Universita' degli studi di Genova

Facolta' di Ingegneria

Laurea Specialistica in Ingegneria Informatica

Sistema di Supporto Mnemonico

Giorgio Ravera

Relatore:
Chiar.mo Prof. Ing. Mauro Migliardi

Anno Accademico 2007/2008

Indice

Ringraziamenti

Nell'arco dei cinque anni in cui ho frequentato la Facoltà di Ingegneria dell'Università di Genova la mia vita è cambiata. Mi sono iscritto nel 2003 dopo aver terminato il liceo scientifico di Savona, di cui ho un pessimo ricordo, sia sul piano umano, sia su quello didattico. Da questo ho tratto forza e voglia di riscatto che mi hanno permesso di poter frequentare l'università con spirito di conoscenza e rivincita.

Qui sono rinato, imparando cosa sia l'impegno, la dedizione e l'amicizia di persone straordinarie che mi hanno dato speranza e gioia anche nei momenti più tristi. Da quando sono entrato in Openlab ho imparato cosa vuol dire responsabilità e impegno e ho cercato di ricambiare, a modo mio, la fiducia che molte persone hanno riposto in me. Essere stato presidente del club per tre anni è stata l'esperienza più bella. Non è sempre stato tutto facile. Le difficoltà mi hanno aiutato a crescere, a capire i miei limiti e a superarli. E' proprio a tutti i membri dell'Openlab che va il mio pensiero e ringraziamento per avermi sopportato e sostenuto, difeso e incoraggiato in ogni momento. Non dimenticherò mai i giorni passati insieme in aula e5 tra studio e sano divertimento, le discussioni su tematiche informatiche e non, i seminari da noi tenuti, le edizioni del Linux Day a cui abbiamo partecipato e i nostri interventi a fiere quali Tecno-Acqui e la mostra internazionale del cinema indipendente.

Tra tutti non posso fare a meno di citare le persone che sono state più presenti nel mio percorso e fondamentali: Alessandro (Cerve), Andrea, Nicolò, Emanuele, Andrea (Teddy), Fabio (il Vecchio), Alessandro (Lomba) e consorte, Simone, Davide (Fuia), Gabriele (Palma), Mauro (Nobile), Fabio (Igno), Andrea (Ros), Gianluigi, Tommaso, Stefano (Benve), Massimiliano e Sara. Un pensiero speciale va a Gianluca con il quale ho condiviso momenti imortanti, in particolare durante il tirocinio triennale.

Un ringraziamento speciale è dedicato ai miei amici di Savona e Genova con i quali ho passato serate molto belle e importanti: Pier, Orges, Isida, Matteo (Colo), Luca, Matteo (Pacchia), Diego (Giaddi), Valerio, Lara, Simone, Alessia, Alessandro (Suzzi), Cecilia, Alessandro (Delfo), Alessandro (Tixe) e alla mia compagnia estiva di Lonano: Ivan, Simone, Simone. Un pensiero particolare per Angela e Beatrice con le quali ho intrattenuto splendide conversazioni riflessive che mi hanno permesso di capire molti errori ed evitare di commetterli nuovamente ma, soprattutto, mi hanno tenuto compagnia in momenti difficili e mi hanno sempre motivato a completare i miei studi. Il ricordo dei momenti vissuti insieme è un elemento fondamentale che mi accompagnerà sempre.

Ringrazio anche il Dott. Stefano Lassi e Dott. Alessandro Pozzi per avermi permesso

di imparare moltissimo nell'ambito delle reti e per avermi dato sempre buoni consigli su come risolvere situazioni spiacevoli.

Un ringraziamento speciale va al Prof. Renato Zaccaria che ha sempre creduto in me, sostenuto e difeso. E' stato sempre presente negli ultimi anni dandomi consigli e lezioni di vita di cui ho fatto tesoro e mi serviranno negli anni futuri. Ringrazio anche i Prof. E. Giunchiglia, Prof. A. Tacchella, Prof. A. Armando, Prof. A. L. Frisiani, Prof. S. Zappatore, Prof T. Vernazza, Prof. G. Cainarca, Prof. A. Morro e Prof.sa C. Zordan e la manager didattica di informatica Dott.sa V. Resaz che sono stati dei riferimenti importanti e hanno avuto la pazienza di rispondere alle mie domande dandomi buoni suggerimenti nell'arco della mia carriera universitaria.

Negli ultimi tre anni, a partire dal tirocinio triennale fino a quello specialistico, la figura che è stata, di fatto, il mio maestro e riferimento, sia umano che didattico, è il Prof. Mauro Migliardi che ringrazio sia per la infinita pazienza dimostrata nel tempo trascorso a lavorare insieme, sia per tutti i suoi insegnamenti, senza i quali non avrei potuto terminare entrambe le tesi ed apprendere molti concetti del mondo dell'informatica, sia per la grande fiducia riposta in me. Ricorderò sempre, con molto piacere, le interessanti conversazioni svolte su argomenti di ogni genere, tutti i saggi consigli e gli aiuti che non mi ha fatto mai mancare. A lui va anche il merito di avermi mostrato il vero significato del termine *nerd* e, con orgoglio e fierezza, posso dichiarare di essere stato suo allievo.

Ringrazio anche mia zia Assunta, mia nonna Antonia, scomparsa nel 2005, con la quale sono cresciuto ed ho condiviso momenti splendidi, mio nonno Carlo che, pur non avendolo mai conosciuto ha rappresentato per me una figura importante da imitare, mio nonno Libero, mia nonna Anna e Nastasia.

Non posso fare a meno di ricordare mio padre, Giuseppe Ravera, scomparso nel 2005, che mi ha insegnato cosa vuol dire essere un Ingegnere e alcuni valori fondamentali quali l'onestà, la correttezza e l'umanità sia nei momenti trascorsi insieme, sia nel modo con cui ha affrontato la sua lunga e difficile malattia.

Infine ringrazio mia mamma, Anna Bonelli, a cui dedico l'intero testo. Mi è stata vicino sempre da quando sono nato, è stata un riferimento importante, capace di trovare il meglio di me e tirarlo fuori, anche in situazioni difficili dandomi l'appoggio e l'amore di cui avevo bisogno. E' stata una guida e un supporto in ogni situazione e, in particolare dalla scomparsa di mio padre, ha sacrificato molto per permettermi di arrivare a questo traguardo. Senza di lei non sarei la persona che sono oggi.

Obiettivo

In campo psico-fisiologico recenti studi correlano lo stress con la difficoltà a trasferire informazioni dalla memoria a breve termine a quella a medio termine. Questa difficoltà spesso ingenera una riduzione dell'efficienza personale poiche' le cose da fare vengono in mente in modo disordinato e non pianificato. La forma estrema è *Activity Thrashing* cioè l'incapacità di concludere alcunchè in quanto qualcosa di diverso da fare continua a venire in mente. E' nostra opinione che questo fenomeno potrebbe essere tenuto sotto controllo fornendo ai soggetti puntuali informazioni relative a quali attività possono essere efficientemente svolte nel suo attuale contesto.

Questo progetto si propone di studiare e sviluppare un sistema che analizzi il comportamento dell'utente per mezzo di canali quali parlato, gestualità e movimenti oculari e traduca questo comportamento in attività prioritarizzate. Queste attività verranno poi tradotte in interrogazioni ad un sistema informativo geografico per fornire all'utente indicazioni sulle attività che possono essere efficacemente svolte nel suo contesto.

Introduzione

In figura 1 è riportato lo schema dell'intero progetto. Come si può notare il software è

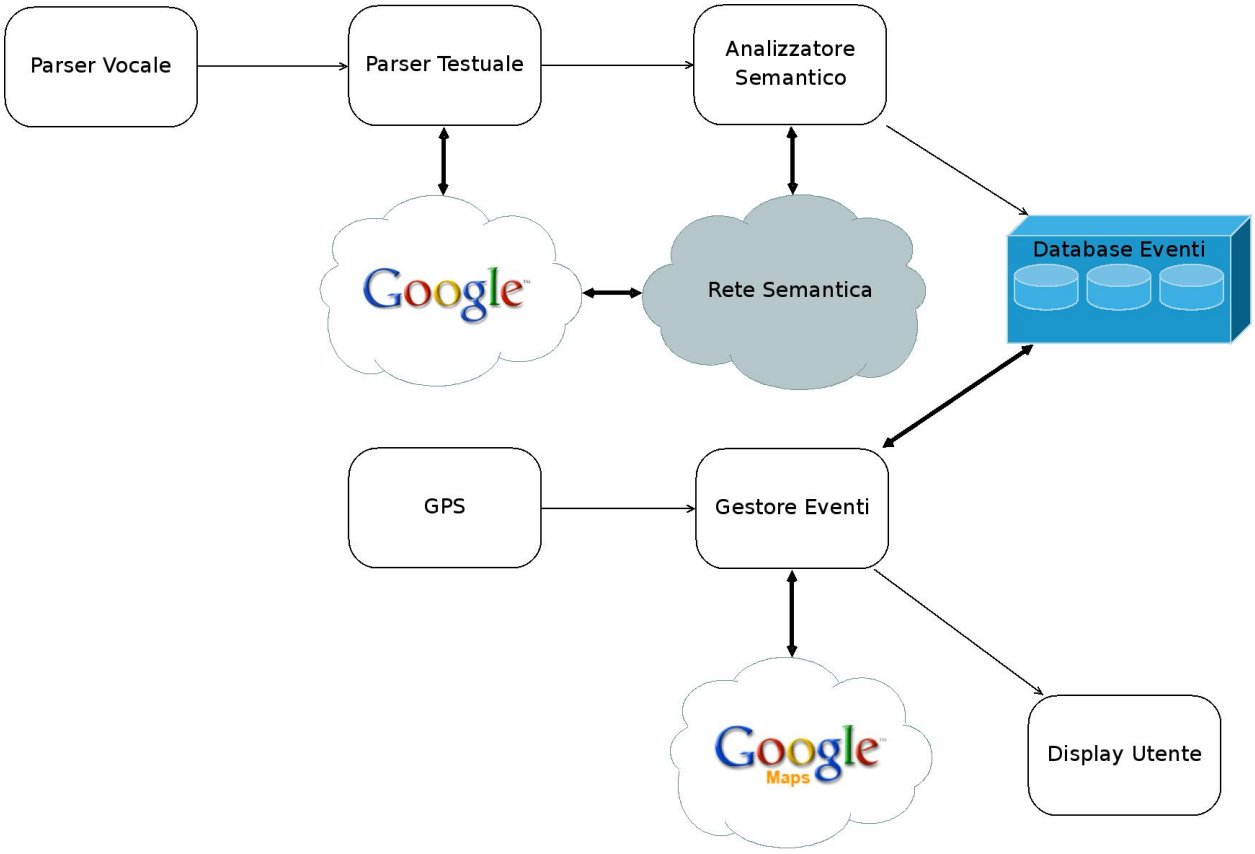

Figura 1: Schema del progetto

composto da diversi elementi:

- **Parser Vocale**: traduce la voce in uno stream di stringhe
- **Parser Testuale**: codifica le stringhe in istanze di classi per l'analisi semantica e, per ogni frase, identifica soggetto, verbo e complemento oggetto con eventuali aggettivi.

- **Analizzatore Semantico**: dato un albero semantico cerca di identificare l'informazione che trasporta. Utilizza la rete semantica per aumentare la potenza di analisi. Una volta identificati gli eventi da notificare, vengono caricati nel database.
- **Database di Eventi**: raccoglie tutti gli eventi indicizzati secondo criteri di notifica e priorità.
- **GPS**: Global Positioning System: identifica la posizione dell'utente e invia tali informazioni al gestore di eventi.
- **Gestore di Eventi**: data la posizione dell'utente provvede ad interrogare il database di eventi e Google Maps alla ricerca di un evento che merita di essere notificato e lo invia al display utente.
- **Display Utente**: si occupa di notificare all'utente che un particolare evento si è verificato.
- **Google**: viene utilizzato il motore di ricerca Google.it sia per identificare termini non riconosciuti dal parser, sia per aggiornare la rete semantica, sia per ottenere informazioni precise riguardanti la posizione dell'utente e possibili traiettorie.
- **Rete Semantica**: si occupa di identificare concetti in relazione a termini ad essi correlati.

Lo sviluppo di tale software ha reso necessario un approfondimento di teorie riguardanti l'intelligenza artificiale. Di essi se ne parlerà nella prima parte della tesi, in cui ci si focalizzerà in primo luogo sugli agenti, seguiti dalle tecniche di elaborazione del linguaggio naturale e, infine, dai sistemi di rappresentazione e gestione della conoscenza.

L'idea è quella di presentare prima le basi teoriche che stanno dietro all'intero progetto per poi averne riscontro durante l'analisi del codice. Tale approccio porta a una maggiore comprensione del lavoro svolto. L'ordine con cui vengono presentati gli argomenti è anch'esso pensato per introdurre il lettore al mondo dell'intelligenza artificiale gradualmente.

Parte I

Introduzione all'Intelligenza Artificiale

Capitolo 1

Agenti Intelligenti

1.1 Definizione di Agente

Un **agente** è un entità che agisce, cioé svolge un azione. Dev'essere in grado di percepire l'ambiente che lo circonda attraverso dei **sensori** ed eseguire delle azioni attraverso degli **attuatori**.

Un **agente intelligente** o **razionale** è un agente che agisce in modo da ottenere il migliore risultato o, in caso di incertezza, il migliore risultato atteso.

Consideriamo un esempio comune di agente: una persona umana. Quando un soggetto si trova a dover risolvere un problema per prima cosa analizza l'ambiente esterno, attraverso i sensori (occhi e orecchie). In secodo luogo prova ad elaborare, nella sua mente, un idea del problema e prova ad elaborare una soluzione ottimale correlando tra loro le informazioni in suo possesso (pregresse e acquisite). Infine, attraverso gli attuatori (braccia, piedi o, più in generale, muscoli) la applica.

A questa descrizione molto elementare possiamo ricondurre ogni singola attività, dalla soluzione di un problema matematico, alla scelta di quale canale televisivo guardare oppure semplicemente a quale strada percorrere quando si è davanti a un bivio.
Un agente è composto da due elementi fondamentali ed indivisibili:

- **architettura**: il sistema fisico (corpo umano, hardware di un calcolatore) che compone l'agente e gli consente di eseguire i calcoli.
- **programma**: un insieme di istruzioni che guidano l'agente nella risoluzione dei problemi e consentano di interpretare i segnali provenienti dal mondo esterno e ricodificare i risultati per controllare gli attuatori.

Esistono diverse tipologie di agenti che si differenziano in base alla struttura che implementano. Per ogni tipologia vi è un approccio teorico specifico.

Nei paragrafi successivi verranno approfonditi aspetti legati all'ambiente esterno e alla struttura di un agente.

1.2 Ambienti

Un agente interagisce continuamente con l'ambiente che lo circonda:

- preleva informazioni attraverso i sensori
- compie azioni attraverso gli attuatori

Con il termine **percezione** si indicano gli input che l'agente ottiene, tramite i sensori, dall'ambiente esterno in un dato istante. La **sequenza percettiva** è la storia completa di tutto ciò che l'agente ha percepito nella sua esistenza.

In generale la scelta dell'azione di un agente in un qualsiasi istante può dipendere dall'intera sequenza percettiva osservata fino a quel momento. Se possiamo specificare l'azione prescelta dall'agente per ogni possibile sequenza percettiva, allora abbiamo descritto l'agente in modo completo. Ciò implicherebbe avere una conoscenza completa dell'ambiente esterno e questo non è sempre possibile, come vedremo in seguito.

Si indentifica con **funzione agente** l'insieme di azioni che descrivono il comportamento dell'agente in risposta a ciascun elemento della sequenza percettiva. La si può vedere come una tabella che mette in corrispondenza di ogni percezione una delle possibili azioni. Nel caso non si conoscesse tale funzione è possibile ricavarla (interamente o solo parte di essa) provando tutte le possibili sequenze percettive e registrando il comportamento dell'agente.

La tabella appena descritta ha una validità *esterna* all'agente. Al suo interno ci sarà un **programma agente** che implementerà tale funzione. E' importante sottolinearne la differenza: la funzione è una descrizione matematica astratta, il programma è una sua implementazione concreta in esecuzione sull'architettura dell'agente.

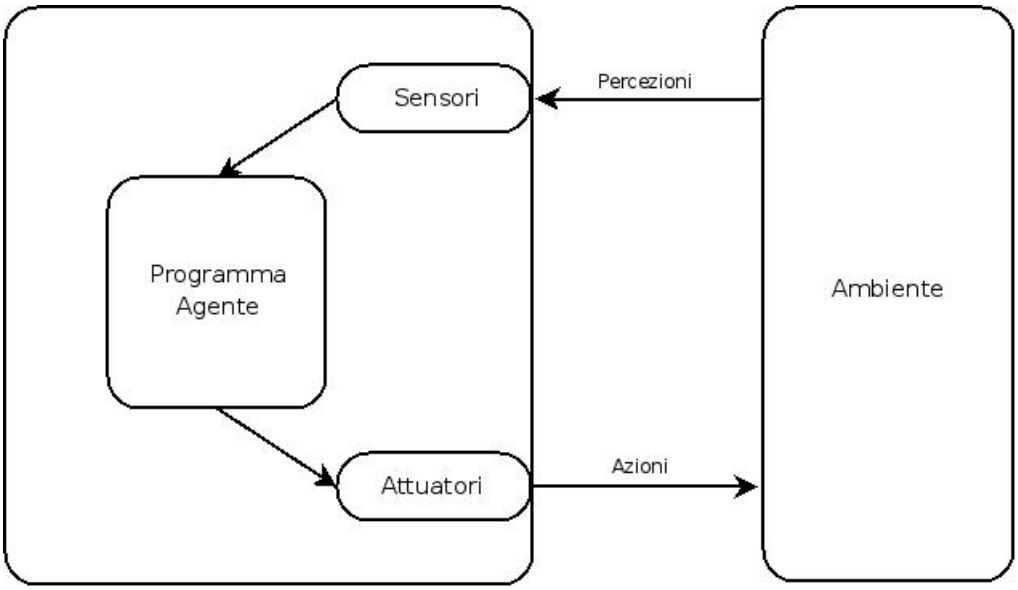

Figura 1.1: Schema di un generico Agente

La figura 1.1 illustra visivamente quanto detto fin ora: come si può vedere viene separato dall'ambiente esterno l'agente che è composto da sensori, attuatori e programma agente. Non è presentato un dettaglio di quest'ultimo per mantenere il più generica possibile la presentazione. Nel paragrafo 1.3 ne verranno mostrate le diverse implementazioni.

1.2.1 Tipologie di Ambienti

Per poter realizzare un agente con un dato obiettivo, è necessario definire l'ambiente che lo circonda. Da esso, infatti, l'agente trarrà tutte le informazioni necessarie per poter effettuare le decisioni e gli elementi per poterle attuare.

E', quindi, necessario identificare delle proprietà in base a cui suddividerli in categorie. Queste proprietà determinano in gran parte la progettazione di agenti appropriati e l'applicabilità delle principali tecniche alla loro implementazione.
Le principali proprietà sono le seguenti:

- **osservabilità completa/parziale**: caratterizza il grado d'accesso dei sensori al mondo esterno. Un ambiente parzialmente osservabile può essere dovuto a sensori inaccurati o imprecisi, alla presenza di rumore oppure all'impossibilità di ottenere alcune informazioni.

- **deterministico/stocastico**: nel caso di determinismo l'agente riesce a prevedere lo stato successivo dell'ambiente sulla base della misura fatta e dell'azione. Qualora ciò non fosse possibile si parla di ambiente stocastico. Nel caso in cui l'ambiente è deterministico ad eccezione delle azioni che altri agenti potrebbero compiere si parla di ambiente **strategico**.

- **episodico/sequenziale**: un ambiente si dice episodico se ogni azione generata non dipende dalle precedenti. Sequenziale nel caso in cui più percezioni influenzano la scelta dell'azione da svolgere.

- **statico/dinamico**: un ambiente è statico nel caso in cui non può cambiare nel periodo in cui l'agente ragiona al fine di prendere la decisione ottima. E' dinamico quando può variare durante la fase di ragionamento. Ciò comporta grossi problemi perché l'azione generata potrebbe non essere più quella corretta. Se l'ambiente non cambia con il tempo, ma la valutazione dell'agente si, allora si dice **semidinamico**

- **discreto/continuo**: tale classificazione è applicata allo *stato* dell'ambiente e a come viene rappresentato il *tempo* alle percezioni e azioni dell'agente. Può essere discreto, campionato ad intervalli regolari, oppure continuo, analizzato con continuità senza salti.

- **agente singolo/multiagente**: si parla di ambiente a singolo agente nel caso in cui eventuali altri elementi autonomi nell'ambiente non influenzano le scelte effettuate. Si parla di multiagente quando altri agenti possono partecipare attivamente al raggiungimento dell'obiettivo.

Si può capire facilmente che il caso più complesso è dato dalla combinazione di ambiente: *parzialmente osservabile, stocastico, sequenziale, dinamico, continuo e multiagente.*

1.3 Struttura di un agente

Come già visto in precedenza vale la seguente legge:

$$agente = architettura + programma \tag{1.1}$$

Con **architettura** si identifica la componente fisica, materiale dell'agente, ovvero l'hardware, i circuiti e le periferiche che lo compongono. E' composta prevalentemente da quattro differenti elementi:

- **strumenti di calcolo**: componenti che si occupano di elaborare i dati per relazionarli ed ottenere gli strumenti per poter effettuare le decisioni. In genere sono rappresentati dal/dai processore/i.
- **sensori**: analizzano l'ambiente esterno e codificano le informazioni raccolte in dati analizzabili dagli apparati di calcolo. Sono in genere microfoni, tastiere, videocamere o misuratori di temperatura.
- **attuatori**: componenti che ricevono istruzioni dagli apparati di calcolo e li traducono in azioni, come ad esempio il movimento di un braccio o la produzione di un segnale visivo/uditivo. Alcuni esempi possono essere display o casse acustiche.
- **memoria**: in alcuni agenti è presente la possibilità di memorizzare informazioni per poter avere più dati da tenere in considerazione nell'elaborazione di decisioni. Possono essere realizzati mediante database memorizzati su dischi o nastri.

Il **programma agente** è un software che deve essere in grado di:

- acquisire correttamente i dati provenienti dai sensori
- codificare tali dati per ottimizzare la funzione di decisione/ragionamento
- inviare agli attuatori azioni da svolgere compatibili con la loro progettazione

Come si può osservare dalla sua definizione, tale programma deve essere progettato per una specifica architettura e, quindi, risultare compatibile con i sensori e gli attuatori scelti.

E' importante sottolinearne la differenza tra il programma agente che prende come input solamente la percezione corrente e la funzione agente il cui input è costituito dall'intera storia delle percezioni. Il programma agente si basa sulla sola percezione corrente perché l'ambiente non può fornirgli nulla di più. Se le sue azioni dipendono dalla sequenza percettiva precedente è necessario utilizzare un sistema di memorizzazione.
Il seguente pseudocodice dovrebbe aiutare a capire la struttura base di un programma:

```
class agente-semplice
{
        map<perception, action> m;

        action calcola(perceptions p)
        {
                return m.lookup(p);
        }
}
```

Esistono sei tipologie diverse di agenti associate a una opportuna tipologia di programma agente che rappresentano i principi alla base di quasi tutti i sistemi intelligenti:

- agenti reattivi semplici
- agenti basati su modello
- agenti basati su obiettivi
- agenti basati sull'utilità
- agenti che apprendono
- agenti basati sulla conoscenza

1.3.1 Agenti Reattivi semplici

Un *agente reattivo semplice* è un agente che sceglie le azioni sulla base della sola percezione corrente, ignorando completamente quelle passate. In altre parole non ha memoria.

Un tale agente è, appunto, molto semplice da realizzare ma ha una intelligenza molto limitata. Si può schematizzare il suo operato attraverso il seguente pseudocodice:

```
class agente-reattivo-semplice
{
        map<state, list<rule>> rules;
        map<rule, action> actions;

        action calcola(perceptions p)
        {
                state s = code_percption(p);
                rule r1 = (rules.lookup(s, p)).getFirst();
                return actions.lookup(r1);
        }
}
```

Per prima cosa, viene codificata in forma astratta la percezione attraverso una sua elaborazione (funzione code_perception). Da qui si potrà ottenere uno stato dell'ambiente esterno. A questo punto è necessario interpretare lo stato e capire che **regola condizione-azione** attribuire a ciò che si è osservato. Nel caso in cui ci fossero più regole per lo stesso stato verrà considerata sempre e solo la prima. A questo punto si procede cercando l'azione da compiere.

Interpretiamo con un esempio quanto appena descritto. Consideriamo un sistema di guida autonomo: ci troviamo dietro ad un'automobile e dobbiamo capire quando questa frena per fare la stessa cosa. I sensori sono una serie di telecamere montate sul parabrezza, gli attuatori sono i freni. Ad intervalli regolari viene catturata un immagine: dall'immagine (percezione) si studiano le luci di segnalazione e in particolare se sono accese o spente (stato). In base al valore appena elaborato si cerca una corrispondenza tra questo e l'azione da svolgere. Si estrarrà, in caso di luci accese una regola condizione azione del tipo:

if (**light_on**) then **start_to_break**

Con questa regola diventa semplice capire che azione svolgere, nel caso corrente frenare.

Anche gli esseri umani fanno uso di connessioni analoghe, alcune basate sull'apprendimento, altre sui riflessi. Questi aspetti verranno approfonditi nei capitoli successivi.

Un tale agente funziona solo e unicamente nel caso in cui l'ambiente sia **completamente osservabile**. Anche una minima parte di inosservabilità potrebbe causare errori molto gravi. Nel caso in esempio la struttura potrebbe non prevedere la possibilità che le luci di segnalazione della macchina davanti non funzionino, oppure non differenziarli dalle luci di posizione. Limitandosi, quindi, alla sola analisi delle luci il sistema non funziona.

Infine un tale agente tende ad eseguire **cicli infiniti** se l'azione da compiere è sempre la stessa per ogni circostanza e lo stato non può mutare per peculiarità proprie dell'azione stessa. Supponiamo di avere un robot che deve muoversi in un terreno in relazione al colore delle piastrelle: se le piastrelle sono bianche si muove a destra o in basso, se sono nere a sinistra o in alto. A un tratto arriva in una piastrella nera con alla sua sinistra un muro. L'azione che dovrebbe essere eseguita sarà il movimento verso sinistra (prima regola) ma non sarà possibile e alla percezione successiva lo stato sarà analogo al precedente.

Per ovviare a questo problema è possibile scegliere in maniera casuale le azioni da svolgere. Ciò dovrebbe evitare cicli infiniti.

1.3.2 Agenti basati su Modello

Il modo più efficace di gestire l'osservabilità parziale, per un agente, è tener traccia della parte del mondo che non può vedere nell'istante corrente. Questo significa che l'agente deve memorizzare una sorta di stato interno che dipende dalla storia delle percezioni e che quindi riflette almeno una parte degli aspetti non osservabili dello stato corrente.

Ritornando al problema della frenata, lo stato interno non è troppo complesso: basta mantenere in memoria il fotogramma precedente scattato dalla telecamera, permettendo cosi all'agente di accorgersi quando due luci rosse, alle estremità laterali del veicolo, si accendono o spengono contemporaneamente.

Aggiornare l' informazione di stato al passaggio del tempo richiede che il programma agente possieda due tipi di conoscenza:

- informazioni sull'evoluzione del mondo indipendente dalle sue azioni
- informazioni sull'effetto che hanno sul mondo le azioni dell'agente stesso

Questa conoscenza sul funzionamento del mondo, viene chiamata **modello del mondo**. Un agente che si appoggia a un simile modello prende appunto il nome di **agente basato su modello**.
Si può schematizzare il suo operato attraverso lo pseudocodice sottostante:

```
class agente-reattivo-con-stato
{
        stete s;
        actions last_action;
        map<state, list<rule>> rules;
        map<rule, action> actions;

        action calcola(perceptions p)
        {
                s.update-state(p,last_action);
                rule r1 = (rules.lookup(s, p)).getFirst();
                last_action = actions.lookup(r1);
                return last_action;
        }
}
```

La parte interessante è la funzione update-state(), responsabile della creazione del nuovo stato interno. Oltre a interpretare la nuova percezione alla luce della conoscenza preesistente dello stato, la funzione utilizza le informazioni sull'evoluzione del mondo per tener traccia delle parti di esso che non sono presentemente visibili. La funzione deve anche conoscere l'effetto delle azioni dell'agente sullo stato del mondo.

1.3.3 Agenti basati su Obiettivi

Conoscere lo stato corrente dell'ambiente non sempre basta a decidere che cosa fare. Oltre che della descrizione corrente dello stato l'agente ha bisogno di qualche tipo di informazione riguardante il suo obiettivo. Il programma agente può unire quest'informazione a quella che riguarda i risultati delle possibili azioni (la stessa usata anche per aggiornare lo stato interno in un agente reattivo) per scegliere quelle che portano al soddisfacimento dell'obiettivo.

Talvolta scegliere un'azione in base a un obiettivo è molto semplice, quando questo può essere raggiunto in un solo passo. Altre volte è più difficile, quando l'agente deve considerare lunghe sequenze di azioni alternative per trovare il cammino che porta al risultato desiderato. La ricerca e la pianificazione sono dei sottocampi dell'Iintelligenza Artificiale dedicati

proprio a identificare le sequenze di azioni che permettono a un agente di raggiungere i propri obiettivi.

Notate che questo tipo di decisioni non ha nulla a che vedere con le regole condizione-azione che abbiamo descritto prima, perchè ora dobbiamo prendere in considerazione il futuro sotto due aspetti:

- cosa accadrà se si esegue l'azione prescelta?
- il risultato dell'azione sarà soddisfacente?

Nella progettazione degli agenti reattivi quest'informazione non viene rappresentata esplicitamente, perchè le regole interne mettono direttamente in corrispondenza percezioni e azioni. L'agente reattivo frena quando vede le luci di frenata. Un agente bassato su obiettivi, in via di principio, potrebbe ragionare che se la macchina davanti ha le luci di frenata accese starà rallentando. Dato il modo in cui evolve normalmente il mondo, l'unica azione che permetterebbe di raggiungere l'obiettivo di non tamponare altre macchine sarà allora quella di frenare.

Benchè un agente basato su obiettivi sembri meno efficiente, d'altra parte è più flessibile, perchè la conoscenza che motiva le sue decisioni è rappresentata esplicitamente e può essere modificata. Se comincia a piovere, l'agente può aggiornare la propria conoscenza circa l'efficienza dei propri freni. Nel caso dell'agente reattivo semplice sarà necessario riscrivere molte regole condizione-azione.

1.3.4 Agenti basati sull'Utilità

Nella maggior parte degli ambienti gli obiettivi, da soli, non bastano a generare un comportamento di alta qualità. Ci sono molte sequenze di azioni che porteranno al raggiungimento dell'obiettivo ma alcune potrebbero risultare migliori, in termini di costo, sicurezza o affidabilità, rispetto ad altre.

Gli obiettivi forniscono solamente una distinzione binaria tra stati soddisfacenti o meno, Occorre una misura di prestazione più generale permettendo di confrontare stati del mondo differenti e misurare precisamente il grado di contentezza che si otterrebbe se l'agente riuscisse a raggiungerli.

Una funzione di utilità assegna a uno stato (o a una sequenza di stati) un numero reale che quantifica il grado di preferibilità ad esso associato. Una specifica completa della funzione di utilità permette decisioni razionali in due categorie di casi in cui non bastano gli obiettivi:

- **conflitti**: quando ci sono più obiettivi da raggiungere che non si possono ottenere insieme. La funzione di utilità specifica quali privilegiare.
- **incertezza**: quando ci sono più obiettivi raggiungibili ma nessuno può essere ottenuto con certezza. Il concetto di utilità fornisce un mezzo per confrontare le probabilità di successo e l'importanza degli obiettivi.

Ogni agente razionale deve comportarsi come se possedesse una funzione di utilità di cui cerca di massimizzare il valore atteso. Un agente che possiede una funzione di utilità esplicita può quindi prendere decisioni razionali, e lo può fare seguendo un algoritmo generale che non dipende dalla specifica funzione di utilità che si desidera massimizzare. In questo modo la definizione globale di razionalità, che definisce razionali quelle funzioni agente che hanno le prestazioni migliori, viene trasformata in un vincolo locale in progetti di agenti razionali che possono esseri espressi in un semplice programma.

1.3.5 Agenti che Apprendono

Secondo Turing, in un articolo pubblicato nel 1950, è possibile costruire macchine capaci di apprendere e poi addestrarle. L'apprendimento presenta un enorme vantaggio: permette agli agenti di operare inizialmente in ambienti sconosciuti, diventando col tempo più competenti di quanto fossero all'inizio, allorchè si basavano sulla sola conoscenza iniziale. Un agente capace di apprendere può essere diviso in quattro componenti astratti:

- elemento di apprendimento
- elemento esecutivo
- elemento critico
- generatore di problemi

La distinzione più importante è tra: **elemento di apprendimento** (learning element), responsabile del miglioramento interno, e **l'elemento esecutivo** (performance element) che si occupa della selezione delle azioni esterne. Quest'ultimo è ciò che abbiamo considerato fin qui come se costituisse l'intero agente: prende in input le percezioni e decide le azioni. L'elemento di apprendimento utilizza informazione proveniente dall'elemento critico riguardo le prestazioni correnti dell'agente e determina se e come modificare l'elemento esecutivo affinchè in futuro si comportino meglio. Il progetto dell'elemento di apprendimento dipende molto da quello dell'elemento esecutivo. Quando si cerca di progettare un agente che impara a svolgere una certa attività, la prima domanda da porsi non è come faccio a fargli imparare questa cosa? ma quale tipo di elemento esecutivo permetterà al mio agente di fare questa cosa, una volta l'avrà imparata?. Qualsiasi progetto di agente può essere migliorato in ogni sua parte dall'aggiunta di meccanismi di apprendimento.

L'**elemento critico** dice a quello di apprendimento come si sta comportando l'agente rispetto a uno standart di prestazione fissato. Quest'elemento è necessario perchè le percezioni, in sé, non forniscono alcuna indicazione del successo dell'agente. Un programma di scacchi potrebbe ricevere una percezione che indica che ha dato scacco matto all'avversario, ma ha bisogno di uno standart di prestazione per sapere che questa è una cosa buona; la percezione da sola non basta. E importante che lo standart sia prefissato: concettualmente lo si può pensare come un'entità del tutto separata dall'agente, dato che quest'ultimo non lo deve modificare per adattarlo al suo comportamento.

L'ultimo componente di un agente capace di apprendere è il **generatore di problemi**, il cui scopo è suggerire azioni che portino a esperienze nuove e significative. L'idea è che se si lasciasse mano libera all'elemento esecutivo, esso continuerebbe a ripetere le azioni che ritiene migliori date le conoscenze attuali. Ma se l'agente è disposto a esplorare qualche altra possibilità, e magari eseguire nel breve termine qualche azione subottima, potrebbe scoprire l'esistenza di azioni molto superiori a lungo termine. Scopo del generatore di problemi è di suggerire tali azioni esplorative.

Applicando quanto detto al caso del taxi possiamo sostenere che: l'elemento esecutivo corrisponde alla collezione di conoscenze ed procedure usate dal taxi per selezionare le possibili azioni di guida. l'elemento critico osserva il mondo e passa delle informazioni a quello di apprendimento. Da esse l'elemento di apprendimento può formulare una regola che viene aggiunta all'insieme. A questo punto il generatore di problemi potrebbe identificare certe aree di comportamento che necessitano di qualche miglioria e suggerire esperimenti da effettuare.

Tale tipologia di agente è applicabile a tutti quelli analizzati in precedenza. L'elemento di apprendimento, infatti, può modificare uno qualsiasi dei comportamenti degli agenti appena descritti.

1.3.6 Agenti basati sulla Conoscenza

Gli agenti visti fin ora appartengono all'insieme degli agenti reattivi che operano attraverso il calcolo delle conseguenze delle azioni. Questo porta ad ottenere una conoscenza limitata non deduttiva della realtà. Consideriamo gli scacchi: un agente reattivo sa calcolare le mosse di ogni singola pedina e le conseguenze di ogni azione ma, in senso generale, non sa giocare a scacchi.

Gli agenti basati sulla conoscenza possono trarre beneficio da conoscenze espresse in forma generale, combinando e ricombinando le informazioni per adattarle a una varietà di scopi. Spesso, tale procedimento può non avere un legame diretto con le necessità immediate. La conoscenza la si può suddividere in due elementi fondamentali:

- **regole apprese**: procedure, informazioni e, in generale, conoscenza acquisita da uno studio o trasmessa da agenti o persone
- **schemi di associazione**: collegamenti tra le informazioni

Il componente più importante degli agenti basati sulla conoscenza è appunto la **base di conoscenza**, o KB (dall'inglese knowledge base). Informalmente, la base di conoscenza è costituita da un insieme di formule, espresse mediante un linguaggio di rappresentazione della conoscenza. Ogni formula rappresenta un'asserzione sul mondo.

La base di conoscenza deve prevedere meccanismi per raggiungere nuove formule e per le interrogazioni. I nomi standard per queste due azioni sono rispettivamente TELL (asserisci) e ASK (chiedi): entrambe possono comportare un processo di inferenza, ovvero la derivazione di nuove formule a partire da quelle conosciute. La risposta a ogni richiesta (ASK), posta alla base di conoscenza, sia una conseguenza di quello che le è stato detto

(attraverso TELL) in precedenza.
Si può schematizzare quanto detto attraverso lo pseudocodice sottostante:

```
class KB-agent
{
        database knowledge;

        action calcola(perceptions p)
        {
                knowledge.tell(p);
                action a = knowledge.ask(p);
                knowledge.tell(a);
                return a;
        }
}
```

L'agente mantiene in memoria una base di conoscenza, KB, che può contenere conoscenza iniziale (background knowledge). Ogni volta che viene invocato, il programma agente fa due cose

- comunica le sue percezioni alla base di conoscenza tramite la funzione TELL.
- chiede quale azione eseguire tramite la funzione ASK

Rispondere a questa domanda può comportare un esteso processo di ragionamento sullo stato corrente del mondo, le conseguenze delle possibili azioni e cosi via. Una volta che è stata scelta una azione l'agente la registra con TELL prima di eseguirla. Il secondo TELL è necessario per dire alla base di conoscenza che l'ipotetica azione è stata effettivamente eseguita.

I dettagli del linguaggio di rappresentazione sono racchiusi nelle tre finzioni che implementano l'interfaccia tra i sensori e attuatori da una parte e il sistema interno di rappresentazione e ragionamento dall'altra. Un agente siffatto si presta bene a essere descritto a livello di conoscenza, in cui per fissare il comportamento basta specificare solamente ciò che l'agente conosce e i suoi obiettivi. Ad esempio, un taxi automatico potrebbe avere l'obiettivo di portare un passeggero in Via Opera Pia 13, e potrebbe sapere di trovarsi a Genova. Allora ci aspetteremo che il taxi identifichi la via migliore per raggiungere la destinazione, perchè sa che cosi facendo raggiungerà il suo obiettivo.

Notate che quest'analisi è del tutto indipendente dal funzionamento del taxi a livello di implementazione: non importa se la sua conoscenza della geografia è realizzata con liste dinamiche o mappe di pixel, o se per ragionare manipola stringhe di simboli memorizzate in registri o propaga segnali in una rete neurale.

Si possono costruire agenti basati sulla conoscenza semplicemente dicendo loro quello che devono sapere. Il programma agente iniziale, prima di cominciare ad analizzare gli input sensoriali, dovrà aggiungere nella sua base di dati tutte le formule che rappresentano la conoscenza dell'ambiente circostante, opportunamente realizzate dal progettista.

Per fare ciò sarà necessario ricorrere ad un linguaggio di programmazione che rende facile esprimere la conoscenza per mezzo di formule. Tale approccio prende il nome di **approccio dichiarativo**. La base di conoscenza potrà essere ampliata attraverso una fase di apprendimento.

Capitolo 2

Elaborazione del Linguaggio Naturale

2.1 Comunicare informazioni

La **comunicazione** consiste nello scambio intenzionale di informazioni attraverso la produzione e percezione di **segni** appartenenti a un sistema convenzionale e condiviso. E' uno strumento attraverso il quale è possibile condividere informazioni.

Il **linguaggio** è il complesso sistema di messaggi strutturati che permettono a due o più entità di comunicare con chiarezza una parte della loro conoscenza. Solitamente un linguaggio prevede *regole sintattiche* e *semantiche* che consentono a due o più interlocutori di capirsi. Appare ovvio, da questa definizione che due o più agenti che vogliano comunicare tra loro devono condividere il linguaggio utilizzato.

Un agente, per produrre un linguaggio, compie un azione che prende il nome di **atto linguistico**. Con questo termine non si intende solamente la produzione di linguaggio parlato ma, più in generale, tutto ciò che può costituire una comunicazione. Ciò che viene prodotto è un insieme di segni comunicativi chiamati **parole**.

Un agente può intraprendere differenti tipologie di atti linguistici:

- interrogare altri agenti su aspetti dell'ambiente esterno
- informare altri agenti riguardo aspetti dell'ambiente esterno
- richiedere ad altri agenti di eseguire azioni
- acconsentire alle richieste
- promettere o impegnarsi di svolgere azioni
- dichiarare situazioni o proprietà del mondo esterno

Il riconoscimento della tipologia dipende dalla frase stessa. E' simile a qualsiasi altro **problema di comprensione**, come il riconoscimento di immagini o la diagnosi medica. L'agente riceve un insieme di input ambigui e deve procedere all'indietro per decifrarne il significato.

2.2 Fondamenti del linguaggio

Un **linguaggio formale** è definto da un insieme di **stringhe**, ciascuna costituita da **simboli terminali** chiamati **parole**. Hanno una sintassi molto rigorosa e sono facilmente interpretabili.

Un **linguaggio naturale**, invece, è definto da un insieme di stringhe costituite da **simboli terminali** ma non presentano una sintassi rigorosa. Sono i linguaggi utilizzati dalle persone per esprimersi.

Una **grammatica** è un insieme finito di regole che specifica un linguaggio. I linguaggi formali possiedono una grammatica ufficiale e ben definita mentre per i linguaggi naturali non è così.

Si possono approssimare linguaggi naturali a linguaggi formali imponendo una sintassi rigorosa basata su uno studio delle forme di comunicazione più comuni gestendo le eccezzioni attraverso aggiornamenti alla grammatica stessa.

Ad ogni stringa valida, ovvero che appartiene al linguaggio, è associato un significato, detto **semantica**. Nei linguaggi naturali è importante anche comprendere la **pragmatica** di una stringa, ovvero il suo significato in relazione al contesto (spazio-tempo) in cui viene prodotta.

I formalismi su cui si fondano le principali grammatiche sono basati sull'idea di **struttura sintagmatica**, secondo cui le stringhe sono composte da sottostringhe, chiamate **sintagmi** e suddivise in categorie. Tale suddivisione è utile per diverse ragioni. Innanzitutto i sintagmi corrispondono ad elementi semantici naturali da cui si può ricostruire il significato di un enunciato. In secondo luogo, una tale categorizzazione aiuta a descrivere quali sono le stringhe accettabili nel linguaggio.
Consideriamo ad esempio una frase:

Il re *è ricco*

Si può evidenziare chiaramente che la parte sottolineata corrisponde a un sintagma nominale, la parte in corsivo a un sintagma verbale.
Da ciò si può generalizzare che una frase è così ottenuta:

```
< frase > = <  sintagma nominale> < sintagma verbale >
```

Questa, di fatto, rappresenta una **regola di produzione** di una grammatica. I simboli in essa presenti sono detti **simboli non terminali**. Si possono intendere come dei segnaposto che dovranno essere sostituiti con altri simboli. A partire da un **assioma**, ovvero un simbolo non terminale, e quindi astratto, si iniziano ad applicare regole di produzione per ottenere, una frase sintatticamente corretta appartenente al linguaggio. Tale processo prende il nome di **derivazione**.

Si tenga presente che non tutte le frase sintatticamente corrette hanno un senso. In altre parle la correttezza sintattica non implica la correttezza semantica.

Le regole delle grammatiche possono essere utilizzate sia per l'**analisi** (accettare/rifutare stringhe, associare alberi che descrivono la composizione di stringhe) sia per la **generazione** di stringhe appartenenti al linguaggio.

2.3 Fasi della comunicazione

La comunicazione tra due agenti è suddivisa in diverse fasi. Ciascuna si occupa di realizzare una specifica funzione, superando i problemi che si possono generare. L'obiettivo è quello di inviare un informazione o una richiesta ad un altro agente.
Consideriamo due agenti A e B. A decide di intraprendere un atto linguistico con B per comunicare la frase:

il re è caduto

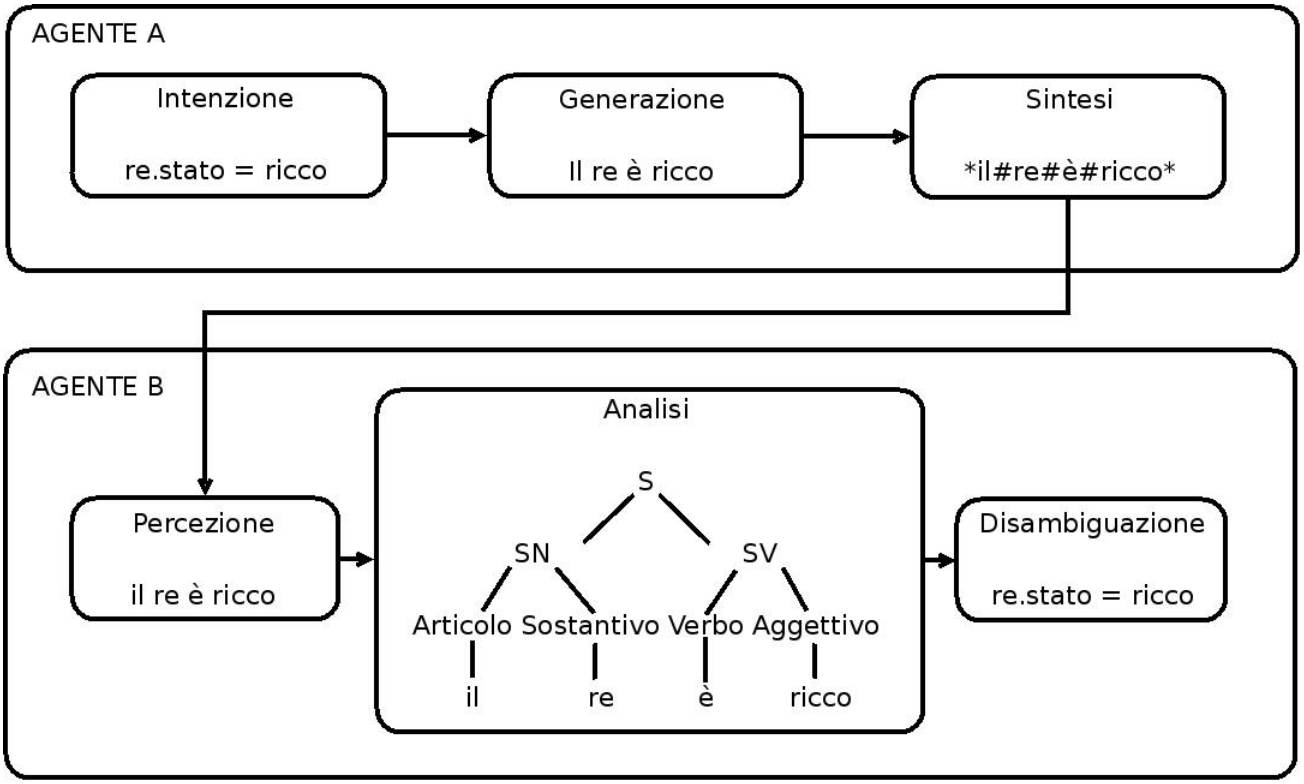

Figura 2.1: Schema di una comunicazione tra agenti

La figura 2.1 mostra schematicamente le fasi che si susseguono al fine di realizzare una comunicazione completa tra i due agenti:

1. **Intenzione**: l'agente A decide di comunicare con B.
2. **Generazione**: l'agente A trasforma l'informazione, codificata nella sua base dati in un enunciato che, una volta percepito dall'ascoltatore nella situazione corrente, con buona probabilità gli farà dedurre il significato.

3. **Sintesi**: il parlante produce la realizzazione fisica delle parole. Questo può essere fatto in svariati modi a seconda di ciò che gli attuatori consentono di fare. Ad esempio inchiostro su carta, vibrazioni nell'aria (suono), variazioni di luminosità o di tensione. Nell'esempio si aggiugono i simboli # e per segnalare il delimitarsi di una parola o di una frase.

4. **Percezione**: l'agente B, tramite i sensori, percepisce la realizzazione fisica delle parole e le decodifica. Se la comunicazione è vocale, tale passo prende il nome di **riconoscimento del parlato**, se è scritto **riconoscimento ottico**.

5. **Analisi**: l'agente B applica alla frase ottenuta la grammatica del linguaggio e procede alla costruzione dell'**albero sintattico** di cui parleremo nel paragrafo 2.4.

6. **Disambiguazione**: può capitare che alla stessa frase corrispondano più allberi sintattici diversi e quindi differenti possibili significati. In questi casi è necessario procedere a considerare gli aspetti pragmatici citati in precedenza. La scelta è guidata dalla probabilità con cui i diversi alberi possono corrispondere alla frase letta.

7. **Assimilazione**: una volta interpretato il significato di una frase, l'agente B può decidere se considerabile valida o meno la comunicazione, vale a dire aggiungere o meno l'informazione alla base di conoscenza oppure eseguire o meno l'azione richiesta.

Per completezza riportiamo di seguito una grammatica tale da riconoscere la frase del linguaggio vista in precedenza:

```
< frase > = < SN > < SV > |
                    < SV > < SN > |
                    < S > <Congiunzione > < S >
< congiunzione > = e | oppure | ma ...
< SN > = < Pronome > |
                 < NomeProprio> |
                 < Sostantivo > |
                 < Articolo > |
                 < Cifra > |
                 < SN > < PP > |
                 < SN > < PropCond >
< SV > = < Verbo > |
                 < SV > < SN > |
                 < SV > < Aggettivo > |
                 < SV > < PP > |
                 < SV > < Avverbio >
< PP > = < Preposizione > < SN >
< Preposizione > = di | a | da | in | con | su | per | tra | fra
< PropCond > = che < SV >
< Pronome > = io | me | tu | te | lui | lei | egli | noi | voi | essi | loro
```

```
< Articolo > = il | lo | la | i | gli | le
< Cifra > = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
< NomeProprio > = ... | re | ...
< Aggettivo > = ... | caduto | ...
```

2.4 Analisi Sintattica

L'**analisi sintattica** (o **parsing**) è il processo atto ad analizzare uno stream continuo in input (letto per esempio da un file o una tastiera) in modo da determinare la sua struttura grammaticale.

Si può pensare al parser come a un agente software che ha per input un file di testo e produce in output un albero sintattico. La costruzione di tale albero dipende dalla grammatica che si utilizza per il riconoscimento del testo.

Il *parsing* può essere considerato un processo di ricerca di un albero sintattico. Ad ogni passo, vengono, infatti, esplorate le regole di produzione fornite dalla grammatica al fine di scegliere quella più opportuna da utilizzare. Esistono due metodologie nella scelta delle regole da utilizzare ad ogni passo di derivazione:

- **derivazione canonica sinistra**: si sceglie sempre di sviluppare il simbolo non terminale più a sinistra. Tecnica più usata perché riflette il metodo di lettura più diffuso (da destra verso sinistra).
- **derivazione canonica destra**: si sceglie sempre di sviluppare il simbolo non terminale più a destra.

Scegliere una delle due tecniche è fondamentale perché evita la creazione di alberi sintattici errati e migliora l'efficienza del parsing stesso.

Le grammatiche che si utilizzano per la rappresentazione della sintassi sono **grammatiche non contestuali** (CFG) che si differenziano dalle altre per la tipologia di regole di derivazione. Queste ultime dovranno avere la seguente caratteristica: *la parte sinistra della regola deve essere un unico simbolo non terminale e la parte destra una sequenza qualsiasi di simboli terminali e non senza vincoli sulla loro disposizione.*

Esistono due diversi approcci nella realizzazione di un parser che dipendono dal modo con cui viene costruito, durante l'analisi dell'input, l'albero sintattico:

- **top-down**: l'albero viene costruito a partire da un simbolo non terminale, assioma, e si aggiungono nodi in relazione alla parola dell'input puntata. Per ogni simbolo terminale che si andrà a trovare nelle regole di produzione si sposterà il puntatore di lettura sulla parola successiva.
- **botton-up**: l'albero viene costruito leggendo un simbolo terminale per scegliere quale regola di produzione utilizzare per creare i nodi dal basso verso l'alto fino ad arrivare al nodo radice, l'assioma.

La realizzazione del parser è una semplice applicazione di sintesi di codice. Il suo funzionamento è intrinsecamente legato alla definizione della grammatica stessa. Esistono molto tool in grado di generare un parser dalle specifiche della grammatica.
I più comuni sono:

- antlr
- javacc
- bison
- flex
- yacc
- lex

2.4.1 Parsing Top-Down

Un parser **top-down** è un parser che costruisce l'albero sintattico a partire dalla radice. Lo può pensare come un software di ricerca che opera nel seguente modo:

1. **stato iniziale**: si parte da un albero avente un unico nodo, l'assioma della grammatica. Lo stato è rappresentato da un albero sintattico che si evolve nel tempo.
2. **funzione successore**: sceglie nell'albero il nodo più a sinistra da sviluppare (ovvero un codo corrispondente a un simbolo non terminale senza figli), quindi cerca le regole nella grammatica che hanno come parte sinistra tale nodo. Per ogni regola trovata la funzione vengono creati tanti nodi figli quanti sono gli elementi nella parte destra della regola. La scelta della regola è un proglema molto importante e dipende da differenti fattori.
3. **test obiettivo**: verifica se le foglie dell'albero sintattico corrispondono esattamente alla stringa in input.

2.4.2 Parsing Botton-Up

Un parser **botton-up** è un parser che costruisce l'albero sintattico a partire dalle foglie per arrivare al nodo radice. Lo può pensare come un software di ricerca che opera nel seguente modo:

1. **stato iniziale**: si parte dalla lista di parole che compongono la stringa di input. Lo stato, in questo caso, è rappresentato da una lista di alberi sintattici che dovranno convergere tutti al nodo radice.

2. **funzione successore**: considera la posizione i-esima nella lista di alberi e la parte destra di ogni regola di produzione. Se la sottosequenza della lista di alberi che comincia con i corrisponde alla parte destra, viene sostituita da un nuovo albero la cui categoria è la parte sinistra della regola e i cui figli sono la sequenza stessa.

3. **test obiettivo**: verifica se le foglie dell'albero sintattico corrispondono esattamente alla stringa in input.

2.4.3 Problematiche del parsing

Esistono alcune problematiche riguardati l'analisi sintattica che possono portare a seri problemi:

- inefficienza
- ambiguità
- ricorsione a sinistra

Efficienza del parsing

Entrambe le tipologie di parser analizzate possono risultare inefficienti a casua di molteplici modi in cui si possono combinare più analisi sintattiche di sintagmi differenti. E' possibile perdere tempo cercando in porzioni non rilevanti dello spazio di ricerca. Il parsing top-down può generare noti intermedi che non potranno mai legare con nessun simbolo terminale nella stringa di input, mentre il botton-up può generare sottoalberi che non si potranno unire tra loro per arrivare all'assioma.

Questo aspetto appare evidente se si pensa al **backtrace**. Nel caso in cui venga provata una derivazione e si arrivasse a un simbolo terminale differente da quello puntato in input, è necessario tornare indietro e provare un altra regola di produzione.

Esiste una tecnica di parsing che si ispira alla **programmazione dinamica** che può migliorare l'efficienza dell'analisi. Il concetto è il seguente: *ogni volta che si analizza una sottostringa, si memorizzano i risultati in modo da non doverla rianalizzare in seguito.* Questa tecnica prende il nome di **chart parser**.

Ambiguità

Si genera un **ambiguità** quando alla stessa stringa di input corrispondono più alberi sintattici e quindi più interpretazioni semantiche diverse. Il problema non dipende dal parser ma dalla grammatica che viene utilizzata. Una grammatica è ambigua se possiede più derivazioni canoniche (destre o sinistre) per la stessa frase.

L'ambiguità di una grammatica è una **proprietà indecidibile**, pertanto non è possibile costruire un algoritmo che identifichi automaticamente se una grammatica è o no ambigua e l'eliminazione delle ambiguità dalla grammatica è un procedimento necessariamente manuale. Consideriamo il seguente esempio:

```
input: aaa

< A > = < A > < A > | a
```

Tale grammatica è ambigua. Dallo stesso input (aaa) si possono estrarre due alberi sintattici (destra e sinistra). Si può riscrivere la grammatica per renderla non ambigua e ottenere la seguente:

```
< A > = < A > < B > | < B >
< B > = a
```

oppure

```
< A > = < B > < A > | < B >
< B > = a
```

Ci sono casi in cui non si possono eliminare le ambiguità. Si utilizzano **grammatiche non contestuali probabilistiche** (PCFG) in cui si pesano con le probabilità le regole di produzione. La scelta dell'albero sintattico, quindi, è un applicazione di un problema probabilistico. E', infine, possibile modificare le probabilità attraverso tecniche di apprendimento.

Ricorsione a sinistra

Un problema tipico dei parser top-down è la ricorsione a sinistra. Questo può portare a loop infiniti dal quale il parser non potrà più uscire.
Consideriamo le seguenti regole:

1. `< A > = < A > x | y`
2. `< B > = < C >`
3. `< C > = < B > | y`

Consideriamo la prima regola: se si applica la procedura di derivazione canonica a sinistra il processo continuerà a sostituire la regola con se stessa senza uscire da questo ciclo infinito. Questa ricorsione prende il nome di **ricorsione diretta**. Le regole 2 e 3, invece, analizzate da sole non generano ricorsione infinita ma, quando la 2 viene applicata, dovrà essere applicata la 3 successivamente, che a sua volta richiamerà la 2. Questa ricorsione prende

il nome di **ricorsione indiretta**.
Rimuovere le ricorsioni è facile e si può applicare un algoritmo che va a modificare le regole di produzione come segue:

```
< A > = < A > x | y

si trasforma in:

< A > = y < B >
< B > = x < B > | < NULL >
```

2.5 Analisi Semantica

L'analisi semantica ha il compito di ha il compito di attribuire un significato all'albero sintattico ottenuto. In altre parole deve trovare un significato degli enunciati.

I significati, ovvero i sensi, spesso sono rappresentati tramite collezioni di sinonimi o synset. Un synset definisce un concetto noto all'uomo e classificato in una gerarchia di concetti. Un concetto può essere espresso tramite una o piu parole, note come lessicalizzazioni.

Attribuendo ai vari synset i codici univoci, si può classificare i concetti in una struttura reticolare con le relazioni, detta ontologia, ed arrivare a traduzione automatica che permette di passare da una lessicalizzazione ad altra.

La semantica è direttamente collegata con la rappresentazione della conoscenza, di cui parleremo nel capitolo 3. La si può vedere come una fase di traduzione volta a convertire un albero sintattico nel formalismo con cui vengono rappresentate le informazioni, obiettivi e azioni nell'agente stesso.

Spesso, l'analisi semantica contribuisce alla generazione di linguaggi attraverso l'uso di **grammatiche aumentate** o grammatica a clausole definite (DCG), che si occupano di etichettare le regole per poterle scegliere correttamente al fine di generare una frase che, oltre ad essere corretta sintatticamente, lo è anche semanticamente. Se ad esempio si volesse generare:

Io voglio il budino

si applicherebbe una struttura grammaticale simile alla seguente:

```
< pronome > < verbo > < nome >
```

Budino è chiaramente un nome e ne indica l'oggetto desiderato. Il verbo è espresso alla prima persona ed implica una volontà, quindi si sceglie voglio ma per il pronome? Qual'è la differenza tra Io e Me? Il primo è utilizzato in caso di soggetto, il secondo in caso di oggetto. Si potrebbe scegliere di utilizzare regole aggiuntive ma ciò renderebbe molto complicata la grammatica. L'alternativa è quella di effettuare annotazioni sulle regole stesse. Riprendendo la grammatica vista in precedenza, si modificherebbe come segue:

```
< frase > = < SN(Soggetto) > < SV > | ...
< SN(Type) > = < Pronome(Type) > |
                  ...
                  < SN > < PP(Type) >
< SV > = < Verbo > |
                  < SV > < SN(Oggetto) > |
                  ...
                  < SV > < PP(Oggetto) >
< PP > = < Preposizione > < SN(Oggetto) >
< Pronome(Soggetto) > = io | tu ...
< Pronome(Oggetto) > = me | te ...
```

Un metodo alternativo è rappresentato dagli **schemi di traduzione** che rappresentano una notazione per collegare particolari azioni, sotto forma di istruzioni, in un qualche linguaggio di programmazione, alle regole di produzione di una grammatica. Quando viene scelta una regola durante l'analisi della frase, viene svolta l'azione ad essa associata. Il risultato della generazione è, quindi, il risultato dell'esecuzione di tutte le azioni combinate fra di loro nell'ordine indotto dai costrutti della grammatica.

2.6 Induzione di grammatiche

L'**induzione di grammatiche** è il processo di apprendimento di una grammatica partendo dai dati. E' un'attività che viene spontaneo tentare, visto che costruire grammatiche a mano è molto faticoso mentre sulla rete sono disponibili miliardi di enunciati esemplificativi gratuiti. Il compito è difficile poiché lo spazio delle possibili grammatiche è infinito e verificare che una data grammatica generi un particolare insieme di frasi è computazionalmente molto costoso.

Un modello interessante è quello del sistema **SEQUITUR** (Nevill-Manning e Witten, 1997). Non è richiesto alcun input tranne un singolo testo (che non dev'essere precedentemente suddiviso in frasi). Il sistema produce una grammatica in forma molto specializzata, e precisamente quella che genera una singola stringa: il testo originale. Si può dire che SEQUITUR apprende tanta grammatica quanta è strettamente necessaria ad eseguire l'analisi sintattica del testo di input.

SEQUITUR è basato sull'idea che una grammatica è buona quando è compatta. In particolare sono sempre rispettati i seguenti due vincoli:

1. Nessuna coppia di simboli adiacenti deve comparire più d' una volta nella grammatica. Se la coppia A B compare nella parte destra di più regole, la si deve sostituire con un nuovo simbolo non terminale C e aggiungere la regola C AB.

2. Ogni regola dev'essere usata almeno due volte. Se un simbolo non terminale C compare una sola volta in tutta la grammatica, si deve eliminare la regola in questione e sostituire il suo singolo uso con la parte destra.

Questi due vincoli sono applicati all'interno di una ricerca golosa che esamina l'input da sinistra a destra, costruendo incrementalmente la grammatica e imponendo il rispetto dei vincoli il più presto possibile.

Capitolo 3

Rappresentazione della Conoscenza

3.1 Ingegneria della conoscenza

Il processo generale di costruzione di una base di conoscenza prende il nome di **ingegneria della conoscenza** (knowledge engineering). Un ingegnere della conoscenza investiga un particolare dominio, impara quali concetti sono importanti e scrive una rappresentazione formale degli oggetti al suo interno e delle relazioni tra essi. In altre parole è la disciplina che si occupa di come rendere le conoscenze disponibili ai sistemi informatici.

3.1.1 Conoscenza e uso della Conoscenza

I termini *conoscenza*, *informazione* e *dato* hanno molteplici significati e non è facile tracciare un confine largamente accettato. cerchiamo di dare un significato consono a ciò che si intende realizzare. Un **dato** è un elemento atomico ricavato dall'osservazione di un certo fenomeno, mentre un informazione è il risultato di un elaborazione di più dati. Il problema ora è come identificare la conoscenza. In letteratura si distinguono due tipi di conoscenza:

- **conoscenza procedurale**: saper svolgere un operazione (parlare, scrivere, programmare)
- **conoscenza dichiarativa**: avere informazioni su eventi, relazioni, aspetti dell'ambiente esterno

La nostra trattazione rifletterà prevalentemente le conoscenza dichiarative che ci consentono di rappresentare le informazioni da poter analizzare ed elaborare riguardanti l'ambiente esterno. Queste ultime possono essere classificate in due categorie principali:

- **conoscenza terminologiche**: conoscenza del lessico di una lingua, sapere a che cosa corrispondono i termini
- **conoscenza nomologiche**: conoscenza di regolarità, letti e relazioni che governano l'ambiente

Le conoscenze provengono da diversi fonti. La principale deriva dall'interazione diretta tra agente ed ambiente stesso. Tuttavia la maggior parte della conoscenza umana deriva dal **ragionamento**, che può essere deduttivo (dalle premesse alle conclusioni) o induttivo (dai fatti alle regole generali). Qualunque sia la fonte è essenziale la funzione della memoria, intesa come la capacità di conservare elementi di conoscenza nel tempo e reperirli quando servono.

Il problema principale è capire come poter rappresentare la conoscenza attraverso un linguaggio che può essere comprensibile ed elaborabile da un agente software. Quest'ultimo dovrà essere in grado di gestire le informazioni e utilizzarle tramite, procedure di ragionamento automatico, per interpretare la realtà, prevederne l'evoluzione, interprendere azioni per modificarla ed interagire con altre entità.

3.1.2 Realtà e sua Rappresentazione

Il primo passo per capire come la conoscenza possa essere resa comprensibile alle macchine è domandarsi che rapporto c'è tra realtà e sua rappresentazione. Consideriamo un qualsiasi enunciato, ad esempio:

c'è un asino che vola

Tale frase è sintatticamente corretta ma non ha alcun significato inquanto si sa perfettamente che un asino non può volare. Ciò viene dedotto dalle caratteristiche specifiche dell'animale. La frase:

c'è un asino in cortile

potrebbe essere corretta, ma il suo significato dipende dal contesto in cui ci troviamo. Appare, quindi, evidente che un enuncuato rappresenta o descrivo uno *stato di cose* nel mondo del discorso. E' *vero* se ciò che descrive sussiste (l'asino è effettivamente nel cortile), è *false* se non sussiste (l'asino è da un altra parte o non esiste). Tale definizione deriva da Aristotele ed è nota come **definizione corrispondentista della verità**.

Tale definizione assume che esistano solamente due classi di verità: vero e non vero. Purtroppo la realtà non può essere rappresentata con un assunzione binaria perché molte verità che affermiamo dipendono da punti di vista o non sono completamente vere o false. La si può approssimare se la si descrive molto dettagliatamente, cercando di identificare le eccezione in modo da gestirle in maniera specifica. Altrimenti è necessario ricorrere a tecniche che si basano sulla *teoria della probabilità* che consentono di esprimere diversi gradi di credenze.

Entrando più nel dettaglio è necessario definire formalmente il concetto di **enunciato** . Esso è costituito da una frase in una lingua, proferita in uno specifico contesto, contesto di enunciazione. Quest'ultimo determina il *mittente* e il *destinatario* dell'enunciato, il *luogo* e il *tempo* dell'enunciazione e altre informazioni. Una **frase**, invece, è una sequenza di parole che rispetta la grammatica di una lingua. La differenza risiede nel contesto stesso. Un enunciato è una frase contestualizzata.

Il **mondo del discorso** è la porzione di realtà di cui si parla negli enunciati. Può essere limitata a una stanza o essere estesa a tutto l'universo, può ricoprire un periodo temporale definito o essere spalmato su un intera era o più. Il mondo del discorso non va confuso con il contesto di enunciazione. Quest'ultimo è la situazione in cui si parla, mentre il mondo del discorso è la situazione di cui si parla.

Un mondo del discorso è costituito da **stati di cose**. In prima approssimazione, quest'ultimo, è composto da determinati individui, dalle loro proprietà e dalle relazioni che sussistono tra di essi. Per **individuo**, che in latino vuol dire indivisibile, si intende qualunque entità, concreta o astratta, vivente o non vivente, animata o inanimata, che possa essere trattata come un unico elemento a cui si possa fare riferimento. In generale agli individui si fa riferimento con *sintagmi nominali*. In genere le *proprietà* di individui sono espresse da preicati con un argomento, le *relazioni* da predicati a due o più argomenti.

Infine occorre definire con precisione la **valutazione di un enunciato**. Esso è un procedimento complesso che richiede di gettare un ponte fra il piano del linguaggio e quello della realtà. Consideriamo l'enunciato:

A è un animale

è vero nel mondo del discorso, secondo la definizione corrispondentista della verità, quando siamo in grado di *identificare* A e quando *comprendiamo* il significato della proprietà essere un animale. Ovvero quando siamo in grado di gestire il problema del **symbol grounding**.

Un modo intuitivo di capire il problema è considerare il problema proposto da Van Harnad nel 1990, ovvero domandarsi se sia possibile imparare il cinese stando chiusi in una stanza e avendo a disposizione solo un dizionario cinese-cinese. Il tentativo di capire un simbolo porta solo a passare in rassegna altri simboli senza dare luce al significato del simbolo. Il dizionario lega ogni simbolo ad altri in un sistema completamente autoreferenziale e non è possibile andare al di la dei simboli, perché non ci sono legami con la realtà. Non basta quindi possedere un dizionario, occorre anche mettere in giusta relazione i termini con la realtà e con il loro significato.

Se l'enunciato diventa più complesso, vale il **principio di composizione**: il valore di verità di un enunciato complesso è funzione dei valori di verità delle sue parti. Ad esempio, consideriamo la frase:

Il panettone è grande, con le mandorle ed è sul tavolo

La frase è versa se lo sono le sue componenti:

```
X è un panettone
Y è un tavolo
X è grande
X ha le mandorle
X è su Y
```

Le prime formule derivano dalla capacità di identificare i due oggetti, panettone, tavolo, grande, mandorle. L'ultima deriva da una relazione tra i due oggetti. La validità dell'enunciato globale dipende quindi da:

- capacità di identificare gli oggetti
- capacità di identificare la relazione tra loro

E' sempre necessario tenere presente che un enunciato è vero o falso relativamente al mondo del discorso. E può essere contemporaneamente vero in un contesto e falso in un altro.

3.1.3 Fasi di Ingegnerizzazione della Conoscenza

Tra i diversi progetti di ingegneria della conoscenza ci sono grandi differenze di contenuto, dimensione e difficoltà, ma tutti includono i seguenti passi.

1. **Identificare il compito della base di conoscenza**. L'ingegnere della conoscenza deve delineare la gamma di domande a cui la KB dovrà rispondere e le categorie di fatti disponibili per ogni specifica istanza di problema.

2. **Raccogliere la conoscenza rilevante**. L'ingegnere della conoscenza potrebbe già essere un esperto del dominio, o potrebbe lavorare insieme ad altri esperti per estrarre quello che sanno: quest'ultimo processo prende il nome di *acquisizione della conoscenza*. In questa fase non viene utilizzata una rappresentazione formale; lo scopo è comprendere l'entità della base di conoscenza, determinata dal suo compito, e capire come funziona effettivamente il dominio.

3. **Definire un vocabolario di predicati, funzioni e costanti**. A questo punto occorre tradurre i concetti importanti del dominio in nomi di simboli logici. Nel far questo sorgono molti problemi di stile: come lo stile di programmazione, anche quello di ingegneria della conoscenza può avere un impatto significativo sul successo di un progetto. Una volta fatte queste scelte, il risultato sarà un vocabolario che prende il nome di ontologia del dominio. Con la parola **ontologia** si intende una particolare teoria riguardante la natura dell'esistenza. Un'ontologia definisce le categorie di oggetti esistenti, ma non le loro specifiche caratteristiche e le relazioni tra esse.

4. **Codificare la conoscenza generale riguardante il dominio**. L'ingegnere della conoscenza scrive gli assiomi per tutti gli elementi del vocabolario. Questo definisce precisamente (nei limiti del possibile) il significato di ogni termine permettendo all'esperto del dominio di verificare il contenuto. Spesso questo passo rivela malintesi o lacune nel vocabolario, che dovranno essere colmate tornando al passo precedente e ripetendo il processo iterativamente.

5. **Codificare una descrizione della specifica istanza del problema**. Se l'ontologia è ben definita, questo passo dovrebbe risultare semplice: si tratta di scrivere semplici formule atomiche che riguardano istanze di concetti che fanno già parte dell'ontologia. Per un agente logico le istanze dei problemi sono fornite dai sensori, e la base di conoscenza iniziale è riempita di formule aggiuntive allo stesso modo con cui i programmi tradizionali ricevono dati di input.

6. **Interrogare la procedura di inferenza e ottenere da essa risposte**. A questo punto si possono raccogliere i frutti del proprio lavoro: la procedura di inferenza, partendo dagli assiomi e dai frutti specifici del problema, sarà in grado di derivare i fatti che ci interessa conoscere.

7. **Fare il debugging della base di conoscenza**. Purtroppo le risposte saranno raramente corrette al primo tentativo. Per essere più precisi le risposte saranno *quelle giuste per la base di conoscenza così com'è scritta*, presumendo che la procedura di inferenza sia corretta, ma potranno essere diverse da quelle che ci saremmo aspettati. Se mancasse un assioma (traducibile in assenza di un informazione), alcune interrogazioni non potranno avere risposta. L'assenza di assiomi può essere rilevata facilmente cercando interruzioni della catena di ragionamento.

3.2 Caratteristiche della Conoscenza

Abbiamo visto che un ontologia organizza tutti gli elementi del mondo in una gerarchia di categorie. Per realizzare un ongologia è necessario capire alcune caratteristiche della conoscenza che ci aiuteranno nell'identificare una serie di elementi utili al fine di ottenere una sua rappresentazione.

3.2.1 Classi

Una **classe** è l'insieme di tutte le entità che hanno caratteristiche comuni. Tale definizione pone l'accento sul fatto che si tratta di un *insieme*, non di una singola entità e, come tale, viene utilizzato per discriminare gli elementi ad essa appartenenti rispetto a tutti quelli che possono essere presi in considerazione.

Per poter effettuare questa discriminazione è essenziale che allinsieme sia associata una modalità che permetta di valutare lappartenenza di un elemento generico allinsieme stesso. Questa modalità è generalmente chiamata **funzione dappartenenza**, in quanto è pensata come una funzione che, applicata ad un elemento generico deluniverso del discorso (nel nostro caso unentità), fornisce un grado di appartenenza di tale elemento allinsieme (nel nostro caso una classe). Quando si parla di classi, la funzione dappartenenza è funzione delle proprietà della classe. In altri termini: tanto più unentità ha proprietà coerenti con quelle che descrivono la classe, quanto più tale entità appartiene alla classe stessa.

Il fatto che una classe permetta di definire delle proprietà comuni ad un insieme di elementi è di importanza fondamentare per unefficace gestione dela conoscenza. In questo modo si riesce a raggruppare diverse entità in una unica (la classe appunto) e a ragionare in modo efficiente ed in termini generali su di essa.

Tramite questo concetto è anche possibile effettuare un particolare processo detto **classificazione**, cioè di ricondurre unentità ad un certo insieme. Questo permette di aumentare la conoscenza a disposizione, in quanto, nel momento in cui si sa che unentità appartiene ad una certa classe, si può assumere, finché non viene provato il contrario, che quellen-

tità possegga tutte le proprietà che descrivono la classe, anche se, in realtà, si conoscono direttamente solo alcune di esse.

Un concetto legato alle classi è quello dell'**ereditarietà**: quando si cerca di rappresentare la conoscenza, solitamente si iniziano a classificare le entità in relazione a caratteristiche comuni. Man mano che il livello di dettaglio aumenta si creano dei *sottoinsiemi* sempre più piccoli di entità. Ciascun elemento di tale sottoinsieme ha proprietà comuni con elementi di altri sottoinsiemi ma altre differenti. Si può dire, quindi, che un elemento appartenente ad un sottoinsieme, che prende il nome di classe derivata, possieda tutte le proprietà che caratterizzano l'insieme principale, con l'aggiunta di altre peculiari a tale raggruppamento.

Consideriamo, per esempio, un cane: sappiamo con certezza che questo è un mammifero. Di conseguenza possiamo dire che possiede tutte le caratteristiche peculiari di tale insieme di esseri viventi. Un cane, però non è una scimmia. Ciò porta ad ottenere differenti classi di animali che fanno tutti parte della classe Mammifero ma differiscono tra loro per caratteristiche comuni. Il cane, ad esempio, ha una forma e un verso completamente differenti rispetto alle medesime caratteristiche della scimmia.

L'uso delle classi è stato integrato completamente nei linguaggi di programmazione per la semplicità di utilizzo e per le caratteristiche già elencate in precedenza.

3.2.2 Esemplari

Un **esemplare** è unentità singola, descritta da sue proprietà caratteristiche. Di solito gli esemplari sono messi in relazione con una o più classi. Il fatto che un esemplare possa appartenere a più classi è possibile solo quando le classificazioni sono fatte secondo criteri diversi. Ad esempio, uno struzzo può essere classificato come uccello, ma anche come corridore, o come animale del deserto. Le due classi nascono da criteri di classificazione diversi: la prima nasce da una classificazione zoologica, mentre le altre seconda nascono da una classificazione basata sulle caratteristiche degli esemplari (nel nostro caso caratteristiche di locomozione e di luogo di presenza).

Il concetto di esemplare serve per poter descrivere singole entità con le loro caratteristiche specifiche. Le proprietà che descrivono un esemplare possono essere le stesse che si trovano nella sua classe od in un prototipo, oppure possono essre ulteriori proprietà caratteristiche dellesemplare specifico.

Consideriamo, ad esempio, Pippo: possiamo dire che è un cane (e quindi, è un mammifero, si nutre, respira, abbaia ecc), abita in una cuccia, ed è un amico di Topolino.

Da quanto visto fin ora emergono due problematiche riguardanti gli esemplari:

- identificazione
- numero di esemplari

Il primo problema è relatvo alla presenza di più esemplari distinti dal solo nome, ma descritti tutti nello stesso modo. Questa situazione è caratteristica in carenza di informazione,

quando cioè la conoscenza che permetterebbe di distinguere diversi esemplari non è disponibile. Quando si verifica una situazione di questo genere, il progettista dovrebbe chiedersi se effettivamente ha senso avere diversi semplari, o, dualmente, perchè ritiene che debbano esserci diversi esemplari distinti dal solo nome. Così operando si riesce solitamente a far emergere caratteristiche che permettano di rappresentare conoscenza che a sua volta permetta di distinguere i due esemplari.

Il secondo problema citato si riferisce al numero di esemplari. Nelle applicazioni, avere un alto numero di esemplari per una certa classe può essere indice di carenza di conoscenza che permetta di specificare meglio la classe. In questo caso occorre chiedersi se non sia possibile o conveniente identificare ulteriori elementi comuni ad un sottoinsieme degli elementi appartenenti alla classe, in modo da partizionare ulteriormente la classe in sottoclassi.

Nella programmazione orientata agli oggetti si identifica questo concetto con **istanza di classe**. Solitamente un istanza appartiene ad una sola classe. Questa però può essere ereditata da più classi fornendo alle sue istanze tutte le caratteristiche peculiari di ciascuna di esse. Su queste tipologie di ereditarietà possono nascere problemi tali per cui si preferisce evitare ereditarietà multiple.

3.2.3 Proprietà

Le **proprietà** permettono di descrivere le entità e differenziarle tra loro. Per quanto detto finora è evidente che il concetto di proprietà è fondamentale per la rappresentazione della conoscenza: permette non solo di descrivere entità, ma anche di creare relazioni fra di loro.

Occorre notare che, trattanandosi di un concetto di cui si può parlare, la proprietà è a sua volta unentità, descrivibile da proprietà per essa caratteristiche. Ad esempio il range di valori tipico per una proprietà, il livello di affidabilità che abbiamo circa il fatto che un esemplare abbia davvero una proprietà con un certo valore, ecc.

Tramite il concetto di proprietà, quindi, andiamo a realizzare una classe: tutte le entità che posseggono tali caratteristiche sono entità di una specifica classe. Un cane, ad esempio, ha, come un gatto e un topo, una proprietà che è l'*età*. Tale proprietà può assumere un range di valori tra 0 e 30 (valore indicato per solo scopo esemplificativo non ottenuto da alcun tipo di analisi).

Non sempre è facile identificare una classe e le sue proprietà. Lo stesso concetto può essere espresso in svariati modi utilizzando una sola classe con tantissime proprietà, oppure molte classi con poche proprietà, oppure classi che usano istanze di altre classi come proprietà. Non esiste un unica rappresentazione assoluta e universale. Tutto dipende dall'obiettivo che ci si pone e, una volta definito in maniera formale ed esplicita, dalla parte di realtà che si vuole utilizzare per tale fine.

3.2.4 Relazioni e gerarchie

Una **relazione** è un legame che coinvolge due o più entità ed è caratterizzata da una proprietà che ne identifica il rapporto. Un padre è legato a un figlio da un rapporto di parentela

Una **gerarchia** è una generica relazione di ordinamento in relazione ad una certa proprietà. Le gerarchie a cui siamo abituati a pensare (si pensi alle gerarchie di classificazione tradizionali come quelle delle Scienze Naturali), sono gerarchie sviluppate secondo le proprietà di specializzazione e generalizzazione.

Altre gerarchie possono essere definite, ad esempio, secondo la proprietà *PartOf*, che crea una relazione tra le parti di unentità e lentità stessa. Altra gerarchia comunemente usata è quella definita attraverso la proprietà *instanceOf*, tra esemplari e classi.

Il concetto di relazione gerarchica serve per realizzare il meccanismo delleredità già visto in precedenza.

In generale, unentità può essere in relazione gerarchica con diverse altre entità. Di solito, nel meccanismo di eredità, è adottata una politica in cui le proprietà definite ai livelli inferiori della gerarchia hanno la meglio in caso di conflitto, rispetto alle analoghe proprietà definite ai livelli superiori. Questo meccanismo si chiama sovrascrittura (**overriding**) dei valori e viene attuato quando le proprietà considerate non sono quelle caratteristiche, ma quelle tipiche delle classi.

3.3 Sistemi di ragionamento per categorie

Le categorie sono gli elementi principali per la costruzione di schemi di rappresentazione su grande scala. Questo paragrafo descrive i sistemi esplicitamente progettati per organizzare e ragionare sulle categorie. Ci sono due categorie di famiglie di sistemi, strettamente imparentate:

- **reti semantiche**: permettono di visualizzare graficamente una base di conoscenza e forniscono algoritmi efficienti per inferire le proprietà di un oggetto sulla base della sua appartenenza a una categoria
- **logiche descrittive**: rappresentano un linguaggio formale per costruire e combinare definizioni di categorie e forniscono algoritmi efficienti per determinare le relazioni di sottoinsieme e superinsieme tra categorie.

In questo testo ci soffermeremo sulla trattazione delle prime.

3.3.1 Reti semantiche

Nel 1909 Charles Peirce propose una notazione grafica basata su nodi e archi che chiamò **grafi esistenziali** e definì la logica del futuro. Cominciò cosi un lungo dibattito tra i paladini della logica e quelli delle reti semantiche. Sfortunatamente, le diatribe nascosero il fatto che le reti, almeno quelle la cui semantica è ben definita, sono una formula di logica. Per certi tipi di formule la notazione offerta dalle reti semantiche è spesso più comoda, ma se non consideriamo le questioni di interfacciamento con gli esseri umani, i concetti sottostanti (oggetti, relazioni, quantificazioni ecc.) sono identici.

Ad una prima approssimazione, le reti semantiche sono dei formalismi per rappresentare la conoscenza aventi una struttura a grafo. Solitamente i nodi rappresentano oggetti,

concetti, stati, mentre gli archi, eventualmente etichettati, rappresentano le relazioni tra i nodi. Esse condividono lidea di base che strutture complesse possano essere descritte come una collezione di attributi, e valori associati, e che il ragionamento è, nella sua forma più astratta, una realizzazione di collegamenti, la cui giustificazione è riconducibile al buon senso: esperienza, somiglianza, dipendenza, tipicità.

Sono state spesso proposte come modello della memoria umana, largamente utilizzate per la comprensione del linguaggio naturale e sono attualmente considerate un tipo comune di dizionario leggibile da una macchina.

Ci sono molte varianti di reti semantiche, ma tutte sono capaci di rappresentare oggetti singoli, categorie di oggetti e relazioni. Una notazione grafica tipica rappresenta i nomi degli oggetti e delle categorie racchiusi in ovali o rettangoli, collegati con archi etichettati.

INSERIRE DISEGNO

La notazione delle reti semantiche rende molto semplice ragionare sull'**ereditarietà delle proprietà**. Consideriamo, per esempio, un *Cavallo*: questo è un mammifero e, come tale, condivide alcune caratteristiche con una scimmia e un delfino, ma ne ha alcune di proprie che ne identificano univocamente l'appartenenza a questa categoria. L'eredità diventa complicata quando un oggetto può appartenere a più di una categoria, o quando un a categoria può essere un sottoinsieme di più di un'altra categoria; in questo caso si parla di ereditarietà multipla. Ora un algoritmo di ricerca potrebbe trovare , in risposta a una interrogazione, due o più valori in conflitto. Per questa ragione alcuni linguaggi di programmazione orientati agli oggetti (OOP), come Java, proibiscono l'uso dell'eredità multipla nella gerarchia delle classi. Nelle reti semantiche, comunque, essa è generalmente consentita. Consideriamo, ad esempio, un cavallo alato. E' chiaro che dovrà avere i caratteri peculiari degli uccelli ma anche dei cavalli. Ma tutte e due le categorie derivano direttamente da *Mammifero*. Ciò porta a una rappresentazione simile alla seguente:

INSERIRE DISEGNO

Una limitazione delle reti semantiche è data dal fatto che i collegamenti tra le bolle possono rappresentare solo relazioni binarie. Tale restrizione ha come conseguenza la creazione di una ricca ontologia di concetti reificati.

Uno degli aspetti più importanti è la capacità di rappresentare **valori di default** per le categorie.

Parte II

Implementazione del Sistema

Capitolo 4

Parser Testuale

4.1 Linea guida

Il parser testuale è l'oggetto che si occupa della prima fase dell'analisi, ovvero l'**analisi sintattica**. Tale operazione è suddivisa nei seguenti passi:

- identificazione delle singole stringhe
- identificazione del verbo
- identificazione soggetto dell'azione
- identificativo dell'oggetto dell'azione (non sempre presente)

Alla base del parser si è definita una specifica grammatica che si basa sul seguente modello:

```
< frase > = < SN > < SV > |
                    < SV > < SN > |
                    < S > <Congiunzione > < S >
< congiunzione > = e | oppure | ma ...
< SN > = < Pronome > |
                  < NomeProprio> |
                  < Sostantivo > |
                  < Articolo > |
                  < Cifra > |
                  < SN > < PP > |
                  < SN > < PropCond >
< SV > = < Verbo > |
                  < SV > < SN > |
                  < SV > < Aggettivo > |
                  < SV > < PP > |
                  < SV > < Avverbio >
< PP > = < Preposizione > < SN >
```

```
< Preposizione > = di | a | da | in | con | su | per | tra | fra
< PropCond > = che < SV >
< Pronome > = io | me | mi | tu | te | ti | lui | gli | le | lei | egli |
                noi | ci | voi | vi | essi | loro
< Articolo > = il | lo | la | i | gli | le
< Cifra > = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
< NomeProprio > = ....
< Aggettivo > = ...
```

Come si può notare, tale grammatica dovrebbe essere in grado di riconoscere frasi del linguaggio italiano. Il problema di applicarla è legato, però, alla sua natura ambigua e a problemi di efficienza nell'analisi stessa. Un linguaggio parlato, infine, non è un linguaggio che segue una sintassi rigorosa e formale: spesso vengono espresse frasi in cui non compare un verbo o non sia indicato neppure un soggetto ma utili al fine di identificare una necessità dell'utente.

Frasi del tipo *che fame* non rispecchiano alcuna costruzione derivabile dalla grammatica mostrata in precedenza ma è utilissima al fine di identificare un bisogno specifico dell'utente.

Il parser si deve, quindi, occupare di identificare alcuni elementi fondamentali dai quali si potrebbe dedurre necessità o bisogni. Per fare questo si è scelto un'implementazione alternativa che non segue i modelli visti in precedenza nei paragrafi 2.4.1 e 2.4.2. Alla sua base vi è, comunque, la grammatica elencata in precedenza, ma si cerca di non vincolare l'utente a generare frasi ben formate.

Si può, quindi, affermare che la grammatica non è *la* regola di riconoscimento ma *una* linea guida. Le frasi vengono riconosciute anche se non sono propriamente conformi alla sintassi ad essa associata.
Il risultato che si ottiene è rappresentato in figura 4.1:

Figura 4.1: Schema del parser

Come si può notare il parser è un oggetto che prende frasi in ingresso, le processa e da esse estrae un oggetto che contiene una lista di oggetti corrispondenti alle stringhe generate in ordine di lettura, il soggetto, il verbo ed, eventualmente, l'oggetto dell'azione. Tale

rappresentazione è in contrapposizione all'albero sintattico generato da parser top-down e botton-up.
La figura 4.2 ne mostra un esempio.

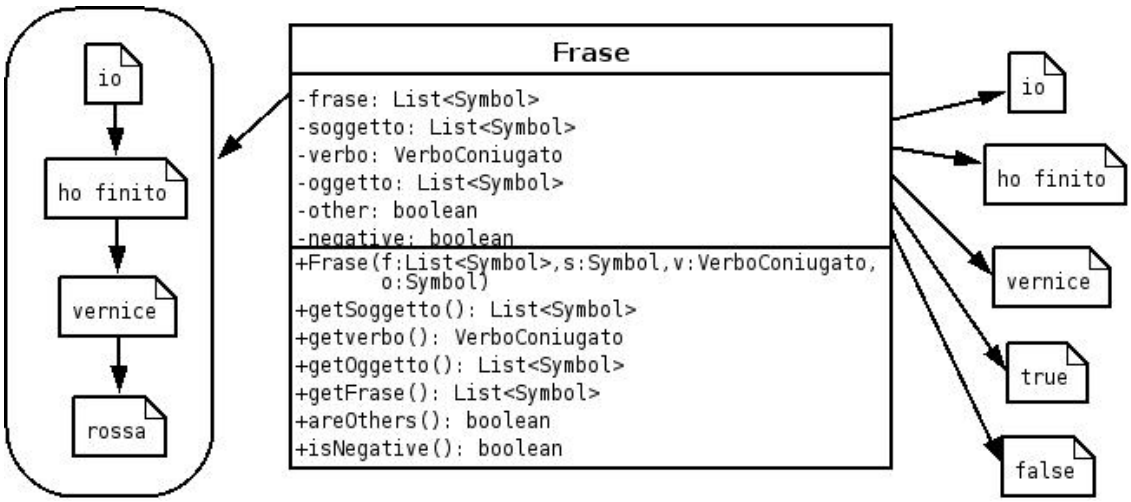

Figura 4.2: Schema UML della classe Frase per la frase *io ho finito la vernice rossa*

Tale oggetto è l'output fornito dal parser sintattico in corrispondenza della frase:

Io ho finito la vernice rossa

Come si può osservare, la frase ha un soggetto esplicito (*Io*), un verbo (*ho finito*) e un oggetto (*vernice*), quest'ultimo arricchito da un aggettivo (*rossa*). Al fine di una corretta analisi semantica sono necessari prevalentemente i primi tre elementi, identificati e isolati dal resto della frase. Il fatto che l'attributo soggetto e oggetto siano liste è dovuta al fatto che una frase può avere più soggetti e più oggetti come ad esempio:

Immanuel e Checilia sono colleghi
mi serve il pane e la carne

In entrambe le frasi abbiamo un soggetto multiplo (*Immanuel e Checilia* nella prima e *pane e carne* nella seconda) legati dalla congiunzione *e*.

Tuttavia, viene raccolta in una lista l'intera sequenza di stringhe in ordine di produzione. Ciò viene fatto per poter analizzare eventuali elementi aggiuntivi, nell'esempio riportato l'aggettivo *rossa*, in grado di aumentare il contenuto informativo dell'azione stessa. Se non venissero considerati, il sistema non sarebbe in grado di essere specifico. Ciò porta a un maggiore grado di dettaglio e di precisione riguardo al supporto fornito all'utente. Se non si identificasse, nell'esempio, che è finita la vernice rossa, l'utente non saprebbe che colore acquistare una volta giunto al negozio più vicino.

Grazie a questo oggetto è possibile semplificare il riconoscimento di necessità o bisogni dell'utente tramite l'analizzatore semantico che verrà analizzato nel capitolo 5.

4.2 Pacchetti e Classi Java

Per capire il funzionamento del parser è necessario, innanzitutto, mostrare la struttura delle classi, e quindi le strutture dati, utilizzate. Per semplicità si è scelto di realizzare tre diversi pacchetti nei quali sono raccolte le varie classi sulla base della loro utilità.
I pacchetti sono:

- **types**
- **databases**
- **parser**

4.2.1 Pacchetto types

All'interno di questo pacchetto sono definite le classi che corrispondono ai tipi di simboli che compaiono nel linguaggio, in questo caso adattati alla lingua italiana. Ciascuna classe rappresenta un *simbolo non terminale* della grammatica vista in precedenza. Si noti il forte legame tra grammatica e classi: ogni volta che viene analizzato un simbolo in ingresso, rappresentato da un oggetto della classe String (fornita dal linguaggio Java) il parser cerca di identificare l'oggetto ad esso associato. Vale a dire cerca di capire se l'oggetto appena letto è una parola, un verbo, un aggettivo ecc. Sulla base di tale identificazione il parser può scegliere la regola di produzione opportuna da applicare e, di conseguenza, ciò influenza il riconoscimento della frase.

Le classi contenute in questo pacchetto non sono altro che una forma di rappresentazione dei *simboli non terminali* della grammatica, mentre le funzioni fornite dalla classe Parser si occuperanno di applicare le opportune regole grammaticali per l'identificazione di soggetto, verbo e oggetto. Gli oggetti di classe String letti, sono i *simboli terminali*.

Il fatto di utilizzare oggetti specifici per ogni tipologia di simbolo consente di evidenziare, tramite valori a dati membro della classe, caratteristiche e proprietà peculiari che aiutano il parser nel riconoscimento e ricostruzione della sintassi della frase. Un verbo, ad esempio, possiede la caratteristica di essere coniugato in un certo tempo e una certa persona. Sulla base di questi dati si è in grado di identificare, all'interno della frase chi o quali potrebbero essere i soggetti dell'azione da esso rappresentata.

La figura 4.3 mostra l'intero schema UML del pacchetto types, sottolineando l'albero di derivazione delle classi. Da questa rappresentazione è omessa la derivazione implicita dalla classe Object perché priva di contenuto informativo.

Classe Symbol

La classe principale è la classe **Symbol** che contiene una stringa corrispondente a ciò che viene letto. Tale classe disponde di un *costruttore*, una funzione *equals* che si occupa di verificare se due oggetti sono identici e una funzione **getType()** che ritorna il codice corrispondente alla tipologia di oggetto di cui si ha l'istanza.

Il costruttore accetta come argomenti:

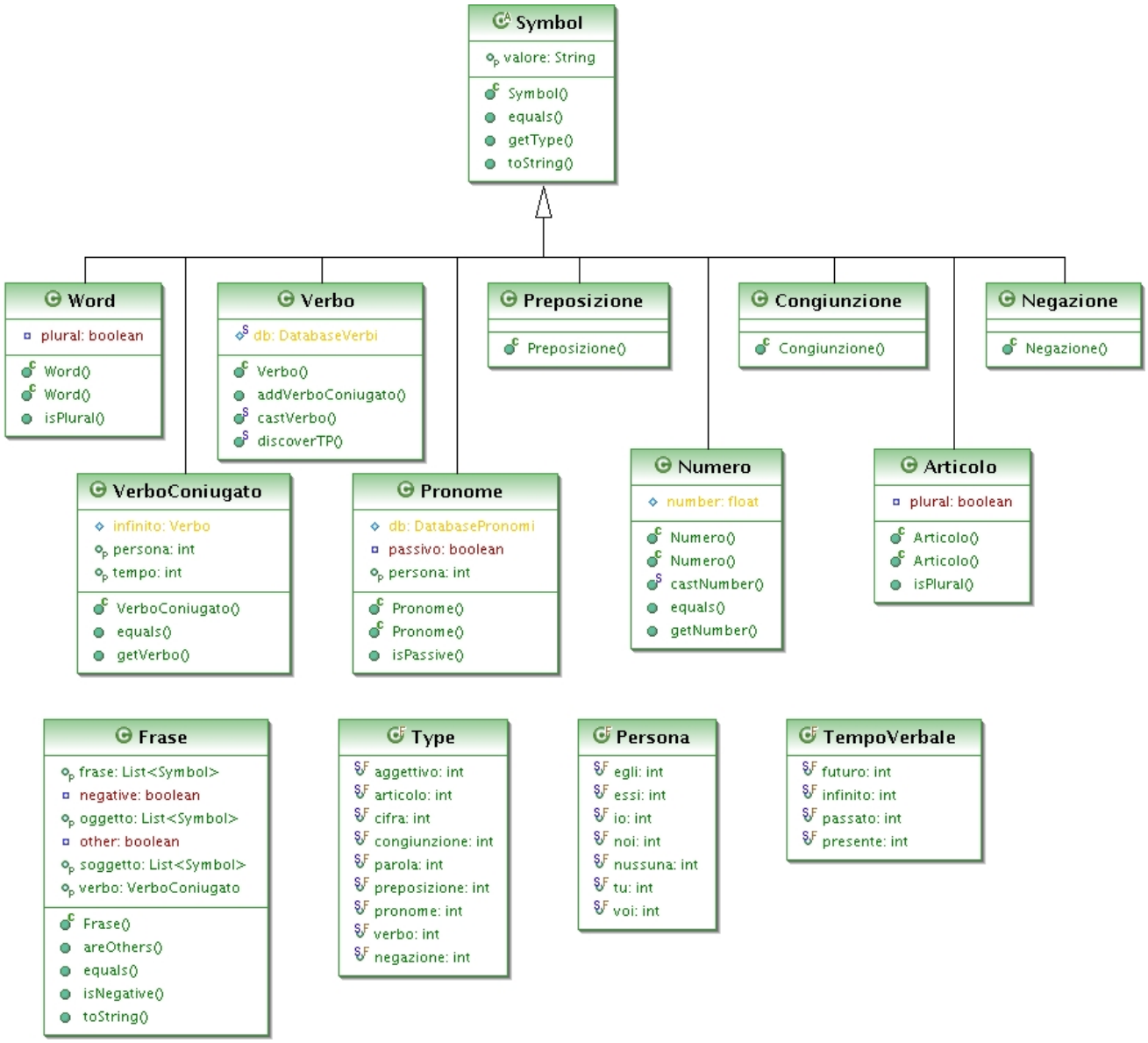

Figura 4.3: Schema UML del pacchetto types

- String s: stringa corrispondente alla parola letta
- int t: tipo di oggetto

Il tipo di oggetto è un intero che corrisponde a una costante fornita dalla classe *Type*.

Classe Type

Per rendere più parametrizzabile possibile il codice si è introdotta una classe, **Type**, che contiene delle costanti, una per ogni tipologia di oggetto che il parser riesce a riconoscere. questi sono:

- parola = 1;
- verbo = 2;
- pronome = 3;
- articolo = 4;
- cifra = 5;
- aggettivo = 6;
- preposizione = 7;
- congiunzione = 8;
- negazione = 9;

L'utilizzo di tali costanti aiuta, durante l'esecuzione del codice, a capire di quale simbolo si tratti evitando di avere problemi derivati all'ereditarietà. Maneggiando puntatori alla classe base, l'invocazione del metodo *instanceof* può generare ambiguità dovute al fatto che una parola (*Word*) è sempre un Symbol come lo è un aggettivo (*Aggettivo*). L'uso di getClass e il conseguente controllo su un oggetto di tipo Class eviterebbe simili problemi ma computazionalmente è molto oneroso.

Classe Persona

Come visto in precedenza per la calsse Type, per parametrizzare pronomi e dare una congruenza con i verbi, si è scelto di usare una classe, **Persona**, che si occupa di definire delle costanti da condividere in ogni parte del codice:

- nessuna = 0;
- io = 1;
- tu = 2;

- egli = 3;
- noi = 4;
- voi = 5;
- essi = 6;

Da notare la prima costante: *nessuna* che indica, appunto, nessuna persona. Il suo utilizzo è dedicato a casi in cui i le coniugazioni verbali non possiedano una persona. Si cosiderino, ad esempio, i tempi *infinito*, *participio* e *gerundio*.

Classe TempoVerbale

Anche per i tempi dei verbi è stata adottata una scelta di parametrizzazione. Questa volta il range non ricopre tutti i possibili tempi verbali ma esprime la seguente classificazione:

- infinito=0;
- presente = 1;
- passato = 2;
- futuro = 3;

Per scelta i verbi espressi hanno solo 3 tempi, infinito escluso. Tali tempi rappresentano le forme prevalenti di collocamento temporale di un azione. In italiano, come anche in inglese e in francese, i tempi sono molti di più e vengono usati per esprimere relazioni tra le varie frasi oppure per indicare azioni in corso ma si possono approssimare tutte a uno di questi tre tempi. La perdita di informazione non è rilevante al fine di identificare i bisogni dell'utente.

Un altro vantaggio riguarda il supporto multilingua. Non tutti i linguaggi esprimono i tempi verbali allo stesso modo ma tutti intendono riferire di fatti collocati al presente, al passato o al futuro. Ciò porterebbe a modificare solo alcuni elementi, specifici della classe Parser nell'individuazione del tempo e non implicherebbe sconvolgimenti di codice in altre parti.

Classe Word

La classe **Word** (parola) si occupa di rappresentare tutte quelle parole corrispondenti a oggetti e persone. Rispetto alla classe base Symbol prevede l'aggiunta di un valore booleano che indica se tale oggetto è singolare o plurale. Di default l'oggetto prevede il singolare. Il parser, come vedremo più avanti, decide che la stringa corrisponente alla parola sia plurale in relazione a tre condizioni:

- sia preceduta da un articolo plurale

- sia preceduta da un numero maggiore di 1
- sia esplicitamente plurale in relazione alla ricerca su un portale web

La funzione **isPlural()**, infine, si occupa di informare se si tratti di un istanza corrispondente a una parola plurale o singolare.

Classe Articolo

La classe **Articolo** si occupa di rappresentare gli articoli, distinguendoli tra singolari e plurali. Tale distinzione è necessaria per identificare le parole che questi precedono. Il funzionamento è analogo a quello della classe *Word*.

Classe Congiunzione

La classe **Congiunzione** si occupa di gestire parole tipo *e*, *oppure* che legano parole o frasi tra loro. E' usata nel Parser per identificare soggetti o oggetti multipli. Attualmente è supportata solo la congiunzione o disgiunzione di parole o aggettivi, non riesce a legare più frasi tra loro.

Classe Negazione

La classe **Negazione** si occupa di rappresentare tutti quei simboli che negano un azione e danno un accezione negativa alla frase. Se all'interno della frase compare una negazione viene impostato a true un apposito campo all'interno dell'oggetto Frase. In questo modo l'analizzatore semantico sarà in grado di identificare l'accezione dell'azione espressa dal verbo in essa contenuto.

Classe Preposizione

La classe **Preposizione** si occupa di gestire le preposizioni, in italiano i termini *di, a, da, in, con, su, per, tra, fra*. Tale preposizioni hanno il compito di legare gli elementi tra loro e aggiungere precisione nell'identificare oggetti, posizioni e azioni stesse. Vengono usate dall'analizzatore semantico al fine di poter effettuare ricerche più mirate.
Si consideri la frase:

```
Mi serve il vaso di ceramica
Ho voglia di gelato
```

La preposizione *di* in entrambi le frasi aggiunge dettaglio all'oggetto del desiderio/necessita'.

Classe Pronome

La classe **Pronome** si occupa di gestire i pronomi. Tale classe è necessaria per il riconoscimento dei soggetti/oggetti all'interno della frase a partire dal verbo. Ogni pronome è

associato a una Persona tramite un dato membro intero che richiama una costante definita dalla classe *Persona*.

Ogni oggetto della classe *Persona* ha un riferimento a un oggetto statico di tibo *DatabasePronomi*. Questo serve per raccogliere i pronomi in una mappa indicizzabile tramite la persona che essi rappresentano. E' usata per ricavare il soggetto corretto dal verbo in frasi in cui non è esplicitato.
Si consideri, ad esempio, la seguente frase:

ho finito il latte

In tale frase il soggetto non è presente ma lo si può dedurre dal verbo che, essendo coniugato alla prima persona singolare, si può dedurre essere *io*.

Ogni volta che un oggetto di tipo *Pronome* viene costruito, viene creata un entry opportuna nel database dei Pronomi che, essendo pensato come singleton, si istanzia non appena viene invocato per la prima volta il suo metodo *getInstance()*, altrimenti la funzione restituirebbe un riferimento all'oggetto già allocato.

Infine si ponga l'attenzione sul dato membro *passivo* di tipo boolean: tale attributo riflette la proprietà di alcune forme di pronomi di subire una specifica azione. Si pensi alla frase:

io sono un nerd
e
ci piace la musica di Immanuel Casto

La prima frase indica che *io* è un pronome soggetto che compie l'azione di *essere nerd*, nel secondo caso il soggetto è la *musica* che piace a chi produce la frase. In questo caso il pronome *ci* subisce l'azione. La frase la si può capire meglio se in forma esplicita:

La musica di Immanuel Casto piace a noi

Entrambi le frasi indicano lo stesso concetto. In questo caso, però, è evidente che il soggetto è la musica e non chi produce la frase.

Classe VerboConiugato

La classe **VerboConiugato** si occupa di rappresentare le coniugazioni dei verbi. Oggetti di tale istanza, derivando dalla classe Symbol possiedono un dato membro di classe String che identifica il termine (ad esempio *ho mangiato*) con l'aggiunta di due dati membro:

- int tempo
- int persona
- Verbo infinito

Il primo assume valori in relazione alle costanti definite dalla classe *TempoVerbale*, il secondo in relazione a quelli fornite dalla classe *Persona*. L'ultimo dato membro identifica l'appartenenza della forma coniugata a un ramo di coniugazioni a partire da una forma infinita. *Ho mangiato* appartiene al verbo *mangiare*.

Vengono forniti i metodi di get per ottenere tali valori: *getTempo()*, *getPersona()* e *getVerbo()*. Essi sono usati dal parser e dall'analizzatore semantico per capire e identificare il senso della frase e/o i possibili soggetti/oggetti dell'azione descritta dal verbo.

Il costruttore, unico, accetta come argomenti: la stringa corrispondente alla coniugazione, il tempo, la persona e un riferimento all'oggetto di tipo *Verbo* a cui tale coniugazione appartiene. Per capire come vengono ricavati questi dati si guardi la descrizione della classe *Verbo*.

Classe Verbo

La classe **Verbo** si occupa di rappresentare le forme infinite dei verbi. L'idea alla base è quella di avere un unico oggetto che descriva il verbo e un insieme di oggetti di tipo *VerboConiugato* che descrivano le coniugazioni nelle varie persone e tempi. In altre parole, il verbo *mangiare* possiede un istanza di classe *Verbo* corrispondente all'infinito e una per *io mangio*, *tu mangi* ecc. di tipo *VerboConiugato*.

Questa classe, derivando direttamente da Symbol, possiede una stringa che identifica la forma infinita del verbo. A ciò viene aggiunto un riferimento a un oggetto di tipo *DatabaseVerbi* che si occupa di raccogliere le varie forme coniugazioni dei verbi. Il parser è in grado di identificare tempo e persona dei verbi regolari grazie a un algoritmo che studia le terminazioni. Per i verbi irregolari, tale procedura è impossibile e diviene fondamentale possedere un database cin le coniugazioni.

Abbiamo visto prima, nel *VerboConiugato*, che, per istanziarne un oggetto, è necesessario sapere tempo e persona in cui la forma verbale è coniugata. La classe Verbo, tramite la funzione **discoverTP(String s)**, si occupa proprio di effettuare tale operazione. Il funzionamento è molto semplice: si verifica se la stringa passata per argomento termina con un postfisso appartenente a una particolare coniugazione. L'appartenenza identifica (quasi) univocamente tempo e persona della forma coniugata analizzata. Il valore di ritorno è un array contenente i seguenti valori:

1. prefisso: parte immutata del verbo comune a tutte le coniugazioni
2. tempo: codice corrispondente ai valori definiti dalla classe *TempoVerbale*
3. persona: codice corrispondente ai valori definiti dalla classe *Persona*

Si consideri, per esempio, la seguente stringa:

```
stringa: mangio

prefisso: mangi
coniugazione: are
```

```
tempo: presente
persona: prima singolare
```

Questa funzione è usata dalla funzione statica **castVerbo(String s)**. L'idea è quella di avere una funzione, indipendente dalle istanze, che verifichi se una stringa è una forma coniugata di un verbo. Il funzionamento è molto semplice e riassumibile nei seguenti passi:

1. verifica se la stringa è contenuta nel database dei verbi. In caso affermativo ritorna l'oggetto corrispondente.
2. verifica se la stringa termina in *are*, *ere*, *ire* (forme infinite). In caso affermativo viene creato un oggetto Verbo e viene ritornata la forma corrispondente di istanza VerboConiugato
3. verifica se la stringa può essere un verbo richiamando la funzione *discoverTP(String s)*. Nel caso in cui si individua persona e tempo viene creato un oggetto di tipo VerboConiugato corrispondente alla stringa rilevata e viene ritornata
4. se nessuna delle precedenti verifiche ha avuto successo, viene ritornato null. Ciò vuol dire che la stringa non è un verbo riconosciuto.

Classe Numero

La classe **Numero** si occupa di rappresentare i numeri, intesi in virgola mobile. Una loro rappresentazione in forma di stringa non garantisce una efficenza in fase di elaborazione (causa problemi di cast e rallentamenti). Alla classe *Symbol* viene aggiunto un dato membro di tipo float che contenga il valore corrispondente.

Anche in questo caso, come nella classe *Verbo* esiste una funzione di cast opportuna: **castNumber(String s)**. Questo metodo prende la stringa e, tramite la funzione **Float.parseFloat(String s)** si prova a convertirla in un numero float. In caso di fallimento (vale a dire che si tratti di una vera e propria stringa alfanumerica) viene raccolta l'eccezzione e ritornato null.

Classe Frase

Come già visto in precedenza, la classe **Frase** raccoglie il risultato dell'analisi sintattica, evidenziando il soggetto, l'oggetto e il verbo della frase analizzata. Tale classe non deriva da Symbol.
Un qualsiasi oggetto della classe *Frase*, possiede i seguenti dati membro:

- List<Symbol> frase: contiene l'intera lista di simboli che costituiscono la frase, priva di articoli.
- List<Symbol> soggetto: contiene la lista di soggetti che compiono l'azione espressa dalla frase.

- VerboConiugato verbo: contiene il verbo coniugato che esprime l'azione che si vuole rappresentare con la frase.
- List<Symbol> oggetto: contiene la lista di elementi che subiscono l'azione espressa.
- boolean other: indica se ci sono altri elementi nella frase che non sono stati identificati come soggetti od oggetti.
- boolean negative: indica se la frase è espressa con accezione negativa (mancanza).

Grazie all'utilizzo della variabile booleana other e della possibilità di avere la frase originaria completa, l'analizzatore semantico è in grado di identificare elementi aggiuntivi che caratterizzerebbero meglio e aggiungerebbero più dettaglio alla frase stessa. Ciò porta a identificare con maggiore precisione eventuali bisogni o mancanze dell'utente, supportandolo con maggiore precisione nelle fasi decisionali.

4.2.2 Pacchetto databases

Il pacchetto **databases** fornisce le classi che si occupano di gestire i database e, più in particolare, quello relativo ai *Verbi* e ai *Pronomi*. Il terzo database, *DatabaseRelazioni*, verrà usato per l'analisi semantica e sarà analizzato nel capitolo 5. Grazie a tali database è possibile identificare le stringhe lette e semplificare il riconoscimento testuale. In figura 4.4 è possibile vedere lo schema UML delle classi appartenenti al pacchetto. Tutti gli oggetti

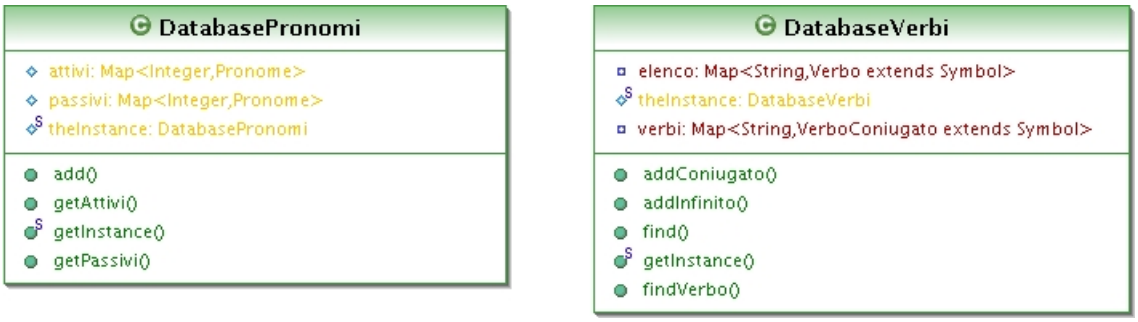

Figura 4.4: Schema UML del pacchetto databases

istanziati sulla base delle classi contenute in tali pacchetti sono dei **singleton**, ovvero un oggetto che viene istanziato la prima volta che viene richiesto e da quel momento resta disponibile per chi ne fa richiesta.

La seguente classe fornisce un modello per tutti i database, istanziati come singleton, che il software richiede:

```
public class Database
{
    private static Database theInstance = null;
```

```
    // Data Members
    private MioSingolo() {}

    public static Database getInstance()
    {
        if (theInstance == null)
        {
            theInstance = new Database();
        }
        return theInstance;
    }

    // Functions Members
}
```

Come si può notare la classe possiede un dato membro statico, quindi comune a tutte le istanze, appartenente alla classe stessa. In questo modo per ogni riferimento di classe, esiste un unico oggetto vero e proprio sul quale vengono effettuate le operazioni. Il costruttore è privato, vale a dire che non può essere invocato se non tramite funzioni appartenenti alla classe stessa.

Per ottenere il riferimento al database vero e proprio è necessario, quindi, richiamare la funzione **getInstance()**. Ciò è possibile perché la funzione è statica e quindi, oltre a poter elaborare dati membri statici (comuni a tutte le classi) non richiede un oggetto sul quale invocarli ma bensi basta conoscere il nome della classe. Si veda l'esempio seguente:

```
Database d = Database.getInstace();
d.method();
```

La funzione, appena viene invocata, verifica se è già stato istanziato l'oggetto statico **theInstance**. In caso affermativo, ne ritorna un riferimento. In caso negativo, vale a dire la prima volta che un componente ne richiede l'utilizzo, lo istanzia e ne ritorna il riferimento.

Classe DatabasePronomi

La classe **DatabasePronomi** fornisce un sistema per immagazzinare e consultare i pronomi memorizzati e riconosciuti dal sistema. I dati membro sono i seguenti:

- Map<Integer, Pronome> attivi
- Map<Integer, Pronome> passivi

La differenza è dovuta al diverso modo di concepire il pronome nel caso di attivo, soggetto dell'azione, oppure passivo, oggetto dell'azione. Tutti i pronomi vengono automaticamente inseriti in tale database già nel costruttore. L'inserimento avviene grazie alla funzione **add(Pronome p)** che verifica se il pronome è da inserire nella prima o nella seconda lista.

I pronomi sono inseriti nella mappa secondo una chiave che corrisponde alla tag assegnato ai pronomi per indicarne la persona. In questo modo la ricerca può avvenire proprio attraverso questo parametro getAttivi(int i), per ottenere il pronome attivo corrispondente, e getPassivi(int i) per ottenere il corrispondente pronome passivo.

Questa rappresentazione risulterà molto più chiara in seguito. L'utilizzo è reso necessario in quei casi in cui la frase non presenta un soggetto esplicito:

ho fame

Tale frase, infatti, non ha un soggetto ma è chiaro che chi ha fame è colui che parla, quindi *io* è soggetto.

Classe DatabaseVerbi

La classe **DatabaseVerbi** si occupa di organizzare e gestire i verbi riconosciuti e le loro coniugazioni. E' anch'essa un singleton e presenta i seguenti dati membri:

- Map<String, Verbo> elenco
- Map<String, VerboConiugato> verbi

La mappa **elenco** si occupa di raccogliere l'elenco di tutti i verbi, espressi in forma infinita da oggetti di classe *Verbo*. La chiave di inserimento e ricerca è il prefisso del verbo, la parte di stringa che rimane immutata in tutte le coniugazioni di verbi regolari. Si consideri il seguente esempio:

```
forma infinita: amare
prefisso: am
desinenza: are

esempi di coniugazione
io amo: (am + o)
ho amato: (am + to)
```

La mappa **verbi** associa la stringa corrispondente alla coniugazione a un oggetto di tipo VerboConiugato che lo rappresenta.

L'idea è quella di avere un unico oggetto che contenga tutti i verbi e tutte le coniugazioni e l'uso delle mappe aiuta a ridurre la complessità computazionale dell'algoritmo di ricerca.

La funzione **addInfinito(Verbo v)** si occupa di aggiungere un nuovo verbo in forma infinita nella lista. E' richiamata dal costruttore della classe Verbo.

La funzione **addConiugato(String s, VerboConiugato v)** aggiunge una forma coniugata di un verbo e la sua stringa.

La funzione **find(String s)** restituisce un oggetto di tipo *VerboConiugato* se la stringa passata come argomento corrisponde a una forma coniugata già vista in precedenza. I database vengono popolati ogni volta che viene istanziato un oggetto corrispondente.

La loro funzione è quella di evitare l'allocamento di ulteriori oggetti per l'elaborazione semantica e velocizzare il riconoscimento.

Infine, la funzione **findVerbo(String s)** verifica se è presente un verbo, in forma infinita, corrispondente alla parte di prefisso fisso per tutte le coniugazioni (es: amare = am + are).

4.3 Funzionamento del Parser

Per capire il funzionamento del parser è necessario iniziare con una descrizione molto generale per poi aumentare sempre di più il livello di dettaglio. La figura 4.5 mostra lo schema UML della classe **Parser**

Figura 4.5: Schema UML della classe Parser

Analizziamo, per prima cosa, i dati membro:

- Scanner read: si occupa di processare l'input per estrarre le stringhe provenienti dal parser vocale.
- LinkedHashMap<String, Symbol> dictionary: è un dizionario di termini conosciuti. Consente di associare un oggetto *String* (letto in input) ad un oggetto derivato da *Symbol* corrispondente. I verbi compaiono solo in forma infinita.
- DatabasePronomi pronomi: è un riferimento al singleton che si occupa di gestire i pronomi4.2.2)
- DatabaseVerbi dbverbi: è un riferimento al singleton che si occupa di gestire i verbi (4.2.2)
- Verbo avere: è un riferimento a un oggetto Verbo corrispondente al verbo ausiliare *avere*

- Verbo essere: è un riferimento a un oggetto Verbo corrispondente al verbo ausiliare *essere*
- List<Symbol> frase: è una lista di simboli corrispondente alla frase che si sta leggendo e analizzando.

Passiamo ora all'analisi dei metodi:

- Frase nextFrase(): si occupa di leggere un intera frase e restituire un oggetto di tipo *Frase*
- Symbol nextSymbol(): si occupa di leggere un simbolo dall'oggetto *Scanner*, identificarne il tipo corretto e inserire nella lista *frase* l'oggetto derivato da *Symbol* corrispondente.
- Symbol find(): si occupa di interrogare opportuni sistemi informativi che si occupano di catalogare i simboli per associarli a un corrispondente oggetto derivato da *Symbol*.
- Frase generateFrase(): si occupa di analizzare la lista di simboli *frase* per identificare i soggetti, il verbo e gli oggetti della frase, isolando anche gli aggettivi ad essi associati.

Per comprendere meglio il funzionamento del parser è necessario analizzare più approfonditamente le funzioni principali.

4.3.1 nextFrase

4.3.2 nextSymbol

4.3.3 generateFrase

Il funzionamento è molto semplice: ogni stringa letta viene identificata e inserita in una lista fino a che non viene notificato dal parser vocale la fine di una possibile frase che avviene nei seguenti casi:

- pausa significativa nella generazione di parole
- cambio di parlatore

Appena viene ricevuto tale messaggio, viene processata la lista di stringhe più volte al fine di isolare gli elementi principali utili all'analisi semantica: soggetto, verbo, oggetto dell'azione. Nel riconoscimento assume un ruolo fondamentale la grammatica: sulla base di essa, infatti, vengono ricercati gli elementi della frase in relazione alla posizione del verbo.

4.4 Risultati ed Esempi di Riconoscimento

Capitolo 5

Analizzatore Semantico

Bibliografia

[1] Google. Website. `http://www.google.com`.

[2] Wikipedia. Website. `http://www.wikipedia.org`.

[3] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach (2nd Edition)*. Prantice Hall, 2003.

[4] Armando Tacchella. Appunti del corso di linguaggi e traduttori, 2007.

[5] Alan M. Turing. Computing machinery and intelligence. *Mind*, 1950.

Indice analitico