists x \; Fratello(x, Luigi) \Rightarrow Alto(x)$$

Un frase del genere infatti è vera se non c'è nessun fratello di Luigi.
In generale data una formula P, un modello m e una variabile x, i due quantificatori possono essere definiti:

- $\forall x\ P$ è vera in un'interpretazione di m *se e solo se* P è vera con x che corrisponde ad **ogni** possibile oggetto del modello.

- $\exists x\ P$ è vera in un'interpretazione di m *se e solo se* P è vera con x che corrisponde a **qualche** possibile oggetto nell'interpretazione.

Possono essere definite alcune proprietà per i quantificatori come:

- $\forall x\ \forall y$ ha lo stesso significato di $\forall y\ \forall x$.

- $\exists x\ \exists y$ ha lo stesso significato di $\exists y\ \exists x$.

- $\exists x\ \forall y$ *non* ha lo stesso significato di $\forall y\ \exists x$.

- $\forall x\ \exists y\ Ama(x, y)$ significa che “esiste una persona nel mondo che ama tutti”.

- $\exists x\ \forall y\ Ama(x, y)$ significa che “tutti nel mondo sono amati da almeno una persona'.'

Inoltre può essere espresse formule *duali*:

$$\begin{array}{r|l} \forall x\ Gradisce(x, Acqua) & \neg\exists x\ \neg Gradisce(x, Acqua) \\ \exists x\ Gradisce(x, Latte) & \neg\forall x\ \neg Gradisce(x, Latte) \end{array}$$

1.4.2 Modus Pones Generalizzato

Come per la logica proposizionale anche in ***FOL*** possono essere specificate una serie di *regole di inferenza* con le quali creare serie di ragionamenti.
Si devono innanzi tutto introdurre due nuovi concetti, quello di **sostituzione** e quello di **unificazione**.
La *sostituzione* è una serie di associazioni tra una variabile ed un termine, che per esempio si può scrivere come:

$$\delta = \{x/Marco, y/Luigi\}$$

I generale quindi data una formula S e una sua sostituzione δ, $S\delta$ rappresenta il risultato di sostituire δ *in* S, si prenda ad esempio:

$$S = Fratello(x, y)$$
$$\delta = \{x/Marco, y/Luigi\}$$
$$S\delta = Fratello(Marco, Luigi)$$

L'*unificazione* invece è una regola che permette di trovare una sostituzione ed inferire una formula, per esempio dato il problema:

$$\forall x\ Femmina(x) \wedge Genitore(x) \wedge Persona(x) \Rightarrow Madre(x)$$
$$Femmina(Anna)$$
$$\forall y\ Persona(y)$$
$$Genitore(Anna)$$

Vogliamo dedurre direttamente *Madre(Anna)*, quindi si deve cercare una sostituzione θ tale che *Femmina(x), Genitore(x) e Persona(x)* abbiano una corrispondenza con *Femmina(Anna), Genitore(Anna) e Persona(y).* Come è facilmente deducibile la sostituzione corretta è $\theta = \{x/Anna, y/Anna\}$, in generale si può dire che:

$$UNIFY(\alpha, \beta) = \theta\ se\ \alpha\theta = \beta\theta$$

È possibile infine **generalizzare** il ***Modus Ponens*** nella ***FOL***, per avere a disposizione uno strumento con il quale dedurre formule nelle KB:

$$\frac{p_1^1, p_2^1, \ldots, p_n^1,\ (p_1 \wedge p_2 \wedge \ldots p_n \Rightarrow q)}{q\theta}$$
$$dove\ p_i^1\theta = p_i\theta\ per\ tutte\ le\ i$$

Molte tecniche di AI sono basate sulla logica dei predicati ed utilizzano il ***Modus Ponens Generalizzato***[17] per inferire regole, in quanto è molto semplice da programmare ed abbastanza potente. Il ***GMP*** è *solido* e *completo*, permette di derivare solo formule vere che sono direttamente implacabili dalla KB. La principale limitazione del ***GMP*** risiede nel fatto che funziona con KB che contengono solo implicazioni di *letterali positivi*[18], quindi nel caso si utilizzino disgiunzioni ($\vee$) o negazioni ($\neg$) queste interrompono

[17] d'ora in poi ***GMP***.

[18] a.k.a. Clausole di Horn.

le regole.
Nell'esempio precedente la formula può essere utilizzata come:

p_1^1 *corrisponde a* $Femmina(Anna)$ p_1 *corrisponde a* $Femmina(x)$
p_2^1 *corrisponde a* $Persona(y)$ p_2 *corrisponde a* $Persona(x)$
p_3^1 *corrisponde a* $Genitore(Anna)$ p_3 *corrisponde a* $Genitore(x)$

θ *corrisponde a* $\{x/Anna, y/Anna\}$ q *corrisponde a* $Madre(x)$
$q\theta$ *corrisponde a* $Madre(Anna)$

1.4.3 Algoritmi di concatenazione in *FOL*

Si possono ridefinire in ***FOL*** i due principali metodi di ragionamento che utilizzano le regole di inferenza, il *forward chaining* e *backward chaining.*

1.4.3.1 Forward Chaining

Applicando ripetutamente il ***GMP*** è possibile derivare conseguenze dalla KB creando così nuove informazioni. Usando l'inferenza si crea una catena di ragionamento sempre più profonda, lo scopo è individuare una corrispondenza con un fatto nella KB che soddisfi la premessa e trarre poi le conclusioni di quella determinata regola, si prenda la seguente KB:

$$Genitore(x,y) \wedge Maschio(x) \Rightarrow Padre(x,y)$$
$$Padre(x,y) \wedge Padre(x,z) \Rightarrow Fratello(x,y)$$
$$Genitore(Luca, Marco)$$
$$Genitore(Luca, Luigi)$$
$$Maschio(Luca)$$

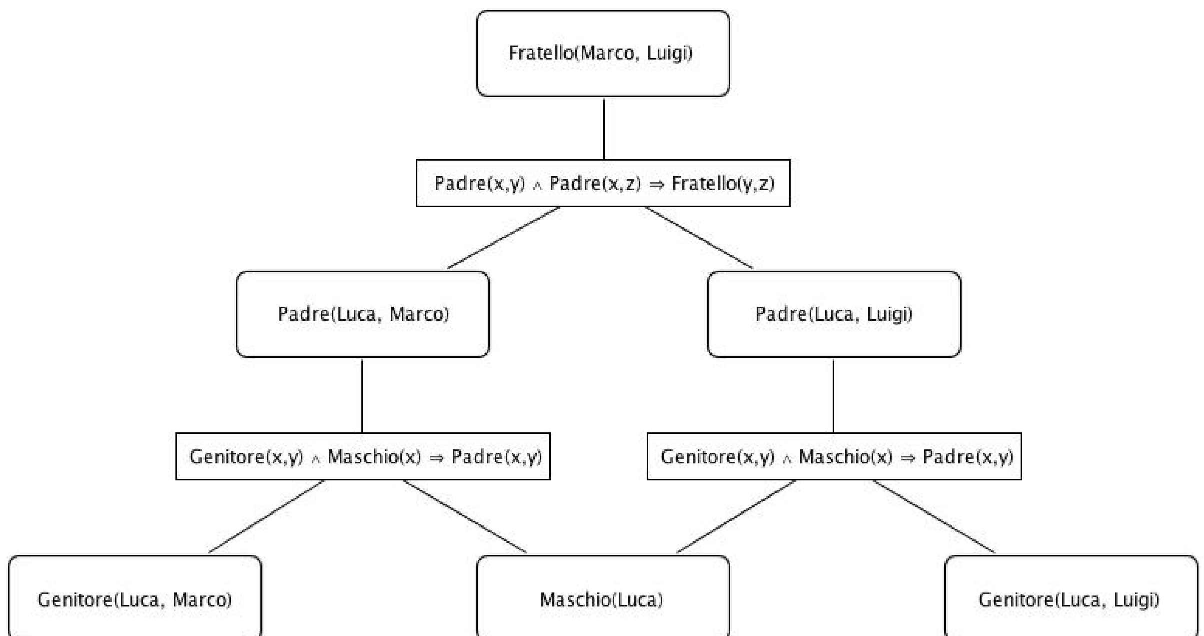
e un possibile pseudo-codice dell'algoritmo in Figura[1.2][12].
Vogliamo dimostrare che "Marco è fratello di Luigi", il ragionamento può essere esploso con il seguente albero, tenendo aggiornata la KB, per ogni regola esplosa si inserisce la sua conseguenza nella base di conoscenza. Per la risoluzione si applica la regola $Genitore(x,y) \wedge Maschio(x) \Rightarrow Padre(x,y)$ due volte ed per giungere al Goal si applica $Genitore(x,y) \wedge Maschio(x) \Rightarrow$

Figura 1.2: Pseudocodice della concatenazione in avanti.

function FOL-FC-ASK(KB, α) **returns** a substitution or *false*

  repeat until *new* is empty
    new $\leftarrow \{\}$
    for each sentence *r* **in** *KB* **do**
      $(p_1 \land \ldots \land p_n \implies q) \leftarrow$ STANDARDIZE-APART(r)
      for each θ such that $(p_1 \land \ldots \land p_n)\theta = (p'_1 \land \ldots \land p'_n)\theta$
            for some $p'_1, \ldots, p'_n$ in *KB*
        $q' \leftarrow$ SUBST(θ, q)
        if q' is not a renaming of a sentence already in *KB* or *new* **then**
do
          add q' to *new*
          $\phi \leftarrow$ UNIFY(q', α)
          if ϕ is not *fail* **then return** ϕ
    add *new* to *KB*
  return *false*

Figura 1.3: Albero di dimostrazione forward.



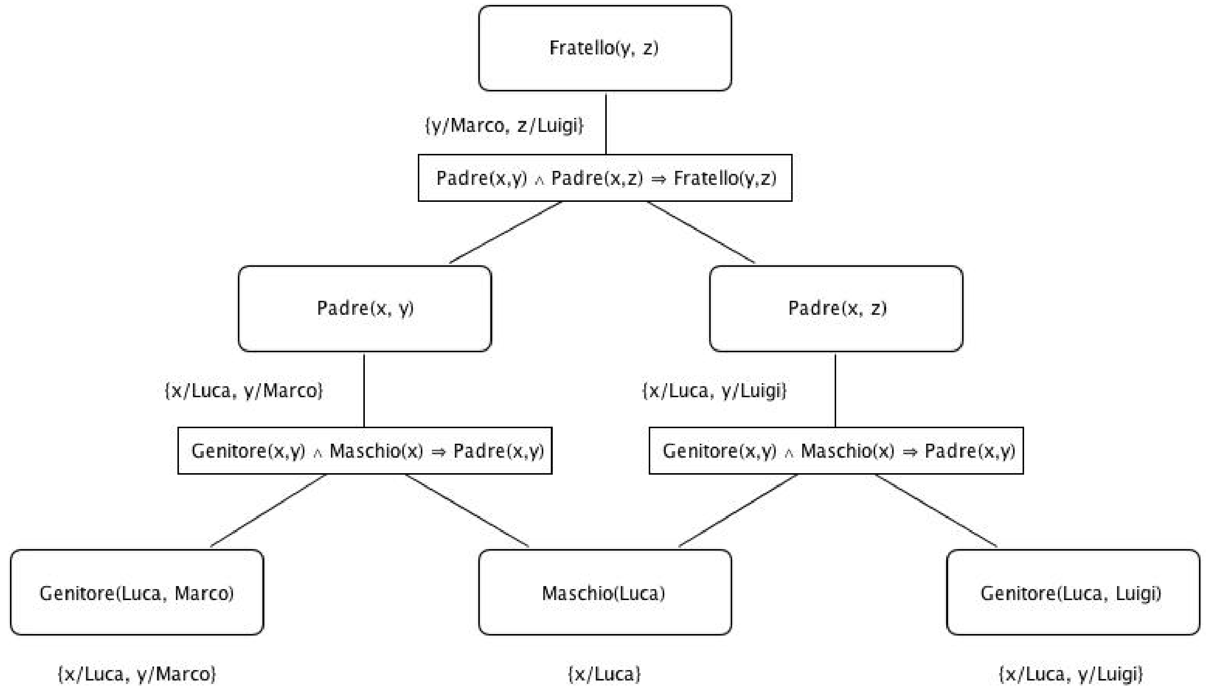
$Padre(x, y)$, inserendo ogni volta le relative conclusioni nella KB.
Nell'albero[1.3] viene mostrata una situazione in cui tutte le regole utilizzabili sono state applicate e ogni conclusione è esplicitamente presente nella KB, questa particolare situazione prende il nome di **punto fisso** del processo inferenziale.
L'algoritmo utilizza un approccio simile a quello nella logica proposizionale e risulta essere *solido*, in quanto ogni inferenza è l'applicazione del ***GMP*** e *completo* in ***FOL*** solo per KB con clausole definite. Infatti se il Goal non è implicato uno dei principali svantaggi di questo approccio è che, utilizzando l'inferenza, questa può far scattare regole in avanti infinite volte, in quanto non è diretta verso una particolare conclusione ed inoltre può trarre molte conclusioni non utili allo scopo che si vuole raggiungere.

1.4.3.2 Backward Chaining

. Il *backward chaining* parte da una *query* o una formula atomica per la quale deve essere trovata una dimostrazione. La *query* può contenere variabili e al termine del processo deve essere ritornate una serie di congiunzioni possibili per la variabile che soddisfano la *query*.
La prima operazione che viene eseguita è l'unificazione della *query* con tutti i fatti nella KB, per verificare che non sia già dimostrata. Se non si trova nessuna corrispondenza allora si cerca di provare la *query* cercando una regola che abbia nel conseguente ciò che si vuole dimostrare e iterare il ragionamento per tutte le premesse delle formule.
Utilizzando l'esempio precedente, la *query* è *Fratello(x, z)* (o *Fratello(Marco, Luigi)*), si può leggere l'albero di dimostrazione, Figura[1.3], dalla radice verso le foglie. Per prima cosa si cerca di dimostrare la formula $Padre(x, y) \wedge Padre(x, z) \Rightarrow Fratello(x, y)$ (dato che *Fratello(x, y)* non ha corrispondenze nella KB) questa genera due percorsi che sono soddisfatti dalla formula $Genitore(x, y) \wedge Maschio(x) \Rightarrow Padre(x, y)$, a questo punto le premesse *Genitore(x, y) e Maschio(x)* sono presenti nella KB, vengono quindi unificate le variabili e la loro congiunzione propagata alla radice dell'albero, vedi Figu-

ra[1.4]. Questo tipo di concatenazione è un algoritmo di ricerca in profondità

Figura 1.4: Albero di dimostrazione backward



utilizzato nella programmazione logica.

1.4.4 Risoluzione in *FOL*

Come nella logica proposizionale la risoluzione richiede che una formula sia in ***CNF***, in questo caso i letterali possono contenere variabili che possono essere universalmente quantificate. Di seguito le regole per convertire una formula in ***CNF***, si prenda la formula $\forall x\ [\forall y\ Animale(y) \Rightarrow Ama(x,y)] \Rightarrow [\exists y Ama(y,x)]$:

1. Eliminazione delle implicazioni:
   $\forall x[\neg\forall y \neg Animale(y) \vee Ama(x,y)] \vee [\exists y Ama(y,x)]$.

2. Spostamento all'interno delle negazioni, in generale:
   $\neg\forall x\ p\ \ diventa\ \ \exists x\ \neg p$

$\neg\exists x\ p\ \ diventa\ \ \forall x\ \neg p$

3. Standardizzazionoe delle variabili:
   $\forall x\ [\exists y\ Animale(y) \wedge \neg Ama(x, y)] \vee [\exists z Ama(z, x)]$.

4. Skolemizzazione: processo di rimozione dei quantificatori esistenziali per eliminazione, ogni variabile esistenziale è rimpiazzata da una funzione Skolem relativa ad una variabile universalmente quantificata.
   $\forall x[Animale(F(x)) \wedge\ \neg Ama(x, F(x))] \vee\ Ama(G(x), x)$.

5. Omissione dei quantificatori universali:
   $[Animale(F(x)) \wedge \neg Ama(x, F(x))] \vee Ama(G(x), x)$.

6. Distribuzione di $\vee$ su $\wedge$:
   $[Animale(F(x)) \vee Ama(G(x), x)] \wedge [\neg Ama(x, F(x)) \vee Ama(G(x), x)]$.

Infine la regola di risoluzione viene sollevata alla logica ***FOL***, in questa logica due letterali sono complementari se uno unifica con la negazione dell'altro, la formula può essere riscritta come:

$$\frac{l_1 \vee \ldots \vee l_k, \quad m_1 \vee \ldots \vee m_n}{SUBST(\theta, l_1 \vee \ldots \vee l_{i-1} \vee l_{i+1} \vee \ldots \vee l_k \vee m_1 \vee \ldots \vee m_{j-1} \vee m_{j+1} \vee \ldots \vee m_n}$$

dove $UNIFY(l_i, \neg m_j) = \theta$, quindi si applicano passi di risoluzione con $CNF(KB \wedge \neg\alpha)$, l'algoritmo necessita di un'euristica e una strategia per decidere con quale risoluzione procedere allo scopo di controllare la ricerca di una prova, risulta essere completo in ***FOL***.

Concludendo ***FOL*** offre un'ottima capacità espressiva per la stesura di KB di uso comune, offre uno strumento molto più vicino al linguaggio comune con oggetti, relazioni e funzioni, inoltre gli deve essere attribuita un'interpretazione in un determinato dominio.

1.5 Knowledge Engineering

Negli anni '70 nacque il primo sistema esperto[4] e, con il diffondersi di questa novità, ci si trovò di fronte alla necessità di avere un approccio sistematico per costruire sistemi basati sulla conoscenza, come accade per le metodologie dell'ingegneria software, di questo si occupa la ***Knowledge Engineering***[19]. Durante gli anni, questa disciplina si è evoluta nello studio di una teoria, metodi e strumenti per per sviluppare applicazioni ad alto contenuto conoscitivo[16].
Agli albori la ***KE*** era vista come un *processo di trasferimento* della conoscenza umana nella KB implementata, il processo si basa sull'assunzione che la conoscenza richiesta dal sistema già esiste e che deve solo essere raccolta e implementata.

"This transfer and transformation of problem-solving expertise from a knowledge source to a program is the heart of the expert-system development process"[23]

Con il passare degli anni la definizione ***KE*** si è evoluta e venne definita, con consenso crescente, un processo di *modeling activity*[22], ovvero la fabbricazione di un sistema basato sulla conoscenza significa costruire un modello con lo scopo di realizzare tutte le capacità di risoluzioni di problemi paragonabili a quelle di un esperto del dominio.
Clancey analizzò la prima generazione di sistemi esperti, sviluppati con lo scopo di risolvere compiti diversi e realizzati con formalismi di rappresentazione differenti, e scoprì un comportamento di risoluzione dei problemi comune, astraendo da questo fu in grado di derivare un *pattern* chiamato *Classificazione Euristica*, che descrive il comportamento di questi sistemi ad un livello astratto, che viene chiamato ***Livello della Conoscenza***[22]. Questo livello permette di descrivere il ragionamento in termini di Goal che devono essere raggiunti, azioni necessario per raggiungere questi Goal e conoscenza necessario per eseguire queste azioni.

[19] d'ora in poi KE.

Il processo di costruzione di una KB in uno specifico contesto, coinvolge principalmente due figure[12]:

- **Esperto del dominio**: Colui che detiene la conoscenza che si vuole incorporare nel sistema.
- **Ingegnere della conoscenza**: Colui che investiva nel particolare dominio di conoscenza dell'esperto, impara quali concetti sono importanti e scrive una rappresentazione formale degli oggetti all'interno del dominio e delle loro relazioni.

Il processo utilizza la logica per rappresentare gli aspetti più importanti del mondo reale, lo si può dividere principalmente in sette passi[12]:

1. *Identificare il compito della KB.*
2. *Raccogliere la conoscenza rilevante.*
3. *Definire un vocabolario di predicati, funzioni e costanti.*[20]
4. *Codificare la conoscenza generale riguardante il dominio.*
5. *Codificare una descrizione della specifica istanza del problema.*
6. *Interrogare la procedura di inferenza ed ottenere da essa risposte.*
7. *Fare il debugging della KB.*

1.6 Rappresentazione della Conoscenza

Nella sezione precedente, è stato affrontato il problema di come racchiudere una conoscenza di un particolare dominio all'interno di un sistema, ma come è possibile rappresentare questa conoscenza?

[20] vedi paragrafo sull'Ontologia.

1.6.1 Ontologia

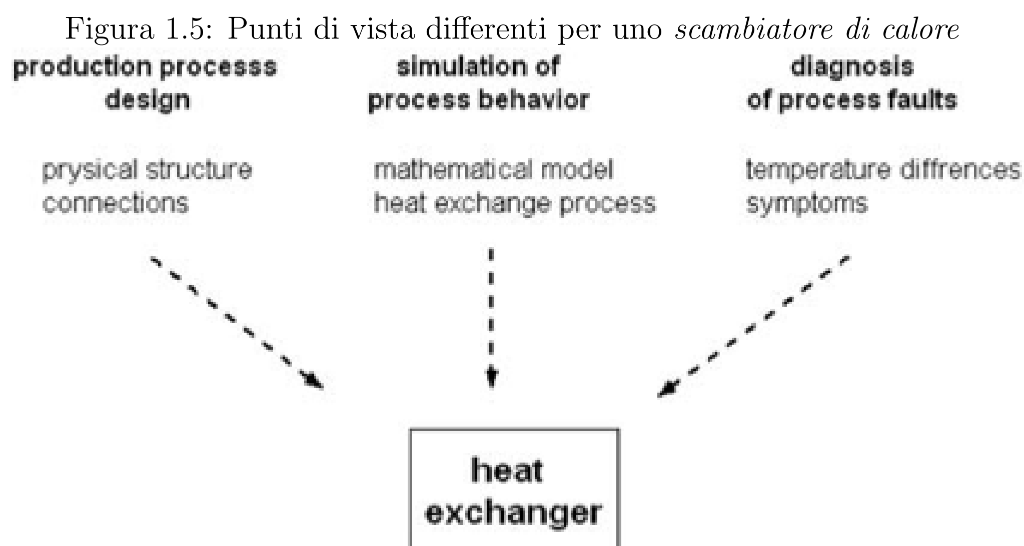
Durante gli anni '90 le ***Ontologie*** divennero popolari in informatica. Gruber[19] definisce un'***Ontologia*** come "explicit specification of a conceptualization"[21], questa definizione col tempo è stata rivista e per il momento una sua definizione è:

"An ontology is an explicit specification of a shared conceptualization that holds in a particular context."[16] [22]

È molto importante sottolineare il concetto di *sharing*, infatti lo scopo primario di un'***Ontologia*** in informatica è la *condivisione* della conoscenza. Fino alla fine degli anni '90 l'uso delle ***Ontologie*** non era molto diffuso e quindi anche la stessa parola era un termine abbastanza di nicchia, usato da pochi ricercatori nei campi dell'Ingegneria della Conoscenza e Rappresentazione. Con lo svilupparsi del ***World Wide Web*** è emerso un forte bisogno di condividere concetti e conoscenza, facendo così diventare le ***Ontologie*** uno strumento di uso comune per la condivisione di un vocabolario tra persone e programmi, sopratutto nel *Semantic Web*. Nella pratica tutto questo nasce da un "problema" molto semplice, ovvero che di fronte ad un'idea, questa può avere diversi modi per essere espressa, tutto in base al *punto di vista*, perfino all'interno di uno stesso dominio. La Figura[1.5] mostra come la concettualizzazione di uno *scambiatore di calore* è molto diversa in base a tre punti di vista, quello della progettazione del processo produttivo, nella simulazione di un processo di comportamento e nella diagnosi di fallimenti di processi.
Un altra nozione da tenere in considerazione è il *contesto* nel quale una ***Ontologia*** viene utilizzata. Fondamentalmente non ci si può aspettare che altre persone (o programmi, nel caso dell'informatica) capiscano un determinato concetto se non si esplicita in quale contesto lo si utilizza. Sono stati

[21] trad. lett. "specificazione esplicita di una concettualizzazione".
[22] trad. lett. "Un'ontologia è una specificiazione esplicita di una concettualizzazione condivisa in un particolare contesto".

Figura 1.5: Punti di vista differenti per uno *scambiatore di calore*



anche affrontati studi per definire quali *spazi contestuali* vengono usati più frequentemente, Lenat ha anche tentato di sviluppare una teoria[20].

Le ***Ontologie*** vengono utilizzate in varie forme, possono però essere grossolanamente divise in tre tipologie[16]:

- ***Ontologie*** *foundational*: Queste ***Ontologie*** hanno lo scopo di provvedere una concettualizzazione di nozioni generali, come spazio, tempo, eventi e processi.

- ***Ontologie*** *domain-specific*: Sono utilizzate per condividere concetti e relazioni in una particolare area di interesse.

- ***Ontologie*** *task-specific*: Specificano le concettualizzazioni che sono necessario per eseguire un determinato compito.

La disciplina che si occupa di costruire e mantenere le ***Ontologie*** viene chiamata ***Ontology Engineering***, questa offre linee guida per la costruzione di domini concettuali, come ad esempio la costruzione di gerarchie.

1.6.2 Description Logic

La ***Description Logic*** è una famiglia di logiche basate su linguaggi per la rappresentazione della conoscenza, che possono essere utilizzati per specificare conoscenza e struttura in uno specifico dominio di applicazione.
Il nome deriva principalmente dal fatto che le nozioni più importanti di un dominio sono rappresentate da *descrizioni* di concetti, in particolare le espressioni sono costruite da:

- **concetti** (predicati unari).
- **ruoli** (predicati binari), che usano i concetti.

Nella ***Description Logic***, oltre ai costruttori booleani, un'affermazione è tipicamente costruita da due parti:

- **TBox**: parte terminologica. Descrive nozioni importanti di un certo dominio applicativo dichiarando proprietà, ruoli e relazioni dei concetti.
- **ABox**: parte asserzione. Descrive una situazione concreta, dichiarando proprietà di individui.

Si predano i seguenti esempi, si vuole descrivere la frase "un uomo spostato con un dottore":

$$Umano \sqcap \neg Femmina \sqcap (\exists sposato.Dottore)$$
$$\underbrace{UomoFelice}_{TBox} \equiv Umano \sqcap \neg Femmina \sqcap (\exists sposato.Dottore)$$
$$UomoFelice(\underbrace{BOB}_{ABox}),\ \neg Dottore(\underbrace{MARIA}_{ABox})$$

1.6.3 OWL

Esistono vari formalisti per specificare un'***Ontologia***, in particolare l'articolo di Davis[21] analizza vari aspetti dei linguaggi per la rappresentazione della conoscenza e definisce cinque ruoli per la rappresentazione della conoscenza:

1. *"Un surrogato per le cose nel mondo reale."*
2. *"Un insieme di impegni ontologici"*
3. *"Una teoria per la costruzione rappresentazionale più l'inferenza di sanzioni/cosigli "*
4. *"Un medium per una computazione efficiente"*
5. *"Un medium per l'espressione umana"*

Si può dire che i linguaggi si concentrano principalmente sui ruoli 1,2 e 5, anche se effettivamente le ***Ontologie*** non vanno specificate con un particolare paradigma di ragionamento in mente.

L'estensione della ***Description Logic*** utilizzata più ampiamente nella definizione di ***Ontologie*** vien denominata ALC, *"Attributive concept Language with Completements"*. Nella pratica si utilizzano linguaggi basati sulla ***Description Logic*** come possono essere OIL, DAML+OIL e ***OWL***.
Quest'ultimo è un linguaggio per il web semantico (con sintassi **XML**), sviluppato dal *W3C Web-Ontology* la cui semantica può essere vista come una traslazione dalla ***Description Logic***.
Un'***Ontologia OWL*** può essere vista come una corrispondenza a un TBox della ***Description Logic*** con una gerarchia di ruoli, che descrivono il dominio in termini di *classi* (che corrispondono ai concetti) e *proprietà* (che corrispondono ai ruoli). Un'***Ontologia*** consiste in un insieme di assiomi che asseriscono relazioni tra *classi* e *proprietà* Tabella[1.1].
Come nella ***Description Logic***, le classi ***OWL*** possono essere nomi o espressioni costruite da classi più semplici e proprietà che usano costruttori. Le tabelle 1.6 e 1.7 riassumo i costruttori e gli assiomi utilizzati in ***OWL*** con l'equivalente sintassi in ***Description Logic***.

Si possono scrivere alcuni esempi in **XML** con sintassi **RDF**, $Umano \sqcap Maschio$

Figura 1.6: Costruttori OWL

Constructor	DL syntax	Example
intersectionOf	$C_1 \sqcap \cdots \sqcap C_n$	Human $\sqcap$ Male
unionOf	$C_1 \sqcup \cdots \sqcup C_n$	Doctor $\sqcup$ Lawyer
complementOf	$\neg C$	$\neg$Male
oneOf	$\{x_1 \ldots x_n\}$	{john, mary}
allValuesFrom	$\forall P.C$	$\forall$hasChild.Doctor
someValuesFrom	$\exists r.C$	$\exists$hasChild.Lawyer
hasValue	$\exists r.\{x\}$	$\exists$citizenOf.{USA}
minCardinality	$(\geqslant nr)$	($\geqslant$ 2 hasChild)
maxCardinality	$(\leqslant nr)$	($\leqslant$ 1 hasChild)
inverseOf	r^-	hasChild$^-$

Figura 1.7: Assiomi OWL

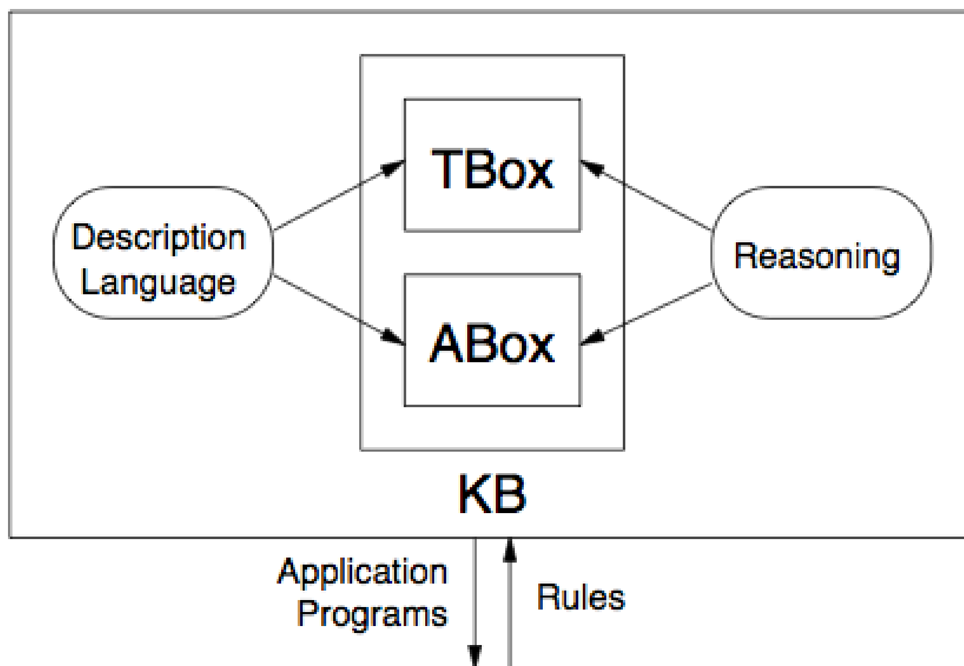
Axiom	DL syntax	Example
subClassOf	$C_1 \sqsubseteq C_2$	Human $\sqsubseteq$ Animal $\sqcap$ Biped
equivalentClass	$C_1 \equiv C_2$	Man $\equiv$ Human $\sqcap$ Male
subPropertyOf	$P_1 \sqsubseteq P_2$	hasDaughter $\sqsubseteq$ hasChild
equivalentProperty	$P_1 \equiv P_2$	cost $\equiv$ price
disjointWith	$C_1 \sqsubseteq \neg C_2$	Male $\sqsubseteq$ $\neg$Female
sameAs	$\{x_1\} \equiv \{x_2\}$	{Pres_Bush} $\equiv$ {G_W_Bush}
differentFrom	$\{x_1\} \sqsubseteq \neg\{x_2\}$	{john} $\sqsubseteq$ $\neg${peter}
TransitiveProperty	P transitive role	hasAncestor is a transitive role
FunctionalProperty	$\top \sqsubseteq (\leqslant 1\ P)$	$\top \sqsubseteq$ ($\leqslant$ 1 hasMother)
InverseFunctionalProperty	$\top \sqsubseteq (\leqslant 1\ P^-)$	$\top \sqsubseteq$ ($\leqslant$ 1 isMotherOf$^-$)
SymmetricProperty	$P \equiv P^-$	isSiblingOf $\equiv$ isSiblingOf$^-$

```
<owl:Class>
  <owl:intersectionOf rdf:parseType="Collection">
    <owl:Class rdf:about="#Umano"/>
    <owl:Class rdf:about="#Maschio"/>
  </owl:intersectionOf>
</owl:Class>
```

oppure $\geq 2 haFiglio.Thing$ potrebbe essere scritto come

```
<owl:Restriction>
  <owl:onProperty rdf:resource="#haFiglio"/>
  <owl:minCardinality
    rdf:datatype="&xsd;NonNegativeInteger">2
  </owl:minCardinality>
</owl:Restriction>
```

Figura 1.8: Architettura di un sistema basato sulla rappresentazione della conoscenza in ***Description Logic*** [18].



Concludendo esistono vari tipologie di ***Ontologie*** e vengono largamente usate nel mondo dell'informatica per condividere la conoscenza in un particolare dominio, in modo da creare un vocabolario uniforme per chiunque la utilizzi, siano esse persone o macchine e facilitare in questo modo il processo di costruzione di un modello del dominio.

OWL	DL
classe	concetto
proprietà	ruolo
oggetto	individuo

Tabella 1.1: Differenze tra ***OWL*** e la ***Description Logic***.

1.7 Programmazione Logica

La *rappresentazione della conoscenza* è diventata, col tempo, una delle più importanti aree della AI. Se lo scopo è creare una macchina o un programma capace di comportarsi in modo intelligente in un particolare dominio, si dovrà fornire alla macchina o al programma una conoscenza sufficiente di tale dominio. Per fare questo è necessario un linguaggio non ambiguo capace di esprimere la conoscenza, inoltre servono delle precise metodologie per manipolare gruppi di formule di un linguaggio in modo da permettere di trarre inferenze, rispondere a *query* e di aggiornare quindi sia la conoscenza che il comportamento del programma[26]. Nel 1960, McCarthy[27] propose l'uso di formule logiche come base per un linguaggio di rappresentazione della conoscenza, di seguito le sue esatte parole:

Expressing information in declarative sentences is far more modular than expressing it in segments of computer programs or in tables. Sentences can be true in a much wider context then a specific programs can be used. The supplier of a fact does not have to understand much about how the receiver functions or how or whether the receiver will use it. The same fact can be used for many purposes, because of the logical consequences of collections of facts can be available.

L'idea è stata sviluppata da molti ricercatori, prima di tutto venne usata la logica dei predicati come principale strumento per la rappresentazione della conoscenza. Questa ha una ben definita semantica e un ben definito mec-

canismo inferenziale e si è dimostrata abbastanza potente per rappresentare la conoscenza matematica. Risulta però essere totalmente inadeguata per la rappresentazione della cosiddetta *common sense knowledge*[23], il problema è strettamente legato alla definizione di "monotonicità" delle teorie basate sul calcolo dei predicati.
Una logica si definisce *monotona* se l'aggiunta di nuovi assiomi alla teoria non porta alla perdita dei teoremi provati fino a quel momento nella teoria stessa, il ragionamento comune è però *non-monotono*. Questa osservazione ha portato così allo sviluppo di nuovi formalismi logici, le *logiche non-monotone*.
Ma Green, Hayes e Kowlaski presero un'atra direzione[26] e combinarono l'idea di logica come linguaggio di rappresentazione con la teoria di deduzione automatica e logica costruttiva, che portò Kowalski e Colmerauer alla creazione della ***programmazione logica***[28] e allo sviluppo del primo linguaggio logico, ***Prolog***[29].
Kowalski quindi introdusse un nuovo paradigma di programmazione riassunto dalla seguente formula[30]:

Algorithm = Logic + Control

Un algoritmo può essere visto come l'insieme di due componenti, la *logica* che specifica la conoscenza che deve essere usata per risolvere i problemi e il *controllo* che determina le strategie per la risoluzione dei problemi attraverso l'uso della conoscenza. Inoltre Kowalski identificò due tecniche per interpretare problemi specificati con *clausole di Horn*, una *backward*[24] e una *forward*[25].
Un problema in *clausole di Horn* viene descritto con la seguente forma[30]:

1. un **insieme di clausole** che definisce il dominio del problema.

2. un **teorema** che consiste in:

    - **ipotesi** rappresentate da asserzioni del tipo: $A_1 \leftarrow, \ldots, A_n \leftarrow$

[23] trad. "conoscenza del senso comune".
[24] chiamata da lui *top-down*.
[25] chiamata da lui *bottom-up*.

- **conclusione** che è negata o rappresentata da una negazione: $\leftarrow B_1, \ldots, B_m$.

Nell'approccio *backward* si ragiona dalla conclusione, riducendo ripetitivamente i Goal in sotto-goal finche tutti non vengono risolti direttamente dalle asserzioni. Al contrario nell'approccio *forward* si ragiona dalle ipotesi, derivando nuove asserzioni dalle vecchie finche il Goal originale non viene risolto direttamente da una asserzione derivata.
Prendiamo ad esempio il problema di dimostrare che "Zeus è nonno di Harmonia", il problema può essere risolto sia *backward* che *forward.* Nel primo caso si parte dalle asserzioni:

$$Padre(Zeus, Ares) \leftarrow$$
$$Padre(Ares, Harmonia) \leftarrow$$

si usa poi la clausola $Genitore(x, y) \leftarrow Padre(x, y)$ per derivare le nuove asserzioni:

$$Genitore(Zeus, Ares) \leftarrow$$
$$Genitore(Ares, Harmonia) \leftarrow$$

infine si deriva dalla definizione di nonno l'asserzione che si voleva dimostare:

$$Nonno(Zeus, Harmonia) \leftarrow$$

Ragionando invece in modo *forward* si parte dal Goal originale "Zeus è nonno di Harmonia":

$$\leftarrow Nonno(Zeus, Harmonia)$$

e si usa questa definizione per generare due sotto-goal, negando che qualunque z è "figlio di Zeus e genitore di Harmonia":

$$\leftarrow Genitore(Zeus, z), Genitore(z, Harmonia)$$

continuando si usa la clausola seguente per risolvere di due sottoproblemi:

$$Genitore(x, y) \leftarrow Padre(x, y)$$

derivando:

$$Genitore(Zeus, x) \leftarrow$$
$$Genitore(x, Harmonia) \leftarrow$$

che sono entrambi risolti nelle asserzioni sostituendo il valore di z con "Ares".

Concludendo la programmazione logica mira a mescolare l'utilizzo della conoscenza e del controllo sotto forma di un nuovo paradigma di programmazione, ed è nata dalla necessità di far ragionare le macchine in un modo più vicino al *common sense.*

1.7.1 Prolog

È stato originariamente sviluppato da Alain Colmerauer e Philippe Roussel[25] nel 1972, il nome venne suggerito dalla moglie di Russel che abbreviò la definizione "programmation en logique"[26].
Come detto nelle sezione precedente ***Prolog*** fu uno dei primi linguaggi per la programmazione logica, è utilizzato in molti campi come la dimostrazione di teoremi, sistemi esperti, giochi, sistemi di risposta automatici e sistemi di controllo sofisticati.
La programmazione logica in ***Prolog*** è espressa con *termini* e *relazioni* e il calcolo viene fatto eseguendo *query* su queste relazioni, che sono costruite usando l'unico tipo di dato disponibile in questo linguaggio il *termine.*
Prolog viene definito come un linguaggio *dichiarativo* in quanto un programma viene visto come:

- una lista di *fatti.*
- un fatto è un *atomo* seguito da un punto (".").
- un atomo è un nome di *predicato* (con la prima lettera minuscola), con una lista di termini, il numero degli argomenti determina l'*arità* del predicato

[26] trad. "logica di programmazione".

- un *termine* può essere un numero o una costante (con la prima lettera minuscola)

- *variabili* possono essere qualsiasi termine (con la prima lettera maiuscola)

Formule in logica possono essere tradotte in sintassi ***Prolog*** e viceversa, di seguito alcuni esempi di sintassi e una tabella riassuntiva di alcuni operatori principali con la loro arità:

Implicazione	:-/2
Congiunzione	,/2
Disgiunzione	;/2

```
Fact.
Head :- Body.
Head :- Body1,Body2.
Head :- Body1;Body2.
```

La computazione in ***Prolog*** avviene come una ricerca in un spazio di soluzioni (albero delle soluzioni), quello che fa il motore ***Prolog*** per risolvere un problema è prendere una determinata direzione tra le tante possibili (il ramo di un albero) assumendo che sia quella corretta, se non si dimostra tale utilizza il *backtracking* per ritornare sui passi computazioni ed esplorare un'altra possibilità (un altro ramo dell'albero). Se si dimostra corretta il motore ***Prolog*** ritorna una possibile soluzione ed unifica[27] tutte le variabili con un loro possibile valore. Riprendendo l'esempio precedente nel quale si vuole dimostrare "Zeus è nonno di Harmonia", si hanno i seguenti termini:

[27] vedi cap. sulle Logiche.

padre(zeus, ares).

$$\underbrace{padre(ares, armonia).}_{fatto}$$

$$\underbrace{genitore}_{nome\ del\ predicato}(X,Y) :- \underbrace{\underbrace{padre}_{nome\ del\ funtore}(X,Y).}_{termine\ composto}$$

nonno(X,Z):-genitore(X,Y),genitore(Y,Z).

Prolog alla *query "nonno(X,Y)."* risponde come segue:

yes.
X / zeus Y / harmonia
Solution: nonno(zeus,harmonia)

In conclusione ***Prolog*** è un linguaggio dichiarativo, si presta molto bene per rappresentare la conoscenza, in particolare nei capitoli successivi verrà fatto uso di una versione chiamata *tuProlog*.

1.8 Sistemi Basati sulla Conoscenza

Un sistemo basato sulla conoscenza[28] è un sistema che usa tecniche di intelligenza artificiale per processi di risoluzioni di problemi che simulano le decisioni prese da un esperto, l'apprendimento e le azioni[31].
Feigenbaum, uno degli ideatori del primo sistema esperto, fornisce una definizione ci cosa si intende per conoscenza ed elenca sostanzialmente due tipologie[11]:

- **Fatti del dominio**: questa è la conoscenza più largamente diffusa, comprende tutto il materiale scritto nei libri di testo e nei giornali specialisti.

- **Conoscenza Euristica**: tutta la conoscenza che include le regole di competenza in un determinato campo ovvero le regole di buona

[28]knowledge-based system, d'ora in poi KBS.

esecuzione, che contrariamente ai Fatti del Dominio sono trasmesse indirettamente ed imparate da un'esperto tramite l'esperienza[29].

1.8.1 I Sistemi Esperti

L'atto di ottenere, formalizzare e mettere all'opera una conoscenza viene definito *expertise modeling*[30][11]. Grazie ad esso si possono ottenere i programmi così detti ***"Expert Systems"***[31], il principale obiettivo di questi sistemi è raggiungere prestazioni ad alto livello su problemi che sono abbastanza difficili da richiede la competenza di una persona.

A computer program capable of performing at the level of a human expert in a narrow problem area.[14]

Sistemi di questo tipo consistono principalmente in due componenti[11]:

- la base di conoscenza (*knowledge base*): contiene tutti i fatti del dominio e le euristiche
- *motore inferenziale*: consiste nei processi che lavorano la KB, deduce conclusioni al problema

Il processo che per la creazione di un ***ES***, prevede quindi la raccolta della conoscenza e la messa in opera di un sistema capace di operare e dare risposte in una specifica area, il tutto solitamente viene gestito da quattro attori principali[14]:

- **L'esperto del dominio**: è una persona capace a risolvere problemi in una specifica area di dominio, nella quale ha una grossa competenza conoscitiva, questa è quella che deve essere catturata e racchiusa nella KB.

[29] L'insieme di queste regole è chiamato da *George Polya*, "*the art of good guessing*".
[30] atto di modellare una competenza.
[31] trad. "Sistemi Esperti", d'ora in poi abbreviati con ES.

- **L’ingegnere della conoscenza**: è colui in grado di progettare, costruire e testare il sistema esperto e la base di conoscenza, seleziona i compiti che deve eseguire il sistema esperto, attraverso una serie di interviste all’esperto del dominio cerca di carpire la conoscenza e come un particolare problema viene risolto ed infine decide con quale software implementare il sistema, riassumendo esegue l’*expertise modeling.*

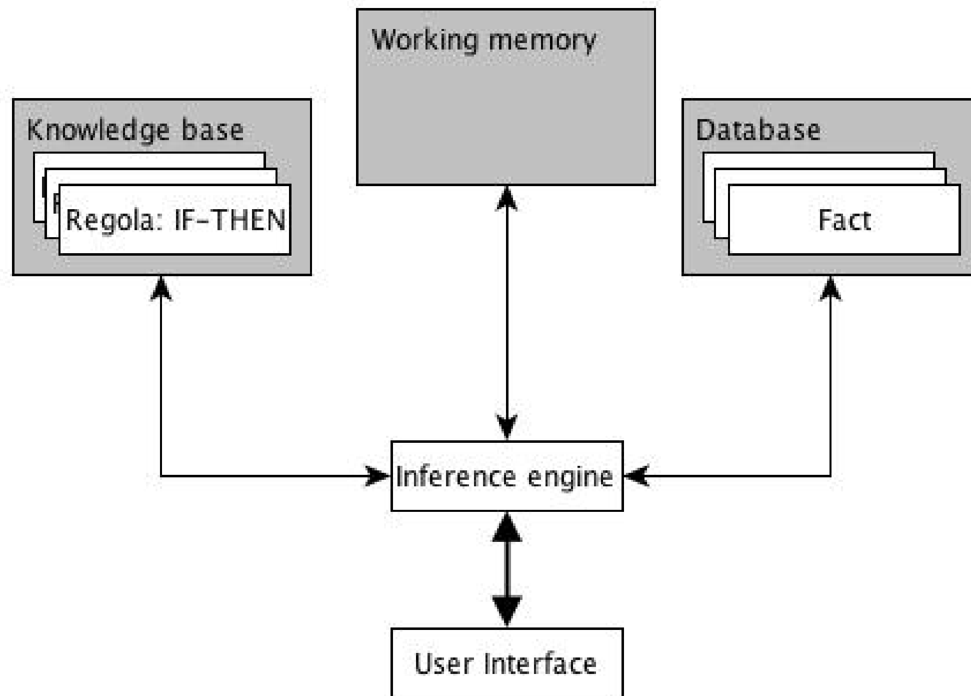
- **Il programmatore**: è la persona responsabile di descrivere la conoscenza del dominio in modo tale che un computer possa capirla.

- **Il project manager**: il responsabile del progetto, decide i vari obiettivi e tiene traccia del lavoro svolto.

In letteratura si possono trovare varie tipologie di ***ES***, in particolare però bisogna distinguere tra quelli *case-based* e *rule-based.*
L’idea alla base del *case-based reasoning* è la risoluzione di nuovi problemi tenendo presente soluzioni prese per problemi precedenti a quello attuale, ovviamente tutti specifici in un particolare dominio e considerati simili tra di loro, in questo caso il ***ES*** viene visto come una *black-box* alla quale viene dato come input un problema e darà in output la soluzione migliore (in base alla conoscenza a disposizione). In questi sistemi la KB è composta da una serie di *casi*, che vengono solitamente forniti dall’esperto del dominio all’ingegnere della conoscenza.
I sistemi esperti *rule-based* sono sicuramente quelli più largamente diffusi[14], il problema viene risolto applicando una serie di regole per dare in uscita una soluzione. In questo caso l’ingegnere della conoscenza deve carpire dall’esperto del dominio tutti ragionamenti che attua a fronte di un particolare problema, tutte queste scelte verrano poi racchiuse nella KB per poi essere utilizzate dal motore inferenziale per effettuare i dovuti ragionamenti. Un ***ES*** di questo tipo ha una struttura formata principalmente da cinque componenti: la base di conoscenza, il database, il motore inferenziale, la memoria di lavoro e l’interfaccia utente.

La ***KB*** contiene la conoscenza del dominio utile a risolvere problemi. In

Figura 1.9: Architettura di un sistema esperto.[14]



un ***ES*** *rule-based* la conoscenza è rappresentata da una serie di regole, ognuna di esse specifica una relazione, una raccomandazione, una direttiva, una strategia o un'euristica, tutte implementate con il *pattern* **IF-THEN**.
Il **database** contiene tutti i fatti che vanno usati per far scattare le regole memorizzate nella KB.
Il **motore inferenziale** implementa il ragionamento, in sostanza specifica come il sistema esperto raggiunge una particolare soluzione, si occupa di collegare i fatti nel database con le regole nella KB.
La **memoria di lavoro** può contenere fatti o goal in base alla tipologia di controllo che viene utilizzata dal motore inferenziale.
Infine l'**interfaccia utente** permette la comunicazione con chiunque voglia interrogare il sistema esperto.

Ogni regola quindi consiste in due parti:

$$IF \quad \underbrace{< pattern >}_{premessa\ o\ antecedente} \quad THEN \quad \underbrace{< body >}_{conclusione\ o\ azione}$$ [32]

e vengono attivate tramite *pattern matching*, quando la premessa è vera allora viene eseguito l'azione, una regola può avere premesse multiple unite dall'*AND* logico oppure dall'*OR*, possono esserci alcuni casi in cui entrambi sono necessari ma di buona norma si cerca di limitare l'uso solo ad uno dei due connettivi.
Le regole si possono dividere in cinque diverse categorie ognuna con una diversa semantica:

- **Relazioni**

  SE il serbatoio è vuoto ALLORA la macchina non parte

- **Raccomandazioni**

  SE la stagione è autunno E il cielo è nuvoloso ALLORA il consiglio è "di prendere l'ombrello"

- **Direttive**

  SE l'auto non si avvia E il serbatoio è vuoto ALLORA riempi il serbatoio

- **Strategie**

  SE l'auto non si avvia ALLORA controlla il serbatoio il serbatoio; passo1 completato
  SE passo1 completato E il serbatoio è pieno ALLORA controlla la batteria; passo2 completato

- **Euristiche**

  SE il liquido è scuro E ha pH $<$ 6 E ha profumo acidulo ALLORA il liquido è aceto balsamico

[32] "se...allora..."

1.8.2 Controllo

Una volta che viene individuata la conoscenza questa deve essere codifica sulla base delle regola sopra citate (descritte con un linguaggio di programmazione logica) e memorizzata all'interno delle KB. A questo punto il motore di inferenza compara ogni regola memorizzata nella KB con i fatti presenti nel database, quando viene trovato una corrispondenza tra una precondizione e un fatto il motore fa scattare (*fire*) la regola, eseguendo la conclusione. La comparazione dei fatti con le premesse delle regole genera le **catene di ragionamento**, in sostanza queste indicano come il sistema esperto applichi determinare regole per giungere alla conclusione del problema.
Principalmente esistono due tipologie di concatenamento[33], quello in avanti (*forward*) e quello all'indietro (*backward*).
Nel primo caso la memoria di lavoro viene inizializzata con i fatti presenti nel database, le regole che vengono fatte scattare sono quelle dove la premessa viene soddisfatta dai fatti nella memoria di lavoro (altrimenti dette *F-Rules*), una volta eseguita una regola la sua conclusione viene inserita nella memoria di lavoro, il procedimento termina o quando viene inserito nella memoria di lavoro il fatto di terminazione oppure quando non ci sono più regole da applicare (fallimento).
Nel secondo caso la memoria di lavoro contiene i Goal da dimostrare, le regole che si vanno ad applicare sono quelle dove la conclusione corrisponde ad un fatto nella memoria di lavoro (altrimenti dette *B-Rules*), se una regola trova corrisponde vengono aggiunti sotto-goal da dimostrare alla memoria di lavoro il procedimento termina se vengono inseriti fatti noti nella memoria di lavoro oppure se non ci sono più regole da applicare.
Può capitare che più regole per un medesimo fatto possano essere applicate, in queste situazioni non esiste una regola precisa su come procedere, si può seguire l'ordine di stesura oppure è anche ipotizzabile la stesura di una *meta-algoritmo* che imponga un ordine sulle regole.
La scelta della tipologia di controllo sostanzialmente dipende da come l'esper-

[33] vedi Sezione Logiche.

to del dominio risolve il problema[14], anche se possono essere fatte alcune considerazioni iniziati in base al numero di fatti o al numero di goal disponibili.
È pensabile inoltre una combinazione dei due metodi sopra citati, detto *controllo bidirezionale*, in questo caso la memoria di lavoro viene suddivisa in due parti e vengono applicate simultaneamente *F-Rules* e *B-Rules*, il ragionamento finisce nel momento in cui le due parti di memoria di lavoro coincidono.
L'architettura di un ***ES*** è molto modulare, per il nuovo paradigma di programmazione [30] il controllo è nettamente separato dalla conoscenza del sistema (Logica), anche se lavorano a stretto contatto. Una ***Expert System Shell*** è sostanzialmente il motore di inferenziale di un ***ES*** (Controllo) senza la conoscenza, questo permette a chiunque di riutilizzare il sistema cambiando la KB, ottenendo così dal sistema risposte diverse basate sulla nuova KB.

Concludendo gli ***ES*** sono dei sistemi basati sulla conoscenza, vengono utilizzati per risolvere problemi non banali e la loro costruzione avviene mediante l'interazione di varie figure, tra cui quella di spicco dell'esperto del dominio che dovrà fornire la conoscenza necessario da racchiudere nella KB. Vedremo nei capitoli successivi la costruzione di un sistema esperto per la risoluzione di un particolare problema, definito *“Menu Planning”*.

Capitolo 2

Approccio al problema del *"Menu Planning"*

Quotidianamente ci si trova ad affrontare il problema di cosa mangiare durante la giornata, di comprare alimenti per sfamare se stessi e la propria famiglia, molto spesso quando si è davanti alla scelta tra due prodotti ci si domanda quale sia quello più adatto, che faccia risparmiare ma che rispetti i nostri i gusti, tante persone si chiedono cosa possono mangiare per dimagrire o mantenersi in forma. Il *"Menu Planning"* cerca di guidare tutte queste scelte, offrendo pianificazioni degli alimenti anche su intere settimane, rispettando una serie di condizioni come il prezzo, i gusti etc.

George Dantzig, conosciuto anche come il padre della programmazione lineare, inventò la materia durante gli anni della Seconda Guerra Mondiale, ne formulò il modello e poco dopo anche un algoritmo di risoluzione[32]. Quello di cui aveva bisogno era un problema su cui eseguire dei test per verificare quello che aveva elaborato. Dopo una riunione al Pentagono gli venne proposto dallo statistico Jerry Cornfield, di lavorare ad un problema sul quale molti anni prima lui si era dedicato senza successo; l'Esercito Americano voleva abbassare i costi degli alimenti ai soldati che combattevano nella Seconda Guerra Mondiale mantenendo però il fabbisogno nutritivo di cui avevano bisogno. Dantzig si interessò, facendo delle ricerche emerse che

anche George Stingler aveva formulato ipotesi su questa problematica[33]. Stingler nel 1982 ricevette il Nobel in Economia per il suo problema di ottimizzazione e Dantzig dedicò un capitolo del suo libro al lavoro di Stingler, così nacquero il primo *diet problem* e la programmazione lineare[34], che col passare dei decenni si sarebbe evoluto nel *"Menu Planning"*.

Un Problema di Programmazione Lineare è un problema di ottimizzazione nel quale sono presenti una *funzione obiettivo*, che deve essere massimizzata o minimizzata e dei *vincoli* su una serie di variabili decisionali, per esempio[34]:

$funzione\ obiettivo = \sum_{i=1}^{N} c_i \cdot x_i = \underline{c}^T \underline{x}$
N = numero di variabili che descrivono il problema
$\underline{c}$ = vettore coefficienti della funzione obiettivo
$\underline{x}$ = il vettore delle variabili

Lo scopo principale del ***diet problem*** è trovare la combinazione meno costosa che però permette un fabbisogno nutrizionale giornaliero minimo ad una persona[32] (nel caso di Stingler erano i soldati statunitensi). Per risolvere il problema devono essere conosciute le informazioni nutrizionali di ogni cibo o prodotto e i vincoli nutritivi che si vogliono mantenere nella persona in questione, la funzione obiettivo è la somma dei costi di ogni cibo e i vincoli sono dai fabbisogni nutritivi, per esempio Soden utilizza questi vincoli giornalieri per una donna di 50 anni sovrappeso[5]:

Energia $\leq$ 1500 kcal
Fibre (in grammi) $\geq$ 20g
Sodio (in mg) $\leq$ 1600mg
Grasso (in grammi) $\leq$ 35% di energia

Si utilizzano le informazioni degli alimenti;

	Energia	Fibre	Sodio	Grasso
Succo d'arancia (150gr)	x_1^1	x_1^2	x_1^3	x_1^4
Lattuga (30gr)	x_2^1	x_2^2	x_2^3	x_2^4
...				

Infine si applica la funzione obiettivo per massimizzare l'apporto nutritivo rispettando i vincoli.

Il problema negli anni successivi è stato rivisitato[1], a fronte delle necessità di molte istituzioni di gestire i cibi già preparati e il fabbisogno di un numero elevato di persone.
Tutte le decisioni che prendono istituzioni, quali possono essere ospedali o mense, riguardanti la scelta dei cibi tenendo conto degli aspetti nutritivi o dei costi, vengono indicate con l'acronimo di *"Menu Planning"*, il cui principale obiettivo è massimizzare il soddisfacimento di una serie di vincoli. Possono essere fatti ragionamenti su intere settimane o mesi, solitamente nei casi di istituzioni quello che si cerca di fare è mantenere una spesa minima garantendo alle persone un certo apporto di energie.
Il problema può essere affrontato a vari livelli di astrazione, per esempio:

- ***Planning* del singolo piatto**: scegliere quali ingredienti in un piatto possono essere inseriti.

- ***Planning* del pasto**: decidere quali alimenti all'interno di una pasto assumere.

- ***Planning* del giorno**: determinare quali alimenti inserire all'interno di una giornata.

- ***Planning* della settimana**[1]: pianificare ogni singolo pasto della settimana.

Con lo sviluppo dei primi calcolatori nacque l'interesse di risolvere il problema con semplici algoritmi[1]. Il *"Menu Planning"* ha seguito molte variazioni d'uso durante gli anni, al di la delle necessità delle istituzioni, utilizzare questo paradigma per affrontare la pianificazione di una dieta per una persona (o gruppi) si è radicato pian piano nella AI. Son state quindi affrontate

[1] Solitamente non si prosegue in quanto un Planning mensile può essere composto da 4 planning settimanali.

una serie di metodologie fin dagli anni '60, partendo dalla programmazione lineare appunto, passando per gli algoritmi genetici fino ad arrivare agli ***ES***, di cui tratta questo elaborato, che risolvono il problema ad un certo livello di astrazione o anche a tutti i livelli visti precedentemente.

2.1 Computer e *"Menu Planning"*

Nell'Aprile del 1964 Balintfy in[1], utilizzando la programmazione lineare, ha creato un codice che pianifica menú trovando combinazioni a costo minimo di cibi, in modo tale da soddisfare vincoli gastronomici o di produzione per una serie di giorni, nacque così il primo *Menu Planner* basato su computer. Negli anni successivi vennero sviluppate tecniche di selezione casuale per il soddisfacimento di svariati vincoli nei menú da Brown (1966)[2], lavoro che fu poi ripreso da Eckstein in[3] l'anno successivo.
A queste pubblicazioni segue un drastico calo nell'interesse in questa materia, probabilmente dovuto alla mancanza di fondi e alle arretrate tecniche software, in questi anni però si verificò un evento che diete una forte scossa all'Intelligenza Artificiale.
Durante i primi anni '70 la NASA inviò una sonda senza pilota su Marte, parte del programma prevedeva la determinazione della struttura molecolare del suolo marziano attraverso l'utilizzo dei dati raccolti da uno spettrometro di massa. Uno studente di Herbert Simon, *Edward Feigenbaum*, un scienziato di computer, *Bruce Buchanan* e un vincitore del premio Nobel per la genetica, *Joshua Lederberg* si trovarono a lavorare insieme per risolvere questo problema. Il sistema che svilupparono serviva per le analisi chimiche, fu chiamato DENDRAL[4] e venne riconosciuto pochi anni più tardi come il **primo Sistema Esperto**.
Negli anni '90 il problema del *"Menu Planning"* ritornò ad essere popolare, le tecniche di Intelligenza Artificiale si erano evolute e venne ritrovato interesse nella materia. Vennero riprese le tecniche di programmazione lineare da Soden[5] e cominciarono a distinguersi due tipi di ***ES*** per il *"Menu*

Planning", *case-based* e *rule-based.* In[6] Hinrichs descrive la progettazione di un sistema *case-based*, JULIA, per il *"Menu Planning"* di pasti che soddisfacessero i gusti di più persone, qualche anno dopo, nel 1995, Ganeshan e Farmer[7] implementarono un sistema di catering per il *"Menu Planning"* utilizzando *Prolog.*
Negli anni successivi (1998) Petot, Marling e Sterling in[8] eseguirono uno studio molto approfondito della materia che portò a CAMP, pianificatore di menú *case-based*, e PRISM, generatore di menú *rule-based.* Vista la presenza di aspetti negati e positivi in entrambi gli approcci si decise di costruire un sistema ibrido che combinasse, la creatività nella generazione di menú di PRISM con la spiccata capacità di soddisfare vincoli di CAMP, creando così CAMPER. Uno dei punti di forza di quest'ultimo sistema (prima utilizzato in PRISM) è l'uso di un database gerarchico che permette di semplificare la scelta dell'alimento sulla base di una particolare tipologia di pasto.
Applicando il *case-based reasoning* ai Sistemi Esperti si ha uno spostamento del noto *"bottle-neck"* dall'acquisizione della conoscenza alla stesura delle regole di adattamento. Normalmente infatti il *"bottle-neck"* si verifica durante la fase di acquisizione della conoscenza, il dialogo tra Ingegnere della conoscenza e Esperto del dominio, risulta spesso molto tedioso e stilare delle regole precise può risultare complicato. Con l'applicazione del *case-based reasoning* questo problema non sussiste, infatti la conoscenza rimane intrinseca nei casi già risolti. Un approccio simile è stato utilizzato in MIKAS (2003)[9], questo sistema utilizzando il *case-based reasoning* e regole RDR classifica il paziente, viene poi applicata una funzione FUZZY che valuta vari parametri e consiglia il caso che più si adatta al corrente. Per migliorare la precisione del menú suggerito possono essere applicate una serie di variazioni basate sulle regole di adattamento (*adaption rules* cit. [9]), cioè *meta-regole* per la gestione del processo inferenziale nell'albero RDR, la definizione di queste regole risulta essere il nuovo "collo di bottiglia" in quanto da esse dipende un miglioramento nella precisione del sistema.
Come riportato sopra il *"Menu Planning"* non è stato affrontato solo utiliz-

zando i Sistemi Esperti, nel 2005 venne sviluppato un generatore automatico di menú utilizzando gli Algoritmi Genetici, MenuGene [10], integrato in un più importante sistema di *lifestyle consulting* chiamato Cordelia. Questo generatore suddivide il problema di un *"Menu Planning"* settimanale in sottoproblemi, frazionando la settimana in giorni, i giorni in pasti ed applicando a tutti i livelli Algoritmi Genetici per soddisfare i vincoli nutrizionali.

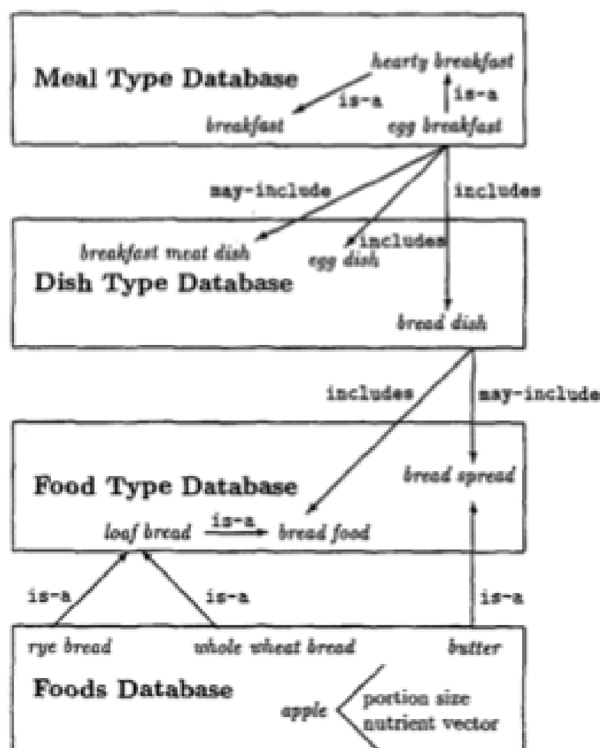
2.2 Il *Common Sense* per il *"Menu Planning"*

Una delle problematiche maggiori da risolvere in un ***ES*** per il *"Menu Planning"* è quella legata al *common sense*. Il suggerimento di cibi deve rispettare quelle regole di buon senso che una persona normalmente adotterebbe, per esempio "pasta con pomodoro" risulta essere un cibo appetibile, al contrario "pasta con banana" troverebbe poco riscontro di gradimento, oppure anche semplicemente un cibo al di fuori di una particolare pasto, per esempio "latte e biscotti per cena", quindi un sistema potrebbe anche essere molto efficiente nella pianificazione ma proporre cibi poco appetibili.
Nei sistemi sopracitati sono stati utilizzati vari approcci. Innanzi tutto bisogna distinguere tra sistemi esperti *rule-based* e *case-based*. Gli ***ES*** *case-based* non hanno questo problema[9], il *common sense* rimane intrinseco nei casi che l'Esperto del dominio fornisce per la costruzione della KB, non lavorando con regole e non generando dinamicamente pasti diversi ma applicando solo *case-matching* non potranno mai verificarsi casi in cui venga suggerito un piatto non appetibile, intuitivamente infatti l'***ES*** nemmeno è ha conoscenza di altre possibili combinazioni al di fuori di quelle che gli sono state fornite.
Il discorso cambia se si tratta di un ***ES*** *rule-based*, in questo caso le regole di creazione dei vari piatti (o pasti, in base al livello di astrazione) potrebbero associare cibi in maniera errata. In [8] il problema è stato risolto con una gerarchia di database (vedi Figura[2.1]). Sono stati utilizzati i così definiti *meal pattern*, che dal punto di vista computazione sono una sorda di ulte-

riore vincolo, in sostanza un pasto è composto da una serie di portate e una portata è formata da una serie di alimenti, per esempio:

Colazione:	carne	carne bianca
	frutta	pera
	pane/sostituti	biscotti
		burro
	bevanda	latte
Pranzo: ...		

Figura 2.1: Gerarchida di database di PRISM.[14]



In questo caso quindi il *common sense* è rappresentato dai *meal pattern* che dovrebbero soddisfare le aspettative di piatti o pasti ben formati.

Concludendo il problema del *"Menu Planning"* è stato largamente affrontato in letteratura, la sua prima formulazione risale agli anni della Seconda Guerra Mondiale, richiede particolare attenzione nella strategia adottata per la risoluzione del *common sense*, in quanto da esso dipende la qualità globale del sistema.
Quello che si è fatto in alcuni dei sistemi sopra citati e che è stato affarontato per questo elaborato, è lo sviluppo di un ***ES*** *rule-based* per risolvere il *"Menu Planning"*, attraverso quindi le tecniche di ***KE*** è stata acquisita e modellata la conoscenza di un'esperto del dominio (nel caso specifico un Dietologo) in modo che egli fornisse tutte le regole necessario per pianificare la dieta di una persona.

Capitolo 3

Progettazione di un Sistema Esperto per il *"Menu Planning"*

Nei capitoli successivi verrà mostrato lo sviluppo di un sistema per risolvere il problema del *"Menu Planning"* con un ***ES*** *case-based*, la conoscenza completa fornita dall'Esperto del dominio purtroppo è protetta da un accordo di non divulgazione, in quanto il sistema è parte di un progetto di *start-up* aziendale. In alcuni casi possono esserci delle sostanziali differenze tra le cose presentate e lo stato dell'arte del progetto, in quanto la conoscenza presentata in questo elaborato è molto ridotta rispetto a quella realmente utilizzata, le informazioni che seguono rispecchiano comunque il lavoro svolto. In particolare si prenderà in esame la costruzione di un menú utilizzando le regole fornite dall'Esperto del dominio.

Si utilizzerà molto spesso la parola *dieta*, è doveroso quindi disambiguare il significato, esiste infatti una differenze sostanziale tra la sua accezione classica e quella *moderna* adottata per questo elaborato, di seguito la definizione:

. . . , una prescrizione alimentare ben definita, in termini qualitativi e soprattutto quantitativi, mirante a correggere particolari condizioni cliniche a scopo terapeutico, preventivo o sperimentale[37], . . .

Si intende cioè una serie di suggerimenti che un Dietologo fornisce a fronte di una richiesta specifica del proprio paziente, che potrebbe voler dimagrire, aumentare di peso o semplicemente mantenere. In particolare nel sistema ristretto si presterà attenzione al caso del dimagrimento.

3.1 Dominio e Attori

Il dominio del sistema in esame è la *dieta* di una persona. Come visto nei capitoli precedenti, per la costruzione di un ***ES*** è necessaria l'interazione tra varie figure professionali, nel caso in questione due persone hanno contribuito: il Dott. Primo Vercilli nelle vesti di Esperto del dominio, ha fornito i requisiti del sistema e la conoscenza necessaria per il corretto funzionamento; il sottoscritto (Davide Dusi) che ha ricoperto i ruoli di Project Manager, Programmatore e Ingegnere della Conoscenza, ha eseguito sessioni di *expertise modeling* e analizzato i requisiti del sistema per costruire un prototipo funzionante.

3.2 Requisiti del Sistema

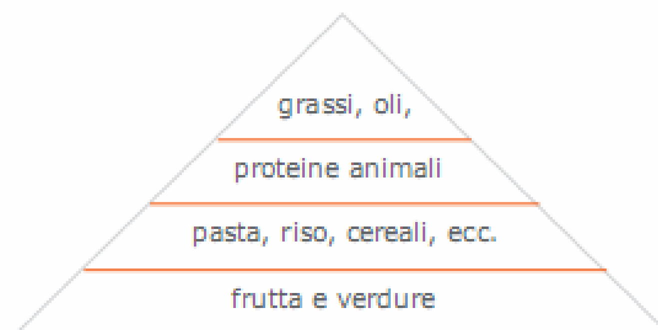
Il sistema deve provvedere alla pianificazione di un menú settimanale per una persona. In ingresso ha il menú abituale dell'utente, a questo dovranno essere eseguite le dovute variazioni per il corretto dimagrimento della persona, nel caso ridotto preso in esame solo l'eliminazione la classe di alimento

più calorica da un pasto, tenendo presente che una settimana media di una persona è composta da:

16 pasti con presenza di carboidrati
14-18 pasti con presenza di frutta
14 pasti con presenza di verdura
9 pasti con presenza di proteine animali, in associazione o meno a carboidrati

Le classi di alimenti vengono eliminate sulla base della loro posizione nella piramide alimentare (vedi Figura[3.1]), più un cibo è in alto ha probabilità di essere eliminato.
Una volta individuato il menú il sistema deve suggerire una serie di cibi possibili per uno certo *slot* in relazione alla classe alimentare al quale è associato, per dare all'utente la possibilità di vedere cosa può mangiare esattamente.[1]

Figura 3.1: Piramide alimentare.



3.3 Prima Analisi e Struttura del Sistema

Da una prima analisi dei requisiti si delineano due fai principali:

[1]Per maggiori informazioni sui requisiti utilizzati fare riferimento alla pagina: `http://www.primovercilli.it/index.php?option=com_content&view=article&id=13&Itemid=3`

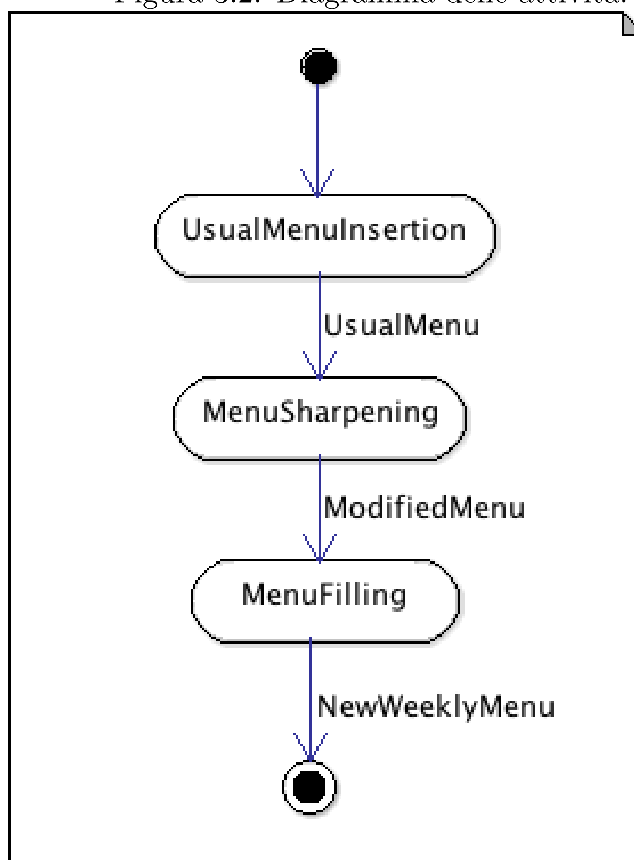
- ***Menu Sharpening***: La prima attività prende in *input* il menú settimanale dell'utente, applica le modifiche in base a delle regole fornite dall'Esperto del dominio (che formeranno la KB) e restituisce un *planning* settimanale.

- ***Menu Filling***: Prende in ingresso un menú settimanale modificato e si occupa di assegnare un cibo ad uno *slot* di un particolare giorno e pasto, in base alla classe alimentare che quello *stot* può contenere.

Il ***Menu Sharpening*** è stato risolto adottando un ***ES*** *case-based*, mentre il ***Menu Filling*** utilizzando un'***Ontologia*** di alimenti.
La Figura[3.2] mostra il diagramma delle principali attività, nella prima fase il *menu* viene modificato utilizzando le regole, fornite dall'Esperto, del ***ES*** *case-based*, una volta che si hanno a disposizione i pasti con i vari *slot* verrà interrogata l'***Ontologia*** degli Alimenti, questa fornisce una classificazione di tutti i cibi, quindi grazie ad una semplice consultazione per ogni *slot* è possibile determinare un cibo appetibile.

É necessario quindi un motore che metta in comunicazione le due attività e che le esegua nel loro corretto ordine. La Figura[2.1] mostra l'architettura generale adottata per risolvere il problema, il *meta-engine* si occupa di gestire il flusso di controllo primario, prende in ingresso gli input dell'utente, invia il menú da rielaborare all'*inference engine*, applicando le regole appropriate fornirà il menú modificato. Interrogando poi l'***Ontologia*** è possibile determinare quale alimento può essere inserito in ogni pasto. Riprendendo il nuovo paradigma di programmazione di Kowalski[30] si mostrerà prima la conoscenza emersa e utile al funzionamento del sistema, poi come è stato sviluppato il controllo all'interno del *Meta-Engine*.

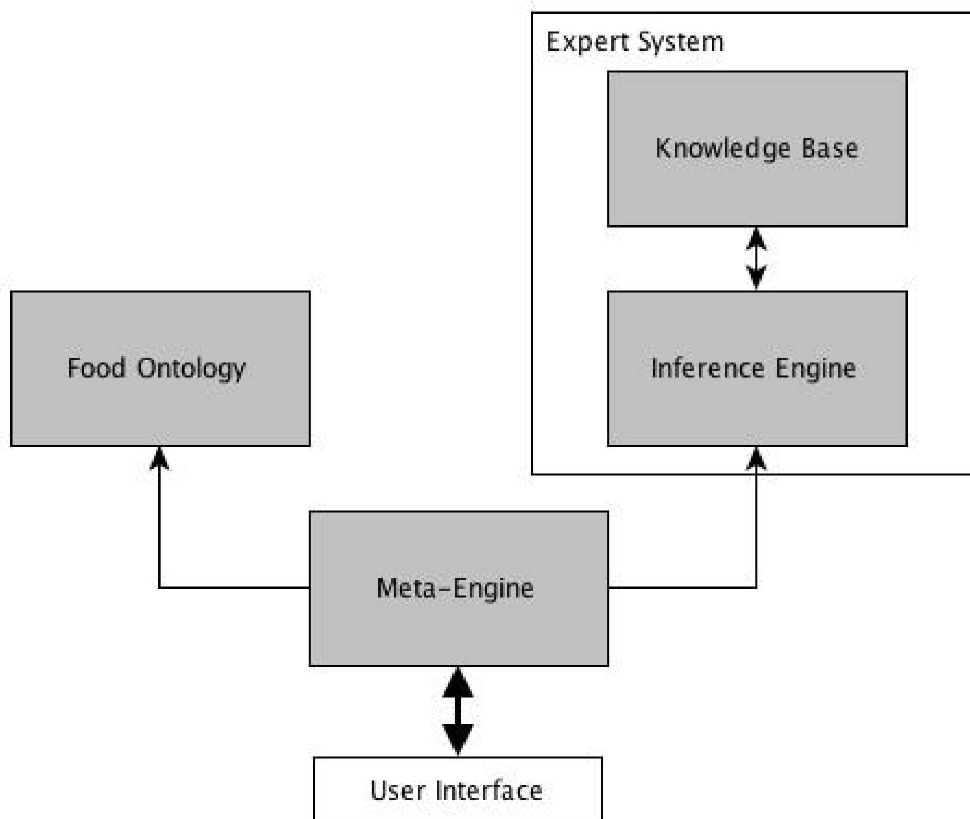
Figura 3.2: Diagramma delle attività.



3.4 Risoluzione del problema del *Common Sense*

Come detto in precedenza il problema del *common sense* può risultare determinate per la buona riuscita del sistema. L'idea che si adottata riprende vagamente le soluzioni viste in [8] e [10]. Un pasto è suddiviso in una certo numero di *slot* che corrispondono al numero di portate, all'interno di un singolo *Slot* di un certo pasto è ammessa una sola classe di alimenti. Quindi una settimana verrà divisa come segue:

Figura 3.3: Architettura del sistema.



	Lunedì	Martedì	...
Colazione	Slot1	...	
	Slot2		
Spuntino	Slot1	...	
	Slot2		
Pranzo	Slot1	...	
	Slot2		

Questo evita di avere pasti ambigui e associazioni non corrette, una colazione non potrà mai contenere "Pesce" in quanto i suoi due *slot* possono essere riempiti solo con *carboidrati e frutta*. L'intersezione dello *slot* con un certo *pasto* in una specifico *giorno*, fornisce una *coordinata univoca* che conterrà una classe di alimento.

3.5 Expertise Modeling

La fase di acquisizione ed elaborazione della conoscenza è una fase molto delicata, come già ampiamente illustrato nei capitoli precedenti. Sono state necessarie numero interviste con il Dietologo, Esperto del dominio, per comprende fino in fondo tutta la conoscenza necessaria ad entrambe le fasi di ***Menu Sharpening*** e ***Menu Filling***. Sostanzialmente quindi si può suddividere la conoscenza globale in tre parti, in particolare avremo:

- Ontologia del dominio per la condivisione della conoscenza
- Knowledge Base del ***ES*** per la fase di ***Menu Sharpening***
- Ontologia degli alimenti per la fase di ***Menu Filling***

Quindi ogni fase sfrutta la conoscenza corrispondente per offrire il *reasoning* adeguato e la struttura del sistema provvede al controllo.

3.5.1 Ontologia del Dominio

È stata necessaria la costruzione di un'***Ontologia*** per il dominio, è un prerequisito difficile ma doveroso per uniformare il vocabolario, decidere quali costrutti base utilizzare e come organizzarli è un punto fondamentale. Di seguito l'***Ontologia*** in DL e nella Figura[3.4] il diagramma delle classi relativo.

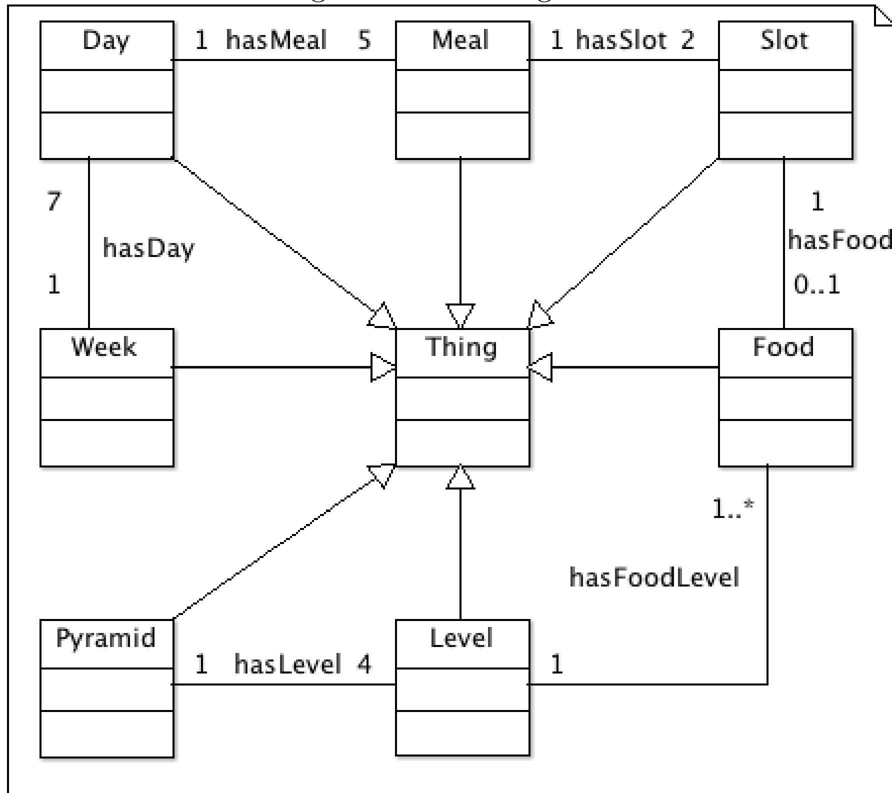
$Day(monday)$	$Week \sqsubseteq Think$
$Day(tuesday)$	$Day \sqsubseteq Think$
$Day(wednesday)$	$Meal \sqsubseteq Think$
$Day(thursday)$	$Slot \sqsubseteq Think$
$Day(friday)$	$Food \sqsubseteq Think$
$Day(saturday)$	$Pyramid \sqsubseteq Think$
$Day(sunday)$	$Level \sqsubseteq Think$
$Meal(breakfast)$	$Week \sqsubseteq \exists_{\geq 7}\ hasDay.Day \sqsubseteq \exists_{\leq 7}\ hasDay.Day$
$Meal(break1)$	$Day \sqsubseteq \exists_{\geq 5}\ hasMeal.Meal \sqsubseteq \exists_{\leq 5}\ hasMeal.Meal$
$Meal(lunch)$	$Meal \sqsubseteq \exists_{\geq 2}\ hasSlot.Slot \sqsubseteq \exists_{\leq 2}\ hasSlot.Slot$
$Meal(break2)$	$Slot \sqsubseteq \exists_{\geq 0}\ hasFood.Food \sqsubseteq \exists_{\leq 1}\ hasFood.Food$
$Meal(dinner)$	$Pyramid \sqsubseteq \exists_{\geq 4}\ hasLevel.Level \sqsubseteq \exists_{\leq 4}\ hasLevel.Level$
	$Level \sqsubseteq \exists_{\geq 1}\ hasFoodLevel.Food$

Questa ***Ontologia*** è stata utilizzata durante i colloqui per condividere informazioni tra Dietologo e Ingegnere della conoscenza, per capire se i requisiti erano stati compresi fino in fondo. Vengono infatti rappresentati i concetti: di *Settimana* che comprende sette i *Giorni*, ogni giorno è composto da cinque *Pasti*, il singolo pasto può essere suddiviso in due *slot* ad ognuno dei quali è assegnato una singolo *alimento*. Mostra inoltre la struttura della *piramide* alimentare, formata da quattro *livelli* ognuno dei quali può contenere svariati alimenti.

3.5.2 Knowledge Base

La KB rappresenta il cuore dell'***ES***, in essa è racchiusa tutta la conoscenza per generare un menú settimanale durante la fase di ***Menu Sharpening***. Tutte le regole su come un particolare *slot* deve essere riempito, sono state redatte dall'Esperto del dominio e rappresentate nella KB, per permettere il *reasoning* del sistema.

Figura 3.4: Ontologia del dominio.



Facendo riferimento a quanto detto sul *common sense* di seguito alcuni esempi delle regole utilizzate in ***FOL***:

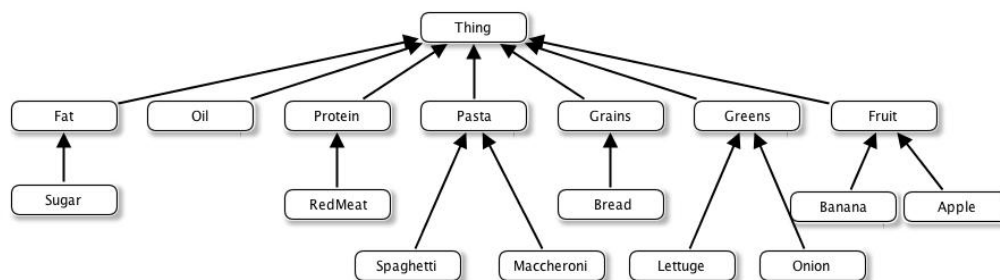
$$\forall x\ \exists d\ \exists m\ \exists s\ Food(x)\ \Rightarrow\ Slot(d, m, s, Food(x))$$
$$\forall d\ \forall m\ \exists x\ \exists y\ Slot(d, m, s1, Food(x))\ \wedge\ Slot(d, m, s2, Food(y)) \Rightarrow\ Meal(d, m, x, y)$$

Le variabili *d,m,s* rappresentano rispettivamente i *giorni*, i *pasti* e gli *slot*. La prima formula dice che ogni cibo deve essere almeno all'interno di uno *slot* in un particolare *pasto* di un certo *giorno*. La seconda invece rappresenta un *pasto* di un *giorno* all'interno del quale i due *slot* contengono un certo alimento.

3.5.3 Ontologia degli Alimenti

L'***Ontologia*** fornisce una tassonomia[2] di informazioni riguardanti concetti e affermazioni di senso comune[16].
È stata utilizzata in questo modo la conoscenza relativa agli alimenti durante la fase di ***Menu Filling***, permettendo così di ragionare sulle classi di alimento (all'interno dei singoli *slot*) per ricavare un cibo direttamente commestibile (sottoclasse).

Figura 3.5: Ontologia degli alimenti.



Nella fase di ***Menu Filling*** si scorrono tutti gli *slot* del nuovo menú settimanale si prendono le relative classi di alimento e si sostituiscono con un possibile cibo, nel caso ci siano più possibilità si offre la funzionalità all'utente di scegliere cosa preferisce. La gerarchia può essere espansa con ulteriori livelli di classificazione, per esempio fino ad arrivare ad un cibo di una determinata marca, si possono inoltre aggiungere proprietà di dato con le caratteristiche degli alimenti (e.g. kcal, grassi, etc.) in modo da fornire ulteriori informazioni di *reasoning*.

Riassumendo la conoscenza necessaria per il corretto funzionamento dell'intero sistema è suddivisa in tre blocchi principali, l'***Ontologia*** del dominio, la KB con le relative regole e l'***Ontologia*** degli alimenti, le ultime due in

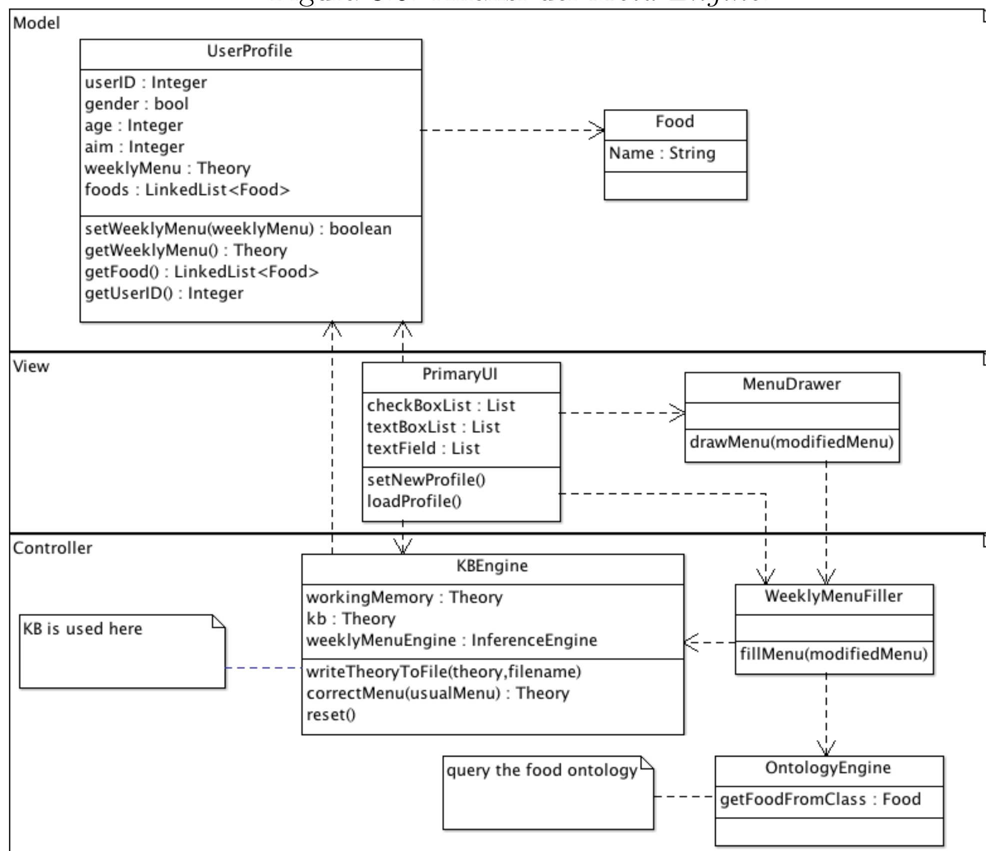
[2] "...studio teorico della classificazione ..."[37]

particolare offrono sostegno diretto al *reasoning* delle relative parti di ***Menu Sharpening*** e ***Menu Filling***.

3.6 Meta-Engine

Il motore principale del sistema rappresenta il *controllo* di tutta l'architettura. È stato adottato il pattern *Model-View-Controller* per analizzare la sua struttura, vedi Figura[3.6].

Figura 3.6: Analisi del *Meta-Engine*.



Il *modello* è composto dalla sola classe *UserProfile* che contiene tutte le informazioni riguardanti l'utente, quindi nome, l'ID, sesso, età, il menú modificato

(una volta che viene generato) e quello attuale. È possibile interagire con il sistema tramite l'*UI* principale, che fornisce all'utente la possibilità di inserire le proprie informazioni e quello che mangia abitualmente durante la settimana, inoltre l'interfaccia che visualizza il menú permette all'utente di avere una visione della propria settimana con i relativi pasti. Tramite l'interfaccia principale l'utente accede alle varie funzionalità e attiva il *KBEngine* e il *WeeklyMenuFiller*, rispettivamente per le fasi di ***Menu Sharpening*** e ***Menu Filling***.
Rispettando il *flusso il principale* si ha che il *KBEngine* si occupa di sottoporre *query* al motore inferenziale del ***ES***, in base al menú che l'utente fornisce le *query* saranno differenti, per esempio capire quale classe di alimento presente nel menú è da eliminare, sulla base di quello che l'utente ha inserito e delle informazioni fornite dalla piramide degli alimenti oppure quale classe di alimento si può inserire in un determinato *slot* di un *pasto* in un certo *giorno*. Il *WeeklyMenuFiller* invece si occupa di sottomettere all'*OntologyEngine query* da effettuare sull'***Ontologia*** [3], quindi riempie un menú con i relativi alimenti, ovvero quali alimenti si possono inserire in uno specifico *slot*, fornendo in uscita il menú completo.

Riprendendo l'architettura di un ***ES*** vista nel primo capitolo[14], essa comprende il database, il motore inferenziale, la memoria di lavoro e l'UI. Che si incastrano nell'architettura mostrata come segue:

- *KBEngine*: gestisce il **database**, **il motore inferenziale** e **la memoria di lavoro**.
- *WeeklyMenuFiller*: si occupa di tutte le operazioni *ontology-related*.
- *PrimaryUI* e *MenuDrawer*: sono la **user interface**.

[3]Nello schema è mostrata un'unica funzionalità, in realtà nel caso reale ne son presenti altre, tuttavia per non alterare troppo i contenuti la struttura è stata lasciata invariata.

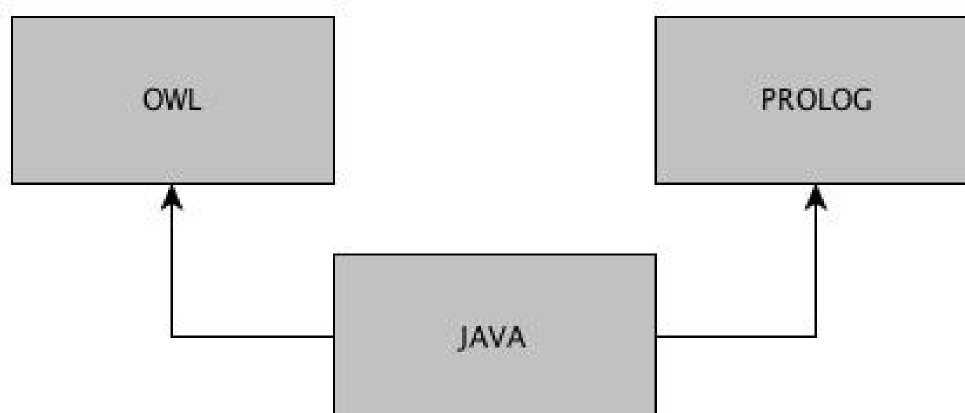
L'architettura mostrata offre potenzialità di *reasoning* molto elevate, la precisione di un ***ES*** *case-based* nell'applicare le regole e l'analisi semantica dell'***Ontologia*** ne fanno uno strumento molto adatto al *"Menu Planning"*.

Capitolo 4

Sviluppo del Sistema Esperto

In questo capitolo conclusivo verranno mostrate tutte le scelte riguardanti l'implementazione dell'architettura mostrata precedentemente, le tecnologie utilizzate per ogni componente e le problematiche affrontate.
Facendo riferimento al sistema nel suo complesso i linguaggi adottati sono: ***Java*** utilizzato per sviluppare il *Meta-Engine*, ***Prolog*** utilizzato per costruire il sistema esperto e ***OWL***-XML per la scrittura dell'***Ontologia*** degli alimenti, vedi Figura[4.1].

Figura 4.1: Progettazione del *Meta-Engine*.



Ricalcando il percorso seguito nel capitolo precedente, si mostrerà prima come è stata gestita la conoscenza, quindi ***ES*** e ***Ontologia***, e poi il controllo, ovvero il ***Meta-Engine***.

4.1 Sviluppo della Conoscenza

Uno dei punti fondamentali è stata la scelta della tecnologia da utilizzare per rappresentare la conoscenza, oltre ad essere il punto focale dell'intero sistema, essendo il progetto legato ad una *startup* aziendale son state fatte delle considerazioni riguardanti le licenze utilizzabili.

4.1.1 La Knowledge Base e il Motore Inferenziale

La KB è racchiusa in una teoria Prolog, le formule ***FOL*** viste nel capitolo precedente per rappresentare i *pasti* e gli *slot* vengono tradotte in **regole** Prolog. L'*implicazione* in questo linguaggio è data dall'operatore *":-"*, per esempio "Head :- Body" se la *Head* è vera allora lo è anche il *Body*. Date le formule:

$$\forall x\ \exists d\ \exists m\ \exists s\ Food(x)\ \Rightarrow\ Slot(d,m,s,Food(x))$$
$$\forall d\ \forall m\ \exists x\ \exists y\ Slot(d,m,s1,Food(x))\ \wedge\ Slot(d,m,s2,Food(y)) \Rightarrow\ Meal(d,m,x,y)$$

Per la sintassi dell'operatore ***Prolog***, i due operandi dell'implicazione ***FOL*** vanno invertiti, si avrà quindi:

```
% slot(?Day,?Meal,?Slot,?Food).
slot(D,M,S,food(X,Num)):-food(X,Num).

% meal(?Day, ?Meal, ?Slot1, ?Slot2).
meal(D,M,X,Y):-slot(D,M,1,food(X,Num)),slot(D,M,2,food(Y,Num)).
```

Le variabili *D* ed *M* sono degli indici interi, che indicano rispettivamente giorno e pasto, con la seguente codifica:

D = 1 for Monday
D = 2 for Tuesday
D = 3 for Wednesday
D = 4 for Thursday
D = 5 for Friday
D = 6 for Saturday
D = 7 for Sunday
M = 1 for breakfast
M = 2 for break1
M = 3 for lunch
M = 4 for break2
M = 5 for dinner

Quindi se si vuole rappresentare ad esempio "il pranzo del lunedì" si scrive:

slot(1,3,1,food(carbohydrates, 1)):-food(carbohydrates, 1).
slot(1,3,2,food(greens, 1)):-food(greens, 1).

Questo significa che in questo pasto dovremmo mangiare la prima occorrenza di carboidrati per lo *slot1* e la prima occorrenza di frutta per lo *slot2*. I termini *food(?Food,?Num)* rappresentano i fatti che compongono il menú di una persona, "Num" serve per diversificare i vari termini dei cibi, sono presenti all'interno della teoria tanti termini quanti sono le occorrenze di quel particolare cibo nel menú, questi sono asseriti dal *Meta-Engine*[1].

Come motore inferenziale è stata utilizzata una particolare versione *Java-based* di ***Prolog***, chiamata ***tuProlog*** [35][36]. È un progetto *opensource* rilasciato sotto licenza *LGPL*[2], questo motore ***Prolog*** è sviluppato dall'Università di Bologna e aggiornato dal gruppo di ricerca *aliCE* di Cesena.
tuProlog è stato progettato per essere uno degli elementi principali nelle applicazioni e infrastrutture Internet, per questo motivo presenta ottime

[1] vedi Sezione successiva.
[2] quindi utilizzabile come libreria anche per scopi commerciali.

caratteristiche di configurabilità dinamica, interazione con ***Java*** e interoperabilità. Inoltre il cuore di ***tuProlog*** è un oggetto ***Java*** che contiene solo le parti essenziali di un motore ***Prolog***, tutte queste caratteristiche lo hanno reso un ottimo candidato per questo progetto. In particolare grazie alla sua perfetta interazione con ***Java*** è stato possibile controllare il motore attraverso il *meta-engine*, utilizzandolo come semplice oggetto e fornendogli *query* in base alle necessità. La computazione avviene come qualsiasi altro motore ***Prolog***, descritta nel primo capitolo.

4.1.2 Sviluppo dell'Ontologia degli Alimenti

Per la creazione dell'***Ontologia*** degli alimenti è stato utilizzato l'editor *opensource*, basato su ***Java***, Protégé[3] il quale grazie alla sua estendibilità risulta essere una fessibile piattaforma per il *rapid prototyping* e *application development*. Per l'interazione ***Java-OWL*** si è fatto uso del *Semantic Web Framework opensource* **Jena**[4], utile alla costruzione di applicazioni basate sul web semantico, offre un ambiente programmatico per RDF, RDFS, ***OWL*** e SPARQL ed include un motore inferenziale rule-based.
Di seguito un estratto dell'***Ontologia*** in *OWL-XML*, si riferisce alla classe *Greens* che contiene come sotto-classi *Lettuge* e *Onion* come mostrato nel capitolo precedente.

[3]http://protege.stanford.edu/overview/protege-owl.html
[4]http://incubator.apache.org/jena/

```
        ...
<Declaration>
        <Class IRI="#Greens"/>
    </Declaration>
    <Declaration>
        <Class IRI="#Lettuge"/>
    </Declaration>
    <Declaration>
        <Class IRI="#Onion"/>
    </Declaration>
    <SubClassOf>
        <Class IRI="#Lettuge"/>
        <Class IRI="#Greens"/>
    </SubClassOf>
    <SubClassOf>
        <Class IRI="#Onion"/>
        <Class IRI="#Greens"/>
    </SubClassOf>
        ...
```

L'***Ontologia*** è stata sviluppata per esteso come mostrato nella Figura[3.5], inoltre nel caso concreto gli alimenti hanno anche delle *Data-Properties* che possono essere inserire per offrire maggiori informazioni di *reasoning*.

4.2 Sviluppo del Meta-Engine

Ricapitolando il sistema è composto dal ***ES*** sviluppato con ***Prolog***, l'***Ontologia*** scritta con Protégé e utilizzata da ***Java*** con le librerie **Jena** ed infine il cuore di tutta l'architettura, il *Meta-Engine* scritto in nel linguaggio ad oggetti ***Java***.
Questo permette un'ottima interazione tra tutte le parti in gioco finora presentate, tutte le classi presentate nell'analisi di Figura[3.6] sono state svilup-

pate. Per quanto riguarda il modello, la classe *UserProfile* contiene tutte le informazioni dell'utente compreso il *weeklyMenu* in una teoria di ***Prolog***. L'interfaccia primaria, *PrimaryUI*, è stata sviluppata utilizzando le librerie *SWING* di ***Java***, è sostanzialmente formata da una serie di *JLayeredPane* che contengono *JButton* e *JTextField* con i quali l'utente può immettere i propri dati, facendo poi l'*override* del metodo *public void actionPerformed(ActionEvent e)* è possibile discriminare i vari eventi[5].

4.2.1 Generazione di un menú

La classe *KBEngine* e *WeeklyMenuFiller* sono sicuramente i perni del sistema, a loro è demandato praticamente tutto il controllo del *Meta-engine*. Si prenda come esempio la generazione di un menú settimanale.

4.2.2 *Menu Sharpening*

Il *KBEngine* implementa l'***ES*** vero e proprio, contiene il motore inferenziale racchiuso nell'oggetto *Prolog*, la KB in un oggetto *Theory* ed infine anche la *workingMemory*. Questo *engine* si occupa di creare anche la teoria del menú, sia quello abituale che quello modificato. Le operazioni che esegue in ordine sono:

- Copia della KB nella *workingMemory*.
- Generazione dei fatti degli alimenti nella *workingMemory* sulla base delle informazioni nello *UserProfile*.
- Esecuzione di una *query* ***Prolog*** "meal(D,M,S1,S2)." per avere tutti i pasti possibili con gli alimenti.

[5]Quest'interfaccia è stata usata solo in fase prototipale, in realtà nell'attuale stato dell'arte del progetto si interagisce con il sistema attraverso pagine *JSP*.

- Generazione dei fatti “meal(?D,?M,?S1,?S2).” nella teoria del menú di tutte le soluzioni della *query* del passo precedente.

- Reset della *workingMemory* alla KB originale per ulteriori computazioni.

Seguendo il flusso principale, dopo la copia della KB avviene l’inserimento dei cibi come fatti nella teoria ***Prolog***, per fare ciò vengono utilizzati i termini ***Prolog*** *assert(+A)* e *retract(+A)*, questo permette di modificare dinamicamente la teoria, prendiamo ad esempio il caso in cui si devono inserire un numero *n* di cibi nella teoria del menú, si può utilizzare il seguente codice ***Java***:

```
for (int i=0; i<n;i++){
 weeklyMenuEngine.solve("asserta(food(fruit,"+(i+1)+")).\n");
}
```

Questo semplice ciclo permette l’inserimento nella teoria di *n* fatti *food(fruit,?X).*, in particolare si è utilizzato il termine *assetta(+A)* che permette l’inserimento dei fatti in cima alla teoria. Questo particolare stratagemma è stato creato perchè il motore ***Prolog*** per eseguire il *matching* scorre la teoria partendo dalla cima, in questo modo alla *query* *“food(fruit,X).”*, la prima unificazione che tornerà per la variabile *X* sarà il valore *n*, si hanno così le occorrenze di ogni cibo nella teoria ***Prolog*** e per sapere quelle relative ad una classe basta eseguire una *query* come appena mostrato, non dovendo creare particolari termini per ricercare il valore massimo all’interno di un insieme di fatti.
Eseguita la scrittura dei vari cibi è possibile ora interrogare ***Prolog*** sulle regole dei pasti definiti precedentemente, riprendendo la KB della sezione precedente si ha “il pranzo del lunedì” ed i relativi cibi definiti come:

slot(1,3,1,food(carbohydrates, 1)):-food(carbohydrates, 1).
slot(1,3,2,food(greens, 1)):-food(greens, 1).
food(carbohydrates, 1).
food(greens, 1).

Eseguendo nel motore ***Prolog*** la *query* “meal(D,M,S1,S2).” si avrà il seguente *matching*:

> yes.
> D / 1 M / 1 S1 / food(carbohydrates, 1) S2 / food(greens, 1)
> Solution: meal(1,1, food(carbohydrates, 1), food(greens, 1))

Quello che viene fatto a questo punto dal *KBEngine* è inserire questa soluzione nella teoria del menú, per esempio con un codice ***Java*** come questo:

```
alice.tuprolog.SolveInfo query=null;
query=workingEngine.solve("meal(D,M,S1,S2).");
weeklyMenuEngine.solve("assert("+
 query.getSolution().toString()+").");
```

Al termine dell’esecuzione di questo codice nella teoria del menú sarà presente il fatto *“meal(1,1, food(carbohydrates, 1), food(greens, 1)).”*, utilizzando questo esempio per ogni pasto è possibile ottenere una teoria ***Prolog*** completa di tutti i possibili pasti che possono essere fatti durante la settimana, da interrogare in qualsiasi momento per vari scopi, come ad esempio la visualizzazione nell’UI.

4.2.3 *Menu Filling*

A questo punto è necessario utilizzare l’***Ontologia***, la *OntologyEngine* si occupa di tutte le funzionalità legate all’***Ontologia***. Per ogni cibo nei vari *slot* dei pasti si ricercano le varie sotto-classi, fornendo così cibi appetibili all’utente. Si interroga la teoria del menú con una *query* del tipo “meal(D,M,food(F1,N1),food(F2,N2)).” in questo modo si può utilizzare il seguente codice per prendere i valori delle variabili “F1” ed “F2” che conterranno il nome della classe del cibo:

```
String food1 = queryMeal.getTerm("F1").toString();
String food2 = queryMeal.getTerm("F2").toString();
```

Si hanno ora a disposizione le classi degli alimenti di cui si vuole avere un a sostituzione con un cibo, è possibile interrogare l'***Ontologia*** con il seguente codice avendo così tutte le sotto-classi:

```
Iterator c1 =
 model.getOntClass(createOntURI(food1)).listSubClasses();
Iterator c2 =
 model.getOntClass(createOntURI(food2)).listSubClasses();
```

Model rappresenta un'istanza di una risorsa, in questo caso ***Ontologia*** degli alimenti, negli iteratori *c1* e *c2* si avranno tutte le sotto-classi che vengono utilizzate per dare all'utente una scelta su quale cibo utilizzare. A questo punto le scelte dell'utente possono o non possono essere memorizzate in una teoria, come mostrato nei precedente *step*, in base alle necessità.

4.2.4 Utilizzo della Piramide Alimentare

La procedura appena descritta spiega come utilizzare la KB per creare un menú, ora si vedrà come si può specificare e utilizzare la piramide alimentare. L'informazione che si vuole ricavare da questa struttura è sapere se un cibo è presente ad un certo livello della piramide, per poi eliminare quello più in alto od eseguire altre operazioni. Per definire un livello si può procedere come per gli *slot*, si faccia prima riferimento alla logica in ***FOL*** che è possibile scrivere come:

$$\forall x \exists p \; Food(x) \;\Rightarrow\; PyramidLevel(p, Food(x))$$

Ogni classe di alimento è all'interno di uno specifico livello p, in sintassi ***Prolog*** si definisce come:

```
%pyramidLevel(?P,?Food)
pyramidLevel(P,food(F,N)):-food(F,N).
```

Il termine sopra ritornerà *true* solo se nella teoria è presente almeno un alimento di quella classe, attribuendogli così un livello dato dalla variabile

P. Se si volesse definire la struttura della piramide vista nella Figura[3.1], si potrebbe scrivere come:

```
%pyramidLevel(?P,?Food)
pyramidLevel(1,food(fruit,N)):-food(fruit,N).
pyramidLevel(1,food(greens,N)):-food(greens,N).
pyramidLevel(2,food(pasta,N)):-food(pasta,N).
pyramidLevel(2,food(rice,N)):-food(rice,N).
pyramidLevel(2,food(grains,N)):-food(grains,N).
pyramidLevel(3,food(protein,N)):-food(protein,N).
pyramidLevel(4,food(fat,N)):-food(fat,N).
pyramidLevel(4,food(oil,N)):-food(oil,N).
```

A questo punto una volta che si sono inseriti i cibi nella teoria ***Prolog*** è possibile interrogare il motore inferenziale con una semplice *query* del tipo "pyramidLevel(L,F)." ed ottenere così in risposta la struttura piramidale in figura con le classi di alimento che si hanno a disposizione.

Concludendo si hanno le seguenti parti in tutta l'architettura:

- Meta-Engine: Scritto in ***Java***, gestisce il *main-stream* del programma, si occupa dell'interazione *Java-Prolog* (framework *tuProlog*) e dell'interazione *Java-OWL* (framework *Jena*).
- il Sistema Esperto composto da:
  - KBEngine: rappresenta un ***ES*** nella sua accezione classica, utilizza una teoria ***Prolog*** come KB e *working memory*, inoltre ha un motore ***Prolog*** da utilizzare come motore inferenziale.
  - Knowledge Base: scritta in ***Prolog***, caricata ed utilizzata dal *KBEngine*.
  - La memoria di lavoro: scritta in ***Prolog*** dinamicamente dal *KBEngine*.

  - Database dei fatti: scritto dinamicamente nella memoria di lavoro dal *KBEngine*, può essere salvato o meno in base agli usi, dall'oggetto *UserProfile* è comunque possibile ricostruire le classi di alimento abituali che si hanno a disposizione.

- Ontologia composta da:

  - Ontologia: scritta in ***OWL***-XML.
  - OntologyEngine: si occupa di tutte le operazioni legate all'***Ontologia***, come ad esempio fornisce possibili cibi a fronte di una certa classe di alimento.

Conclusioni

Oggigiorno il *"Menu Planning"* ha assunto un ruolo fondamentale per molte istituzioni, come ospedali e mese, per pianificare le spese su larga scala in relazione agli alimenti che si hanno a disposizione, inoltre è possibile offrire soluzioni automatizzate per seguire persone che necessitano di particolari diete e suggerire dinamicamente cosa dovrebbero mangiare.
In questa tesi è stata mostrata una nuova architettura che risolve il problema di una pianificazione settimanale di pasti. L'uso di tecniche per racchiudere la conoscenza all'interno di un sistema sono state basilari per la riuscita del progetto, le sessioni di *expertise modeling* insieme al Dott. Primo Vercilli, esperto del dominio, hanno rappresentato il fulcro per la buona riuscita dello sviluppo del sistema, grazie alle sue regole è stato possibile completare il quadro di funzionamento.
Il nuovo paradigma di programmazione introdotto da Kowalski, "Algorithm = Logic + Control", pone la fondamenta di questo intero progetto, e non solo. L'architettura presentata in questa tesi cerca di mettere insieme due metodologie di utilizzo della conoscenza, ***Ontologia*** e ***ES***. Paragonata ad un ***ES*** classico, il sistema presentato offre più versatilità, data dall'utilizzo dell'***Ontologia***.
La parte architetturale che ricopre il sistema esperto potrebbe anche essere *case-based*, non esistono particolari limitazioni, in ogni caso nessun approfondimento è stato portato avanti a riguardo.
Il sistema è attualmente in fase di sviluppo, lo stato dell'arte è ben più avanzato di quello mostrato in questo elaborato, ma ricalca ciò che è stato

ampiamente esposto. L'obiettivo è rendere l'architettura distribuita, fornendo un'interfaccia remota che comunica con il cuore del sistema e che gestisca più utenti alla volta, sarà probabilmente sviluppato utilizzando il *pattern factory*, per le classi che gestiscono l'***Ontologia*** e l'***ES***.
Tutto il progetto si incastra in una ben più vasta architettura per la gestione del benessere fisico di una persona, attualmente alla base di un progetto di *startup* aziendale. La parte della dieta, sviluppata in stretto contatto con il Dott. Primo Vercilli, darà la possibilità di seguire un innovativo metodo per la salute personale.

Appendice A

Appendice

A.1 Resolution Propositional Logic

Figura A.1: Pseudocodice dell'algoritmo di risoluzione.

```
function PL-RESOLUTION(KB, α) returns true or false
  inputs: KB, the knowledge base, a sentence in propositional logic
          α, the query, a sentence in propositional logic

  clauses ← the set of clauses in the CNF representation of KB ∧ ¬α
  new ← { }
  loop do
     for each Ci, Cj in clauses do
        resolvents ← PL-RESOLVE(Ci, Cj)
        if resolvents contains the empty clause then return true
        new ← new ∪ resolvents
     if new ⊆ clauses then return false
     clauses ← clauses ∪ new
```

A.2 Backward Chaining *FOL*

Figura A.2: Pseudocodice della concatenazione all'indietro.

```
function FOL-BC-ASK(KB, goals, θ) returns a set of substitutions
   inputs: KB, a knowledge base
           goals, a list of conjuncts forming a query (θ already applied)
           θ, the current substitution, initially the empty substitution { }
   local variables: answers, a set of substitutions, initially empty

   if goals is empty then return {θ}
   q′ ← SUBST(θ, FIRST(goals))
   for each sentence r in KB
          where STANDARDIZE-APART(r) = ( p1 ∧ ... ∧ pn ⇒ q)
          and θ′ ← UNIFY(q, q′) succeeds
      new_goals ← [p1, ..., pn|REST(goals)]
      answers ← FOL-BC-ASK(KB, new_goals, COMPOSE(θ′, θ)) ∪ answers
   return answers
```

A.3 Algoritmo di Ricerca Completo

Un algoritmo completo trova sempre una soluzione se esiste.

A.4 Euristica

Tecniche utilizzate nel *problem solving* per trovare soluzioni soddisfacenti.

A.5 Herbert Simon

Herbert Alexander Simon (June 15, 1916 - February 9, 2001) era uno scienziato, economista, sociologo e psicologo americano, padre fondatore di molti materie scientifiche, tra cui l'Intelligenze Artificiale.

A.6 Bottle-Neck

Fenomeno che si verifica quando le prestazioni di un intero sistema sono limitate da un numero limitato di componenti o risorse.

A.7 Fuzzy

Logica multi-valore, non come la logica Booleana in cui esistono solo *true* e *false*.

A.8 Licenza LGPL

Lesser General Public License è una licenza per il *software free* pubblicata dalla *Free Software Fundation (FSF)*.

A.9 Web Semantico

Si intende la trasformazione del *World Wide Web* in un ambiente dove tutti i documenti vengono associati ad uno specifico contesto semantico.

Bibliografia

[1] Balintfy JL. "Menu planning by computer". Communications of the ACM. 1964.

[2] Brown, R.M. "Automated menu planning". M.S. Thesis. Kansas State University, Manhattan, KS, USA, 1966.

[3] Eckstein, E.F. "Menu planning by computer: the random approach". J. Am. Diet, 1967.

[4] B. G. Buchanan, G. L. Sutherland, and E. A. Feigenbaum,Heuristic DENDRAL: A Program for Generating Explanatory Hypotheses in Organic Chemistry. In B. Meltzer and D.Michie,(eds), Machine Intelligence 4, Edinburgh: Edinburgh University Press, 1969.

[5] P. M. Soden and L. R. Fletcher,"Modifying diets to satisfy nutritional requirements using linear programming", British Journal of Nutrition, 1992.

[6] Hinrichs, T. "Problem Solving in Open Worlds: A Case Study in Design". Lawrence Erlbaum Associates, Hillsdale, NJ, 1992.

[7] Ganeshan K., Farmer J. "Menu planning system for a large catering corporation". In: Proceedings of the 3rd International Conference on the Practical Application of Prolog, Paris, France, 1995.

[8] G. J. Petot, C. Marling, and L. Sterling. "An artificial intelligence system or computer-assisted menu planning". Journal of the American Dietetic Association, 1998.

[9] A. S. Khan and A. Hoffmann. "Building a case-based diet recommendation system without a knowledge engineer". Artificial Intelligence in Medicine, 2003.

[10] B. Gaal, I. Vassanyi, and G. Kozmann, "A novel artificial intelligence method for weekly dietary menu planning". Methods of Information in Medicine, 2005.

[11] Edaward A. Feigenbaum. "Knowledge Engineering - The Applied Side of Artificial Intelligence". Heuristic Programming Project Computer Science Department Stanford University Stanford,California.

[12] Russel S. Norvig P. Artificial Intelligence: A Modern Approach (Third Edition), ed. Prentice Hall.

[13] Mendelson, Elliot. 1997. Introduction to Mathematical Logic. 4th ed. London: Chapman and Hall.

[14] Michael Negnevitsky. "Artificial Intelligence - A Guide to Intelligent System". (Second Edition), Addison Wesley, 2005.

[15] Church, Alonzo. 1956. "Introduction to Mathematical Logic". Princeton, NJ: Princeton University Press.

[16] F. van Harmelen, V. Liftschitz, B. Porter.Handbook of Knowledge Representation. Foundations of Artificial Intelligence, 1st ed. 2008, Elsevier.

[17] D. Nardi, R. J. Brachman. "An Introduction to Description Logics".

[18] F. Baader, W. Nutt. "Basic Description Logic"

[19] T.R. Gruber. “A translation approach to portable ontology specifications”. Knowledge Acquisition, 5:199-220, 1993.

[20] D. Lenat. “The dimensions of context space”. Technical report, CYCORP. URL: http://www.cyc.com/doc/context-space.pdf, 28 October 1998.

[21] R. Davis, H. Shrobe, and P. Szolovits. “What is a knowledge representation?”. AI Magazine:17-33, Spring 1993.

[22] R. Studer, V Richard Benjamins, D. Fensel. “Knowledge Engineering: Principles and methods”. 21 November 1997.

[23] F. Hayes-Rotb, D.A. Waterman, D.B. Lenat, “Building Expert Systems”. (Addison-Wesley, New York, 1983).

[24] W.J. Clancey. “Heuristic classification”. Artificial Intelligence 27 (1985) 289-350.

[25] Robert Kowalski. “The Early Years of Logic Programming” CACM. January 1988.

[26] Chitta Baral and Michael Gelfond. “Logic programming and knowledge representation Journal of Logic Programming”. 1994, Vol. 19, 73-148.

[27] J. McCarthy. “Programs with common sense”. In Proc. of the Teddington Conference on the Mechanization of Thought Processes, pages 75-91, London, 1959. Her Majesty’s Stationery Office.

[28] J. Lloyd. “Foundations of Logic Programming”. Springer Verlag, second edition, 1987.

[29] A. Colmerauer, H. Kanoui, R. Pasero e P. Roussel. “Un Systeme de Communication Homme-Machine e Francais”. Tecnical report, Groupe de Intelligence Artificielle Universitae de Aix-Marseille II, Marseille, 1973.

[30] R.A.Kowalski (July 1979). “Algorithm = Logic + Control”. Communications of the ACM.

[31] R. Akerkar, P. Sajja. “Knowledge-Based Systems”.

[32] George B. Dantzig. “The Diet Problem”, August 1990.

[33] George J. Stigler. “The cost of Subsistence”, University of Minnesota, 1945.

[34] George B. Dantzig. “Linear Programming and Extensions”. Princenton University Press, 1963.

[35] Enrico Denti, Andrea Omicini, Alessandro Ricci, “Multi-paradigm Java-Prolog integration in tuProlog”, Science of Computer Programming 57(2), Elsevier Science B.V., August 2005.

[36] Enrico Denti, Andrea Omicini, Alessandro Ricci, “tuProlog: A Light-Weight Prolog for Internet Applications and Infrastructures”, Practical Aspects of Declarative Languages, 3rd International Symposium (PADL 2001), Las Vegas, NV, USA, 11-12 March 2001. Proceedings. LNCS 1990, Springer-Verlag, 2001.

[37] Ist. Enciclopedia Italiana. Treccani. 2010.

Ringraziamenti

La lista delle persone da ringraziare si è sicuramente fatta molto lunga in questi sei anni di università. Probabilmente dimenticherei qualcuno vista la mia naturale propensione per la distrazione e non sono particolarmente bravo in queste cose, per questo inizio con un ringraziamento generale a chiunque abbia incontrato in questo percorso, persone che sono entrate ed uscite dalla mia vita e a tutte quelle che si sono trasformate in miei grandi amici.
Un grazie doveroso va alla mia famiglia che mi ha permesso di fare ciò che più mi piace al mondo e che mi ha sostenuto in tutte le scelte che ho fatto.
Ringrazio poi il mio relatore il Prof. Andrea Roli per avermi seguito in questo lavoro, per sostenere il nostro progetto Spinner e per avermi regalato la passione per questa materia.
Un grande grazie e abbraccio ad un grande professionista che ha permesso molte cose durante questi mesi di lavoro che ci ha offerto molte possibilità, ci ha accolto con niente in mano e ci ha dato la possibilità di credere in questo progetto, stò naturalmente parlando del prima amico e poi correlatore Primo, grazie per crede in tre ragazzi solo con la voglia di mettersi in gioco.
Vorrei dedicare un ringraziamento speciale alle seguenti persone, non perchè siano più importanti di altre, ma hanno contribuito alla stesura di questa tesi passivamente (sopportandomi. . .), grazie Valeria per avermi fatto le faccende di casa mentre stavo scrivendo questo elaborato, grazie Lele per tutte le serate passate a giocare a Play Station solo con lo scopo di distrarmi, mi dispiace tu non abbia mai vinto (in realtà non mi dispiace per niente, lo so in realtà una volta hai vinto ma avevo la mano rotta quindi non conta), grazie

Maja per ridere sempre alle mie battute, sinceramente non ho ancora capito chi sia più stupido dei due, se tu perchè ridi sempre o io che dico cavolate.
Grazie ad Enzo e Carmen, complimenti per la bellissima bambina.
Grazie a tutti quelli della palestra di Savignano, in modo particolare a Nicola che mi segue sempre negli allenamenti e nelle riabilitazioni (più spesso in queste). Grazie Giulio, Baby, Giorgio (grazie per i fumetti, finirò di sfruttarti prima o poi, . . . , forse), Claudio, Alberto, Sara e a quelli che non vengo più Marco, Massimo, Enrico. Grazie ai ragazzi dello Spinner Point di Cesena, grazie per aver creduto in noi e averci dato la possibilità di tentare questa avventura, vedremo di non deludervi, grazie Eleonora, Bicio, Claudio e Kristian. Grazie a tutti i ragazzi del bar Bordonchio, grazie a Cri (visto che nonostante le lezioni boicottate per altri scopi, chiamiamoli così, abbiamo finito), Rez (arrivo ai seminari, tranquillo), Fabio, Tello, Sapo, Scatto, G, Rabbit, Rilla e sicuramente mi stò dimenticando qualcuno.
Grazie agli amici ed ex-compagni di classe per tutte le cene di *gossip*, Lisa, Nico, Claudia, Vaso e Sara.
Un grazie generale a tutti ragazzi dell'Italica e a tutti quelli che non rientrano nelle categorie sopra citate, grazie Ale, Paolo, Luca, Manuel.
Un grazie a tutti quelli che mi son dimenticato, perdonatemi, siete importanti anche voi ma sono smemorato e questa è una cosa nota.
Ed infine *last but not least*: Tempo fa, insieme a due tra i miei migliori amici (è si stò parlando di voi Rudi e Piro...) decidemmo di fissare degli incontri dedicati alle nostre più grandi passioni, come l'informatica o l'elettronica. Questi incontri settimanali si svolsero in una casetta di legno a Bellaria, non era un garage nella Silicon Valley ma per noi era più che sufficiente. Ed ora siamo quì vincitori di una borsa di studio per una di quelle idee nate dal nulla, ma ve lo sareste immaginato? dove andremo a finire? ce la faremo? non ne abbiamo assolutamente idea. Ma una cosa è certa, ci stiamo mettendo tutti noi stessi e ci stiamo divertendo e questo è l'importante, ma sopratutto è un'ottima scusa per vedersi spesso e condividere le nostre passioni. Grazie di cuore ragazzi, grazie davvero.