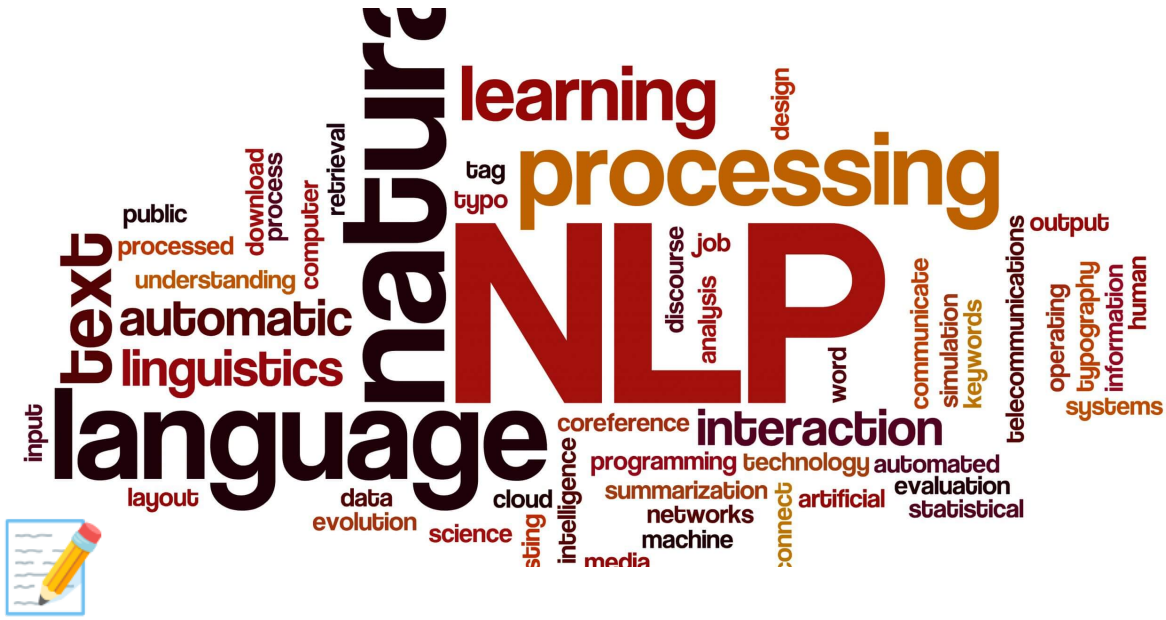


TLN [Mazzei]

> 💡 Questo è un riassunto / dispensa della 1° parte del professore Mazzei del corso di Tecnologie del Linguaggio Naturale (2022). Questo documento è stato definito facendo un *merge* tra le cose dette a lezione dal prof ed altri vari riassunti trovati nel repo.

> 💡 Preciso che potrebbero esserci errori di battitura e/o eventuali parti non ben scritte

> 💡 Questo documento è stato fatto con Notion, se lo usi e vuoi direttamente la pagina (per fare modifiche, ecc..), sono disponibile a passarla. Scrivimi → https://t.me/Aleeee96

▼ Introduzione

Elementi caratteristici del linguaggio umano

- **Discretezza** comunichiamo con elementi atomici, fonemi, morfemi e parole per delimitare il senso
- **Ricorsività** profondità della lingua umana riconosciuta dagli umani

> **Esempio** *Paolo crede che Maria voglia che Gianni veda Pietro*

- **Dipendenza dalla struttura** concetto di sequenza importante per il linguaggio, ma non è detto che una parola dipenda dalla sua precedente

> **Esempio** *I ragazzi di cui mi ha parlato la ragazza partono*

- **Località** la parola dipende dal contesto
- **Ambiguità** ci consente di esprimerci con frasi molto brevi, ma frasi ambigue sono mlto complesse da analizzare per un sistema di NLP

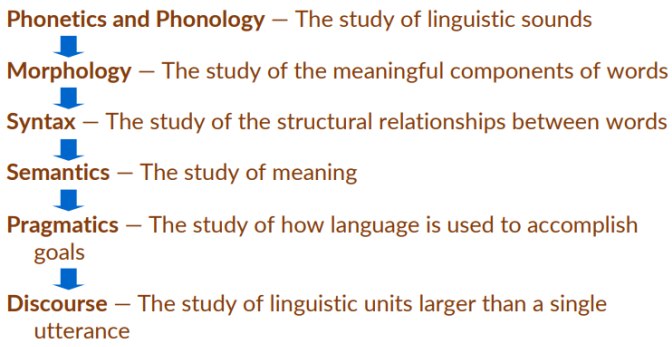
> **Esempio** *I made her duck*

1° ipotesi della linguistica computazionale

posso modellare, studiare ed analizzare il linguaggio a "*livelli*" analizzo singolarmente ogni livello con approcci, algoritimi differenti

2° ipotesi della linguistica computazionale

l'analissi del linguaggio viene visto come una pipelline dei livelli dove l'output del livello precedente fa da input a quello successivo (architettura a cascata), nel nostro cervello la pipeline non c'è quindi dal punto di vista cognitivo è debole questa visione

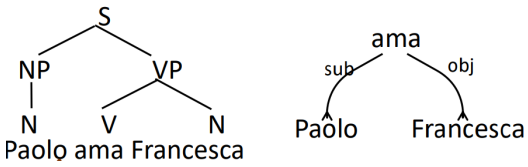


Le strutture dati dei livelli

- **Livello Morfologico e analisi lessicale** —> Lista, la sequenza quindi

> **Esempio** [Noun,Verb,Adj,Noun,Adv]

- **Livello Sintattico** —> Alberi

> **Esempio**
>
> 
>

- **Livello Semantico**
  - Semantica lessicale
    - Insiemi

> **Esempio** SynSet222={casa3,abitazione4,riparo1,rifugio7, …}

    - Vettori

> **Esempio**
>
> | | computer | data | result | pie | sugar |
> |---|---|---|---|---|---|
> | **cherry** | 0.0002 | 0.0007 | 0.0008 | 0.0377 | 0.0021 |

  - Semantica formale —> Logica - Alberi/Grafi (per ricorsione)

> **Esempio** *love(P,F)*

- **Livello Pragmatico e del discorso** —> Frame (Json) senza ricorsione

Se voglio catturare la ricorsività del linguaggio dovrò avere queste strutture dati ricorsive

Keywords

NLP	Natural Language Processing
CL	Computational Linguistics
Lexicon	Dizionario
Morphology	Morfologia
Syntax	Sintassi
Semantics	Semantica
Conversational Interface	Interfaccia di conversazione

Conversational agent	Agente di conversazione
Parsing	Parsing
NLG	Natural Language Generation
MT	Machine Translation
Grammar	Grammar
Treebank	Corpus di frasi annotate morfologicamente e sintatticamente
NL ambiguity	Attachment/coordination

▼ Livello Morfologico/Lessicale

Introduzione del livello Morfologico e l'analisi lessicale

L'analizzatore morfologico controlla la presenza di suffissi nella parola selezionata per capirne la forma morfologica.

> **Esempio** capitano (forma non declinabile, ambigua), capitan+o (nome o aggettivo o
> forma del verbo capitanare) oppure capit+ano (forma del verbo capitare)

Una parte fondamentale del livello morfologico è composta dai PoS. Essi sono le parti del discorso: nome, verbo, aggettivo, avverbio, preposizione, articolo, pronome, congiunzione, interiezioni. Le prime quattro (nome, verbo, aggettivo, avverbio) sono importanti dal punto di
vista del significato perché legate alla nostra cognizione e sono parole di contenuto (content
words). Le restanti invece sono dette function words, perché hanno una funzione di tipo linguistico e hanno un ruolo fondamentale nell'analisi sintattica

Definiamo quindi 2 tipi di relazioni:

- **Paradigmatico** relazioni tra gli elementi della frase e quelli che virtualmente potrerbbero
  alternarsi ad essi: se ad una frase cambio una parola, dal punto di vista semantico potrebbe non essere corretta, ma da quello grammaticale si.

  > **Esempio** *Il cane corre – La sedia corre*

- **Sintagmantico** relazioni che formano un sintagma

> **Esempio** articolo e nome sono in relazione sintagmatica

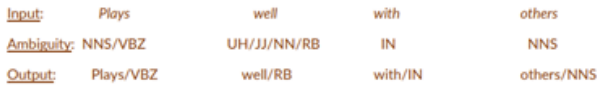
Le parti del discorso possono essere aperte o chiuse

- **Aperte** ogni giorno ne vengono inventate di nuove. In questa categoria rientrano le content word, quindi nomi, verbi, aggettivi, avverbi. Sono dell'ordine delle decine di migliaia.
- **Chiuse i**n questa categoria rientrano le function word, quindi preposizioni, articoli, pronomi,
  congiunzioni, interiezioni. Sono dell'ordine delle centinaia.

POS Tagging

Fa riferimento al livello **morfologico** ed è una tecnica che permette di rimuovere le ambiguità delle parole assegnando una parola ad una delle possibili categorie.

Nell'esempio la frase potrebbe essere molto ambigua in quanto la parola well può essere utilizzata in 4 diversi modi.



L'output del PoS viene utilizzato come input per altre fasi successive, come il parsing sintattico, che usa tale PoS per trovare la struttura sintattica della frase (soggetto, complemento oggetto, ecc.)

Difficoltà l'85% delle parole (function words: articoli, pronomi, congiunzioni ...) non sono ambigui, ma la restante parte sono parole usatissime (esempio: back parola usatissima ma estremamente ambigua).

Un set di POS si chiama un tagset, due esempi famosi di tagset sono

- **Penn Treebank**
- **Universal Dependencies treebank** (è più piccolo del Penn ma più moderno in quanto comprende emoji, simboli strani)

Algoritmi per il POS

Ci sono diversi algoritmi per effettuare PoS tagging

▼ Basati su regole

l'idea di base è la seguente

1. POS decontestualizzato: assegna tutti i possibili tag alle parole di una frase
2. Fase di Remove elimina tutti quei tag che non soddisfano determinate regole
    a. Se c'è un articolo determinativo, allora dopo non potremmo avere un verbo, quindi posso togliere il tag verbo nella parola successiva, oppure
    b. se la parola precedente non è un verbo e la successiva è un aggettivo allora possiamo eliminare tutti gli avverbi tra i tag possibili, ecc..

> **Esempio** di questi algoritmi è **ENGTWOL**. Queste regole sono solitamente scritte a mano oppure apprese con machine-learning

▼ Stocastici

Visione probabilistica del problema, basati quindi sulla statistica, cioè sulle osservazioni che si possono fare per una parola dato un corpus annotato con il PoS corretto fatto a mano.

> **Esempio** di questi algoritmi sono **HMM** e **MEMM**

Le fasi del POS stocastico sono

1. **Modelling** dare un modello formale probabilistico
2. **Learing** un algoritmo per settare i parametri del modello
3. **Decoding** per applicare il modello su uno specifico esempio

---

Vediamo ora due algoritmi stocastici

▼ HMM

È un algoritmo generativo (che usa la regola di Bayes). Si considera il problema del PoS tagging come un problema di Sequence Labeling, dove la migliore sequenza è quella che ha la probabilità più alta (quindi vogliamo massimizzare la probabilità).

▼ Fase di Modelling

L'obbiettivo è: data la sequenza di tag t_1^n trovare la massima probabilità della sequenza di tag, date le parole osservabili w_1^n.
Quindi un tag per ogni parola che massimizzi la probabilità di un tag rispetto ad altri date tutte le parole in ingresso

$$\hat{t}_1^n = \underset{t_1^n}{\operatorname{argmax}} P(t_1^n | w_1^n)$$

Per rendere l'approccio più semplice, applico quindi la regola bayesiana alla nostra formula di prima

$$P(x|y) = \frac{P(y|x)P(x)}{P(y)} \quad \Longrightarrow \quad \begin{array}{l} \hat{t}_1^n = \underset{t_1^n}{\operatorname{argmax}} P(t_1^n | w_1^n) \\ \updownarrow \\ \hat{t}_1^n = \underset{t_1^n}{\operatorname{argmax}} \frac{P(w_1^n | t_1^n) P(t_1^n)}{P(w_1^n)} \\ \updownarrow \\ \hat{t}_1^n = \underset{t_1^n}{\operatorname{argmax}} P(w_1^n | t_1^n) P(t_1^n) \end{array}$$

Applicando Bayes otterremo due probabilità $P(w_1^n \mid t_1^n)$ e $P(t_1^n)$ che sono più facili da calcolare, mentre il denominatore $P(w_1^n)$ lo buttiamo perché noi facciamo *argmax* dei tag non delle parole.

Quindi facciamo un'approssimazione (chiamata di *Markov*) che ci dice un tag dipende dal tag che lo ha preceduto, e che una parola dipende solo dal tag che l'ha emessa, per cui

- $P(w_1^n \mid t_1^n)$ sarà la likehood che dice quale può essere il valore del mio output in base al valore degli osservabili.
- $P(t_1^n)$ è quella a priori di quel tag ed approssimata sarà $P(t_i \mid t_{i-1})$ quindi la probabilità di un tag considerando il tag precedente

$$\underset{t_1^n}{\operatorname{argmax}} \prod_{i=1}^{n} P(w_i|t_i) P(t_i|t_{i-1})$$

▼ Fase di Learing

Il problema è quello di trovare, dato un corpus annottato, i parametri che caratterizzano l'equazione del modelling appena vista. Quindi andiamo a **contare** associando alla frequenza, la probabilità

Le due probabilità di prima ovvero $P(t_i \mid t_{i-1})$ e $P(w_i \mid t_i)$, vengono calcolate in maniera indipendente vedendo il corpus.

- **Quella a priori** $P(t_i \mid t_{i-1})$ [probabilità di transizione] si va a contare la frequenza all'interno del corpus del tag preceduto dal tag -1 e la si divide per il numero di volte in cui compare il tag-1, calcolo così una frequenza relativa che è appunto una probabilità

> **Esempio** $P(NN \mid DT)$ la probabilità di trovare un nome dopo un articolo sarà
> $\frac{count\,(articolo,nome)}{count\,(articolo)}$

- Per la **verosimiglianza** $P(w_i \mid t_i)$ [probabilità di emissione] conto quante volte, una certa parola, è stata etichettata con quel tag e la divido per il numero di volte che compare quel tag nel corpus.

> **Esempio** $P(is \mid VBZ)$ la probabilità che *is* sia un verbo
> $\frac{count\,(VBZ,is)}{count\,(VBZ)}$

▼ Fase di Decoding

Visto che vogliamo la sequenza che massimizza la probabilità, dovremmo provare tutte le possibili sequenze di tag associate ad una frase ma abbiamo però un'esplosione combinatoria delle possibili sequenze (con 30 tag, in una frase da 20 words avremmo 30^{20} possibili sequenze di tag). Quindi non è possibile approcciarsi al problema con un approccio all-path.

Tuttavia, non è necessario calcolare tutte le possibili sequenze, in quanto il nostro modello Markoviano è di di memoria 1, cioè vede solo lo stato precedente, per cui:

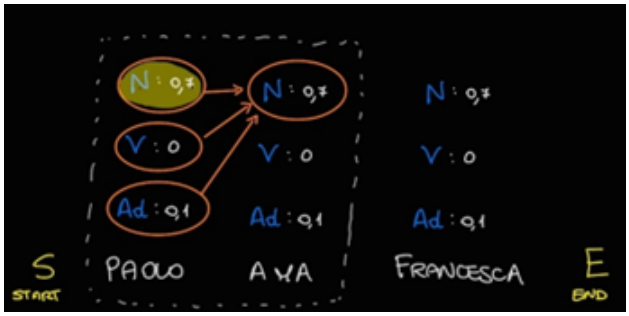
Soluzione: Programmazione Dinamica (Algoritmo di Viterbi)

L'idea è che tale programmazione dinamica ci permetta di "spaccare" il calcolo sulla mia equazione in una maniera tale da non ripetere gli stessi calcoli più volte. Man mano che analizzo la mia sequenza vado solo a memorizzare quale è la sotto sequenza (**prefisso**) con probabilità maggiore (questo ci permette di ridurre la complessità poiché non memorizzo tutte le possibili sequenze)

Quindi il punto chiave nella programmazione dinamica è che si può memorizzare solo il valore massimo per ogni cella senza la necessità di memorizzare ogni path quindi la fisolofia è

> 💡 è solo l'ultima cosa che succede prima a condizionare quello che succede ora

> **Esempio** il numero delle righe sarà il numero dei tag + la frase e ho una finestra di due colonne che si sposta a destra della frase



1. Prima calcoliamo la probabilità che **Paolo**
   - sia un nome
   - sia un verbo
   - sia un avverbio, aggettivo ecc..
2. Dopo vado a calcolare la probabilità che
   - **Paolo** sia un *nome* e **ama** sia un *nome*
   - **Paolo** sia un *verbo* e **ama** sia un *nome, ecc..*

   A questo punto seleziono il prefisso (tra i tag di **Paolo**) che ha probabilità più alta cioè ***Paolo*** = *nome* e così rimarrà quindi non cambierà

L'algoritmo di Viterbi

permette di fare quello appena descritto. Si crea una matrice con

- colonne : numero di parole della frase
- righe : numero dei possibili tag (che corrispondono ai possibili stati) + 2 (S [start] e E [end]).

L'idea è mettere una finestra che va a riempire una parte della matrice considerando solo 2 colonne per volta.

- **Fase iniziale (primo for)** $[\ S \rightarrow primaparola\ ]$
  - Riempie la prima colonna (quella di **Paolo**) che è la probabilità a priori dallo stato **S** ai tag di **Paolo** ovvero, che una frase inizi con un nome, con un verbo ecc..
- **Fase centrale (for annidato)** $[\ parola_{i-1}$ **->** $parola_i\ ]$
  - devo ciclare su tutti i possibili stati precedenti (**Paolo** nome, verbo,ecc..) e su tutti i possibili stati attuali (**ama** nome,verbo..ecc) della mia finestra in questione.

- **Fase di terminazione** $[parolafinale \rightarrow T]$
  - Quando la finestra sarà all'ultima parola e allo stato **T**, quindi la probabilità che quella parola sia di terminazione e quindi il suo adeguato tag, quindi anch'essa una probabilità a priori.

È importante dire che è presente sempre il **back-pointer** che ci dice quale stato precedente ha portato allo stato corrente e finché non arrivo alla fine non so quale sarà la sequenza di tag migliore, lo saprò appunto alla fine quando vado a prendere la sequenza con probabilità più alta grazie all'ausilio del back-pointer

La chiave di questa tecnica è che dobbiamo solo memorizzare il path di massima probabilità ad ogni colonna, con complessità finale quindi di $O(N^2)$.

▼ Problemi HMM

Se estendessimo la finestra dell'algoritmo a 3 spazi (**trigrammi**: quindi considerando la parola attuale e le due che la precedono), potremmo finire in problemi tra cui

- **Sparseness** ovvero che potrebbero esserci dei trigrammi che non compaiono mai nel nostro corpus

> **Esempio** NOME,NOME,NOME è possibile nella lingua italiana? Sì, ma magari nel mio corpus non c'è
>
> - Una soluzione può essere la tecnica dei lambda ovvero una tecnica di smoothing che al posto dello 0, mi restituisce un valore non nullo (filosofia: ciò che non ho visto non è per forza improbabile, quindi non metto 0 drasticamente)

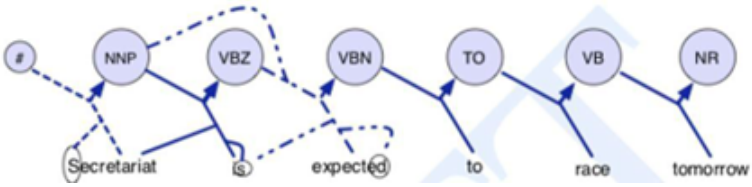
- **Complessità delle parole** come gestire lettere maiuscole e minuscole all'inizio della parola? Dovrei trattarle come due parole diverse

▼ MEMM

È un modello alternativo del HMM, ed è di tipo discriminativo. Mentre in HMM il ragionamento è: *se io ti dico che è un nome qual è la probabilità di Paolo?* in MEMM le parole condizionano direttamente lo stato, piuttosto come prima in cui lo stato condizionava l'apparizione di una parola.

Quindi non c'è una probabilità di verosimiglianza ma abbiamo una probabilità diretta (lessicale)

Il grosso vantaggio è nella possibilità di avere diversi tipi di feature che appaiono come osservabili come nell'esempio la S maiuscola o il fatto che la parola finisce con ed.



> **Esempio** La parola **petaloso** la feature (f_1) che mi dice se inizia con la lettera maiuscola è 0 (False), oppure (f_2) che mi dice se finisce con *“oso”* sarà 1 (True). Queste sono feature lessicali, quindi voglio lavorare su di loro, ovviamente ci sono feature più importanti o meno importanti; perciò, avrò feature pesate in base al loro grado di rilevanza.

▼ Fase di Modelling

Ci basiamo su un **features template** cioè la scelta di un insieme di feature booleane. Esempio la feature che ci dice se una parola finisce con “*oso*” ci aiuta a capire se quella parola sia un aggettivo? Se sì, allora la mettiamo nel feature set.

Una distribuzione del genere si può dimostrare che è di tipo esponenziale infatti la formula sarà $\frac{1}{Z}\exp\sum_i w_i f_i$

▼ Fase di Learning

Ottimizzare i pesi è molto complicato poiché ho bisogno di una massimizzazione che mi porti a scegliere dei pesi per le features che massimizzino la probabilità della giusta etichetta sul corpus di apprendimento.

Quindi devo fare delle approssimazioni quindi siamo di fronte ad un problema di ottimizzazione convessa, come il gradiente

▼ Fase di Decoding

Si utilizza anche qui la tecnica di Viterbi che modifica leggermente la probabilità da calcolare come la probabilità di un certo tag dato l’osservabile e i valori dello stato precedente

▼ Problemi MEEM

La **sparsness** è comunque presente anche qui, quindi se una parola è sconosciuta posso adottare questi approci

1. è probabile che sia un nome (tecnina di smoothing)
2. do a quella parola una probabilità uguale per tutti i tag possibili
3. posso guardare su dizionari esterni quella parola

4. adotto un euristica, ovvero vado a vedere la distribuzione di probabilità dei tag delle parole che compaiono una sola volta nel training, Esempio ho tre parole che compaiono una sola volta nel training: parola x con tag nome, y con tag nome e z con tag aggettivo. Allora quando incontrerò una parola sconosciuta (che per logica appare una sola volta in una frase) posso fare un'analogia con le parole che compaiono una sola volta nel training (cioè x,y e z) e quindi alla parola nuova darò tag = nome. (funziona molto bene, ma dipende dal corpus)

Vantaggi / Svantaggi HMM e MEEM

- Sul **Modelling** vince il MEEM perché è più particolareggiato potendo considerare più features rispetto a HMM dove invece guardiamo la parola precedente e quella in questione. Inoltre su MEEM posso aggiungere altre nuove features in base ai mutamenti della lingua
- Sul **Learning**, vince HMM perchè è molto semplice e veloce calcolare i conteggi mentre per MEMM l'elaborazione è più complessa e più lento (necessità di più risorse)
- Sul **Decoding** siamo pari perché comunque usano entrambi l'approccio di Viterbi

Poi c'è il **NER** ma non è il caso di riassumerlo secondo me

---

▼ Livello Sintattico

Introduzione

Le relazioni sintattiche sono relazioni che vogliono mettere in evivenza delle cose che esistono nella lingua, cioè che il soggetto e il verbo per quanto lontani nella frase sono collegati

Qua non trattiamo sequenze come nel POS perchè lì il fatto che abbiamo un nome implica che vicino ci sia un articolo, mentre qui il soggetto e il complemento oggetto possono essere distanti nella frase.

Chomsky si pose il problema della complessità del linguaggio naturale e quindi della struttura che caratterizza una frase linguistica (**struttura sintattica**).

Infatti nel linguaggio naturale abbiamo problemi di

- Ricorsione
- Ambiguità

Infatti **la sintassi** condiziona la semantica della frase, perchè si occupa

- della formazione delle frasi
- dell’ordine delle parole
- della formazione dei sintagmi (gruppi di parole che mostrano delle caratteristiche di tipo sintattico)

Per cui definiamo la

▼ **Competence**

è la grammatica in generale, il "sapere" questa grammatica, l'insieme delle proprietà intrinseche della lingua stessa (**esempio** nel caso dell’italiano ci direbbe che il soggetto viene prima del verbo)

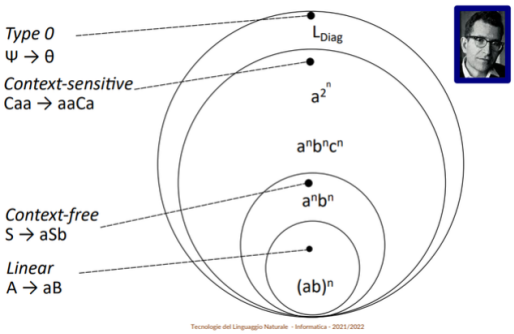
▼ **Performance**

è la capacità di un *ascoltatore* di comprendere la semantica di un discorso

- dove per gli umani, è il ricordarsi effettivamente il discorso con un livello alto di ricorsività
- per i computer è l'algoritmo di Parsing

Chomsky e la gerarchia

Chomsky dice che nella linguistica bisogna disinteressarsi delle performance e dare importanza allo studio delle competenze linguistiche, quindi ci si concentra su questa **linguistic theory** ovvero la grammatica formale (competence) disinteressandosi dell’aspetto algoritmico. L’idea è quella di usare una grammatica formale per descrivere il linguaggio naturale, come il concetto di riscrittura (cioè per esempio se trovo A vado sostituisco con BC [A→BC]).



Chomsky inoltre definisce la **grammatica generativa**, costituita da

- un alfabeto di simboli terminali (le parole della frase)

- un alfabeto di simboli non terminali (A,C,B, ecc..)
- un simbolo di start (s $\in V$)
- e una serie di regole di produzione (da A vado in BC)

Σ = alphabet

V = {A,B,...}

S ∈ V

P = {Ψ → θ,...}

L'idea è che sto generando una frase a partire dalla grammatica, ma il punto cruciale è capire che tipo di regole di produzione mi servono.

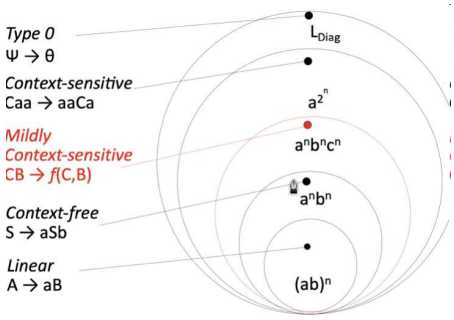
Quindi il quesito principale di Chomsky era capire quanto fosse complesso il linguaggio naturale e le strutture generate dalle grammatiche, ed ha stilato questa gerarchia

Ma dove si trova il linguaggio naturale? è lineare?

No, perchè poi lui stesso dimostra che la grammatiche lineari non sono un buon modello per le lingua naturali, e quindi ci si domandava se fossero CF o CS.

Poi Shieber disse che sicuramente non è neanche CF (dimostrandolo con una frase dialettica tedesca che indicava con una frase mista tedesca e svizzera che non era libera da contesto).

Quindi, le lingue naturali sono CS (Context Sensitive)?



In realtà sono in un livello di mezzo, infatti per **Joshi** le lingue naturali sono context sensitive ma solo leggermente al di sopra delle context-free quindi sono → **Mildly Context Sensitive**, con le seguenti proprietà

- non sono context-free (ma solo per poco)
- tutte le lingue naturali mostrano solo due tipi di dipendenze incrociate (**Nested** e **cross-serial**)
- non dobbiamo superare il tempo polinomiale nell'analisi
- tutte le frasi hanno la proprietà di crescita lineare (se prendo tutte le parole e le metto in

ordine crescente di lunghezza, non avrò mai dei salti ma avrò una crescita lineare)

Dalle MSC sono nate diverse grammatiche tra cui

- Tree adjoining grammars
- CCG (Combinatory Categorial Grammars)
- ed altre..

Tutte queste fanno una cosa simile ovvero potenziano leggermente le context-free cambiando leggermente le strutture elementari e le regole di combinazione / sostuzione. In particolare

▼ **Tree Adjoining Grammars**

abbiamo come

- **struttura** un'albero invece che una lista
- **regole di manipolazione** la
  - sostituzione (cambiare un nodo foglia con un albero
  - l'adjoining (l'innesto di un albero dentro un altro albero, ovvero l'utilizzo di strutture dati elementari per unire le parole nel senso tra di loro).

e questo aumentava il potere generativo

▼ **CCG**

- sono grammatiche bottom-up (mentre le grammatiche generative sono top-down)
- parto dalle foglie, ovvero le categorie.
- Ad ogni parola associo una categoria e la singola parola cerca altre categorie per combinarsi.

> **Esempio** Il verbo amare non sarà altro che un elemento che sta cercando qualcosa alla sua destra (soggetto) e qualcosa alla sua sinistra (complemento oggetto).

Parsing (a costituenti)

Ora vediamo come costruire un algoritmo di Parsing, vedremo prima uno a costituenti e poi uno a dipendenti

Ma prima

Qual è l’anatomia di un Parser?

è definita tra 3 elementi fondamentali

- La **grammatica usata dal Parser** [*Competence*], cioè la competence (CF, TAG, CCG, ecc..)
- L’**algoritmo** [*Performance*], che si dirama in due sezioni
  - la scelta della strategia di ricerca che ci indica le direzioni di partenza e di arrivo (top-down, bottom-up, left-to-right, ecc..)
  - l’organizzazione della memoria (back-tracking, dynamic programming...)
- L’**Oracolo** [E*uristica*] mi indica davanti a delle scelte quale scelta è migliore tra le possibili (probabilistico, rule-based, ecc..)

Utilizzando una grammatica context-free (per ridurre la complessità) testeremo due strategie

▼ **Top-down (parto da S, radice)**

ha lo svantaggio di generare molte possibilità che non servono a niente, dovendo partire dalla radice (tipo una frase imperativa dovrebbe essere trattata solo come imperativa, ma per il top-down non va subito bene dato che deve processarle tutte)

ha un altro problema aggiuntivo: la **ricorsione a sx** (tipica delle lingue naturali) che fa riempire la memoria con una struttura dati sempre più complessa, se non quasi interminabile (loop)

▼ **Bottom-up (parto dalle foglie)**

Qui genero cose che non diventeranno mai una frase, ma creo le varie possibili strade per le parole

Bisogna trovare una soluzione di parser più furba che mi permetta di ovviare a questo problema della **left-recursion**, oltre a quello di **ambiguità, i**n particolare, esistono due fenomeni che fanno riferimento all’ambiguità sintattica

- **attachment ambiguity**, dove la parola sta attaccata nell'albero di parsing?

One morning I shot an elephant in my pajamas.

- **coordination** capire se qualcosa fa riferimento sia a mangiare che bere o solo bere

Si può mangiare e bere qualcosa

A parte il problema di left-recursion, come risolviamo questi due problemi di ambiguità?

Per gestire tali problemi di esplosione combinatoria ritorniamo sulla **Programmazione Dinamica, v**ediamo quindi l'algoritmo CKY.

CKY (Cocke-Kasami-Younger)

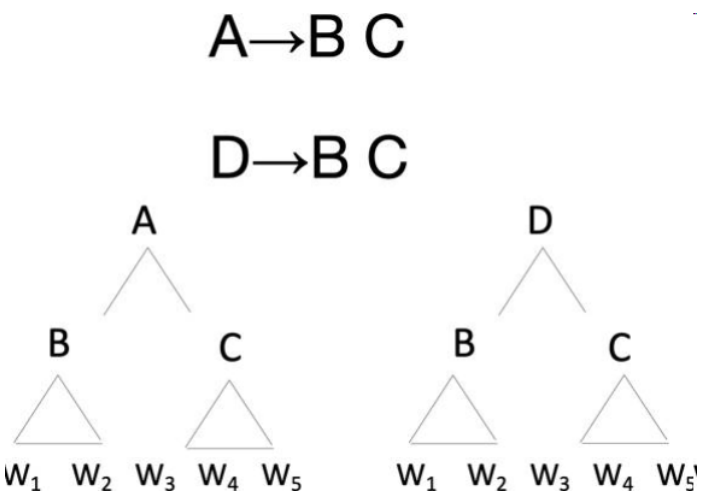
è un algoritmo di Parsing

- per le grammatiche **contex-free** generiche
- **bottom-up**
- con **programmazione dinamica** (e non back-tracking)
- euristica **rule-based**

L'idea del CKY

Se tu hai delle regole context-free tipo $(A \rightarrow BC, D \rightarrow BC)$ e avendo un sotto-albero con radice B e un sotto-albero con radice C, vanno entrambi bene per costruire un albero totale! Quindi gli stessi sottoalberi utilizzati per A saranno riutilizzati allo stesso modo per ricostruire D. Questi alberi pre-calcolati devono essere memorizzati dato che li riutilizzeremo

> 💡 La fisolosia è non riacalcolare quello che ho già calcolato



Quindi mi rendo conto che se con A non trovo la soluzione, verrà esclusa anche la regola anche per D

Prerequisito: dobbiamo convertire una grammatica in **forma normale di Chomsky** quindi

- NON deve esistere una regola tipo $A \rightarrow \epsilon$ (dove *epsilon* è uno spazio vuoto)
- Ma DEVE ma solo di tipo $A \rightarrow BC$ e $A \rightarrow terminal$

Matrice CKY

A questo punto posso costruire la struttura tipo di matrice utilizzata per incrociare i sotto-alberi corrispondenti alle regole di derivazione (non dovremo ricalcolare tutte le volte ma solo andare a "*trovare*" nella matrice i risultati parziali)

Esempio se considero la regola $A \rightarrow BC$ vuol dire che

- se c'è un A allora c'è un B seguito da un C
- se gli indici di A vanno da i a j allora ci sarà un k in mezzo che dividerà la parte di B con la parte di C quindi avrò il sottoalbero
  - B da i a k
  - C da k a j

Cosi facendo quando arriverà una regola che ho già visto di quella forma, ce l'ho già memorizzata.

Quindi la mia condizione di **succcesso** sarà che quando arriverò (partendo dal basso verso l'alto) il simbolo di start cioè **S** allora la frase è stata correttamente parsificata con la grammatica in questione

```
function CKY-PARSE(words, grammar) returns table

for j ← from 1 to LENGTH(words) do
    for all {A | A → words[j] ∈ grammar}
            table[j − 1, j] ← table[j − 1, j] ∪ A
    for i ← from j − 2 down to 0 do
        for k ← i + 1 to j − 1 do
            for all {A | A → BC ∈ grammar and B ∈ table[i,k] and C ∈ table[k, j]}
                    table[i,j] ← table[i,j] ∪ A
```

- Sostanzialmente quando l'**AND** delle tre condizioni finali corrisponde, allora abbiamo trovato il
  match corretto nella cella in questione
- Posso anche riempire con più di un oggetto la mia cella, ed avere *n* risultati all'interno

Complessità

La complessità è $O(n3)$ dove n è il numero di parole

- Non è cognitivo poiché gli esseri umani effettuano delle operazioni di pruning e back - tracking.
- Algoritmo troppo lento per il real time.
    - **Soluzione**: pruning probabilistico, elimino soluzioni poco promettenti.

CKY probabilistico

Devo cercare di eliminare possibili strade che non sono potenzialmente promettenti, cosa che succedeva nel CKY normale dove alla fine dovevo capire quale fosse la parsificazione migliore. Quindi perdo la correttezza a discapito di soluzioni più veloci e promettenti (potenzialmente)

> 💡 Infatti, questa tecnica è una che più si avvicina a quella che fa il cervello umano, infatti noi facciamo pruning delle soluzioni

è un algoritmo di Parsing

- per le grammatiche **contex-free** generiche (come per CKY normale)
- con **programmazione dinamica** + **beam search** (mi porto dietro non più di beam strade da seguire
- euristica **probabilistica**

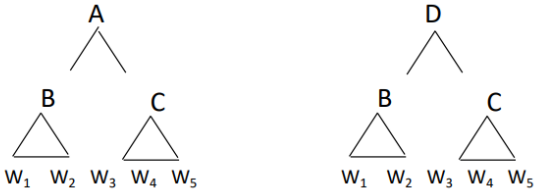
L'idea

- **bottom-up** (come per CKY normale)

voglio associare una probabilità ad ogni regola di produzione e tenere solo quella con probabilità più alta

$A \rightarrow BC \ [p_A]$ $P(1,4,A) = p_A * P(1,2,B) * P(3,4,C)$

$D \rightarrow BC \ [p_D]$ $P(1,4,D) = p_D * P(1,2,B) * P(3,4,C)$



Infatti usiamo questa probabilità per scegliere le regole giuste, non per scrivere meno regole.

Quell'indice k dell'algoritmo oltre a rendere il parser lento, andava ad aumentare l'ambiguità della mia grammatica.

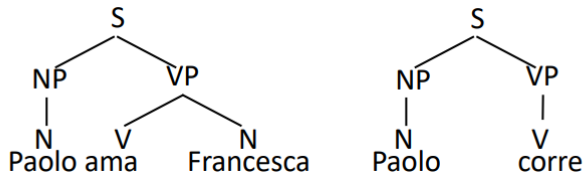
Ma come si calcolano le probabilità di un albero?

Per trovare tali distribuzioni di probabilità è possibile contare il nuemro di regole che compaiono e contando la distribuzione di esse nell'intero corpus (da banche dati come il **Penn Treebank**).

Esempio se devo cercare la probabilità della regola

$P \rightarrow (A \rightarrow \beta)$ allora $\forall$ albero nel TB conto

$(\frac{count(A \rightarrow \beta)}{count(A)})$



P(A→β)=Count(A→β)/Count(A)

P(S→NP VP) =2/2=1 P(NP→N) =2/2=1

P(VP→V N) =1/2=.5 P(VP→V) =1/2=.5

P(N→Paolo) =2/3=.66 P(N→Francesca) =1/3=.33

P(V→corre) =1/2=.5 P(V→ama) =1/2=.5

Algoritmo

è molto simile leggermente modificato per il discorso della probabilità.

La table questa volta è **tridimensionale**, quindi per ogni cella memorizza quali sono state le regole di produzione che hanno creato quel sotto-albero con la relativa probabilità associata al sottoalbero che va da i a j.

In questo modo memorizzo solo i sottoalberi con probabilità più alta, quindi adesso non è possibile ottenere in una matrice più soluzioni (come accadeva nel CKY normale).

Valutazione (Parseval)

Per valutare ci affidiamo alla

▼ Precision

Quanti sono i sotto alberi del mio albero originale che appartengono al gold tree

💡 quanto la soluzione trovata è giusta?

▼ Recall

Quanti sono i sottoalberi del gold tree appartengono anche al mio albero

💡 quanto della soluzione totale giusta viene ritrovata dal mio sistema rispettivamente?

Raramente si arriva ad una precisione totale e raramente tutta la soluzione da trovare viene
catturata dagli algoritmi. Per applicare queste misure in alberi di parser, le foglie devono essere
le stesse, si valuta i percorsi effettuati dalla radice alle foglie e le differenze tra i due.
Precision: Quanti sottoalberi del system appartengono anche al gold.
Recall: Quanti subtree del gold appartengono al system.

Problemi

Quando abbiamo frasi con probabilità molto simili ma una con senso nettamente sbagliato, vorremmo che le PCFG scegliessero questo anche in base al senso

Esempio

> Mangio la pizza con le forchette
>
> Mangio la pizza con la mozzarella

Per il Parser la diferenza è pochissima, quindi è difficle per lui strutturare le differenze.

▼ Si è provato quindi a Lessicalizzare la grammatica

Dove ad ogni simbolo metto una feature (**head**) che si ricorda la parola dominata dal quel simbolo
In questo caso allora ogni regola è potenziata con informazione. Quindi

- nella prima frase *forchetta* sarà legata con *NP* pizza
- nella seconda frase *mozzarella* è legata al verbo *VP mangio*

Svantaggio

avremo molte più regole prodotte

Parsing parziale (chunk parsing)

Faccio un'approssimazione ancora più pesante e dico che il linguaggio naturale non è neanche CF (ma non è vero secondo Chompsky)

Il concetto fondamentale è il **Chunk** come base sintattica che ha come proprietà il fatto di non essere ricorsivo (un chunk, rappresenta quindi una struttura che non ha altre strutture al suo interno)

> 💡 Quindi in questo caso rinuncio all'ipotesi di **ricorsività** per velocizzare il mio processo di parsing

> 💡 assomiglia al NER, posso utlizzare l'algoritmo del BIO tagging per fare questo

Che differenza c'è tra Chunk e Sintagma?

entrambi sono un elemento della frase ma

- il **sintagma** è ricorsivo (può contenere al suo intero altre strutture)
- il **chunk** non ha ricorsività,ma devo avere per forza diversi chunk per lo stesso concetto

---

Parsing (a dipendenze)

Caratteristiche di un Parser a dipendenze

- Termini possono essere **superiori** od **inferiori**, e quindi avere relazioni **master/slave**
- La differenza enoreme è che non ho più i simboli non terminali, quindi il mio albero ha $\#nodi = \#parole$

Perchè funziona l'approcio a dipendenze?

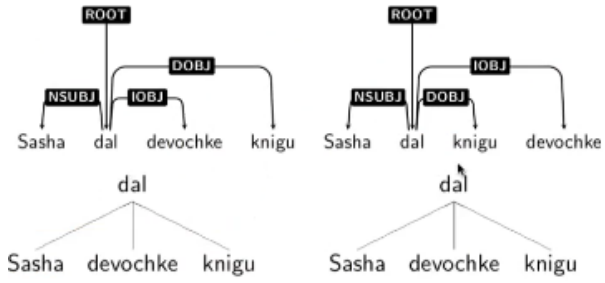
Le dipendeze piacciono perchè stiamo già togliendo ambiguità alla frase.

Esempio se diciamo che Paolo è il *soggetto* e zanzara il *complemento oggetto*, stiamo già avendo una buona informazione sulla frase, questa è chiamata **Information Extraction**

Quindi posso scoprire se c'è qualcosa che lega a qualcos'altro, perciò se c'è una relazione che collega zanzara e uomo, (però ovviamente non posso sapere se è la zanzara che punge l'uomo o il contrario), ma almeno so che sono in qualche modo legati nella frase.

Differenza tra Parsing a costituenti e a dipendenze

A differenza dei parsing a costituenti dove l'ordine delle parole conta, in quelli a dipendenze due frasi con stesso significato (ma con posizioni delle parole in posti diversi) non hanno differenze.



due frasi con posizione delle parole diverse, ma con le stesse dipendenze

Un ulteriore vantaggio è il fatto che si può passare facilmente da una frase in una lingua in un'altra perchè non si basa singolarmente sulla struttura della frase ma sulle dipendenze delle parole.

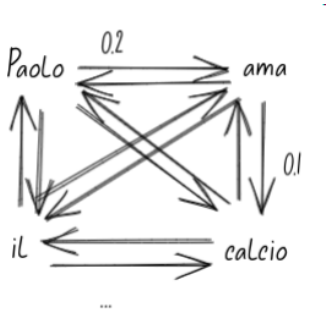
Algoritmi di Parsing a dipendenze

▼ 1) Programmazione dimanica

simile a quello di CKY ma il problema era che erano troppo complessi, fino a che un professore scopri un algoritmo con complessità $O(n^3)$. Per usare tale algoritmo l'idea è quella di trasformare un albero a dipendenze in un albero simile a quello a costituenti. Questo algoritmo però è troppo lento.

▼ 2) Algoritmo a Grafo

- si tilizza un Minimum Spanning Tree per la frase (quindi un grafo completamente connesso)
- poi valuto le frequenze di occorrenza delle parole in base ad un corpus e in base a ciò si costruiscono gli archi fra le parole con associato il peso ovvero la probabilità di connessione.
- Dopo con l'algoritmo MST si costruisce il percorso più probabile, nell'immagine disegnato in nero che permette di determinare l'albero a dipendenze della frase.



- Dopo con un classificatore posso tipare gli archi con le tipologie di connessioni.

▼ 3) Parsing a costituenti e conversione

- Parto da un albero a costituenti qualsiasi
- cerco di convertire tale albero in un albero a dipendenze sulla base di una conoscenza di tipo linguistico che mi dice appunto l'equivalenza tra i costituenti e le dipendenze.
  - La linguistica funziona come un'euristica, memorizzata sottoforma di tabelle di percolazione, come fossero regole linguistiche, che permettono di capire da un albero a costituenti le dipendenze e la tipologia delle dipendenze.
- È una specie di tecnica di lessicalizzazione che prende il nome di X-tag.

▼ 4) Parsing deterministico

Faccio scelte greedy per creare dipendenze tra parole, guidate da machine learning classifiers

▼ 5) Soddisfazione di vincoli

Elimino tutte le possibili dipendenze che non soddisfano certi vincoli

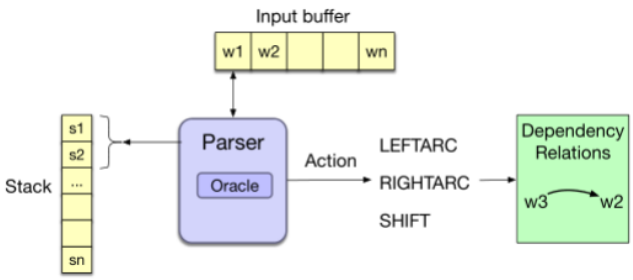
MALT (Parser deterministico a “*transizioni*”)

- **Grammatiche** a dipendenze
- **Algoritmo** bottom-up
- **Memory Organizzation** depth-first (non deve fare backtracking dato che è probabilistico)
- **Euristica** probabilistico

Caratteristiche

Si dice *deterministico* per il fatto che data una qualsiasi sequenza di parole si avrà sempre una soluzione ovvero un albero in uscita, a differenza del CKY, in cui il parser può fermarsi non appena si rende conto che non è derivabile.

L’idea



💡 Ipotesi di proiettività: solo per questi esempi utilizziamo alberi proiettivi, ovvero alberi con dipendenze senza incroci

Strutture usate

- Uno **stack** per le parole **attualmente analizzate** (*stack*)
- Una **lista** contenente le **rimanenti parole** da analizzare (*input buffer* o *word list*)
- Un **insieme** contenente le **dipendenze** create fino a quel punto dell'analisi (*relations*)

💡 Il concetto fondamentale è che le **3 strutture**, ad ogni instante, definiscono uno **stato**

Definiamo quindi i 2 **stati** fondamentali

- **Iniziale** composto da
  - **Stack** vuoto (root)
  - **Buffer** pieno, contenente tutte le parole in ingresso (la frase)
  - L**ista di dipendenze** (inizialmente vuota) contenente l'insieme di tutte le relazioni trovate fino a quel momento
- **Finale** composto da
  - Stack vuoto (root)
  - **Buffer** vuoto (tutte le parole sono state analizzate)
  - L**ista di dipendenze** riempita, contenente l'insieme delle relazioni

- Stato iniziale
  - [[root], [sentence], ()]
- Stato finale
  - [[root], [] (R)]
  - Dove R è l'insieme delle dipendenze costruite. [] è la lista vuota poiché tutte le parole sono state analizzate

Funzionamento

Il parser attraversa la frase da sinistra a destra, successivamente sposta gli item dal buffer allo stack.

Ad ogni passo analizziamo i 2 elementi in testa allo stack e l'oracolo fa una decisione sulla **transizione** da applicare per costruire il parser.

> Le possibili transizioni corrispondono alle azioni intuitive che uno dovrebbe fare per creare un dependency tree esaminando le parole in una sola passata sull'input da sinistra a
> destra.

> Esempio

Avremo quindi 3 **azioni (operatori)** possibili

▼ Shift

prende la prossima parola dalla lista delle parole in input e la pusha sullo stack (rimuovendola dalla lista)

Esempio

$$[\,[root],\ [I\ booked\ a\ morning\ flight],\ ()\,]$$

diventa

$$[\,[root,\ I],\ [booked\ a\ morning\ flight],\ ()\,]$$

▼ Left

fa due cose

1. crea una dipendenza(a, b) tra prossima parola della lista (a) e la parola sul top dello stack(b)
2. poi fa pop dello stack

Esempio

$$[\,[root,\ I],\ [booked\ a\ morning\ flight],\ ()\,]$$

diventa

$$[\,[root],\ [booked\ a\ morning\ flight],\ (booked,\ I)\,]$$

▼ Right

fa tre cose

1. crea la dipendenza (b, a) tra la prossima parola della lista (a) e la parola sul top dello stack (b)
2. rimuove (a) dalla lista
3. poppa lo stack e mette la parola poppata all'inizio della lista

Esempio

$$[\,[root,\ booked],\ [flight],\ ()\,]$$

$$[\,[root],\ [booked],\ (booked,\ fligth)\,]$$

Algoritmo

è "*semplice*" perchè tutto il lavoro lo fa l'oracolo

- Inizializza la frase
- fin quando non arrivo allo stato finale chiedo all'oracolo cosa devo fare
  - poi applico una transizione
- ritorno le dipendenze che ho calcolato

> 💡 Come detto in precedenza, questo algoritmi ritornano UNA sola sequenza possibile per la costruzione dell'albero della frase

Problemi

1. Quella che abbiamo visto è la versione non *tipata* ovvero che non restituisce etichette ma crea solo dipendenze
   - **Soluzione** Invece di usare solo *Shift* e *Right* a questi aggiungo l'etichetta. Esempio *Left_subj*, R*igth_subj*, ecc… . Quindi avrò un numero di operatori $= 2 \cdot \#etichette + 1$ perchè l'operatore *Shift* mica lo puoi tipare
2. Non c'è scritta da nessuna parte la sequenza di operazioni da fare, quindi **come** e **dove** apprende l'oracolo per arrivare a fare consigliare quelle scelte all'algoritmo?
   - **Soluzione** voglio apprendere un **classificatore** per ogni transizione. Per farlo dal Treebank, utilizzo Machine Learning per capire le feature che caratterizzano ogni stato. Ciò comporta:
     - a capire quali sono le features linguisticamente significative. Esempio se sul top dello stack c'è un *articolo* e la prossima parola da predenre è un *nome*, allora dovrò un *Left*
     - a costruire il training data, attraverso un dependency treebank, poi si fa **reverse engineering**, ovvero data la frase e l'albero finale, vado a trovare le regole che portano la frase nell'albero
     - Utilizzare un buon training algorithm, perchè la parte di ML (prima citata) è relativa all'apprendere i pesi che massimizzano lo score della transizione corretta per tutte le configurazioni nel training set.

TUP (Parser a regole per dipendenze a vincoli)

Grammatiche a dipendenze (*vincoli*)

- **Algoritmo** bottom-up
- **Memory Organizzation** depth-first

- **Euristica** rule-based

Idea

Questo sistema divide il parsing in 3 fasi:

- **Chuncking** la frase in ingresso viene suddivisa in chunk
- **Coordination** rimuovo l'ambiguità delle congiunzioni con delle regole
- **Verb sub categorization** una volta ottenuti i chunk questi vengono collegati ai verbi sfruttando una tassonomia/gerarchia di regole, ossia delle regole che hanno delle priorità
  su altre.

> L'ha spiegato letteralmente in 1 minuto

Valutazione

Per valutare questi algoritmi di parsing non ho più bisogno di *Precisin* e *Recall*, perchè come detto all'inizio il $\#nodi = \#parole$

Quindi qui valutiamo con una metrica chiamata **labelled attachment score**

Tabella riassuntiva Parser

Parser	Grammatica (*Competence*)	Algoritmo (*Performance*)	Oracolo (euristica)	Complessità
CKY	Context-free	Bottom-up left-to-right dynamic programming	Rule-based	$O(n^5)$ fino a $O(n^3)$
CKY (probabilistico)	Context-free	Bottom-up left-to-right dynamic programming	Probabilistic	$< O(n^5)$
MALT	Dependency-grammar	Bottom-up depth-first no-backtracking	Probabilistic	*
TUP	Dependency-grammar	Bottom-up depth first constraint-based	Rule-based	*

▼ Livello Semantico

Introduzione

Parliamo di Semantica, quindi abbiamo a che fare con il significato.

- **Non** più chi è il soggetto e chi il complemento
- **Ma o**ra voglio scoprire chi fa un'azione

Come rappresentiamo la semantica?

con la **FOL** (logica del prim'ordine), questo perché, oltre che rappresentare bene la logica dei verbi, permette di esprimere anche il determiner.

> **Esempio** *"Paolo ama Francesca" sarà* $love\,(X, Y)$

Ovviamente non sapremo cosa vuol dire *"love"* ma in questo senso ci interessa la rappresentazione, non come verrà effettivamente fatto.

La FOL ci dice che quella è la definizione di *Palo ama Francesca* ma non mi dice come ci arrivo a quella formula, per questa motivazione introdurremo il **lambda-calcolo**, per risolvere il problema del calcolo della semantica. Lo facciamo principalmente perché poi vorremmo fare inferenza sulla nostra KB

CS (computational semantic)

Algoritmo che si basa sul **principio di composizionalità di Frege**

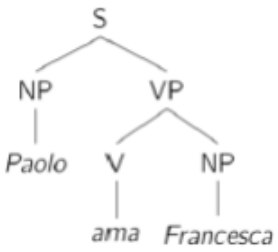
> 💡 il significato del tutto è determinato dalle parti e dalla maniera in cui sono combinate

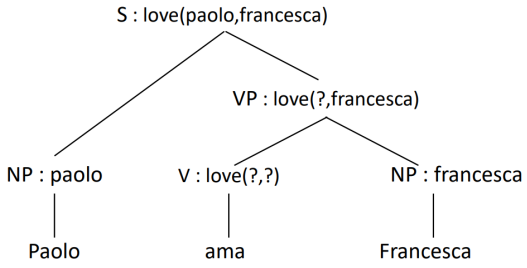
- Parsifico la frase per ottenere l'albero sintattico (dei costituenti)
- Cercare la semantica di ogni parola nel lessico
- Costruire (con un approccio bottom-up) la semantica per ogni sintagma secondo il principio di Frege prima di indicato

> **Esempio**

Considerando la frase di prima ed ottenuto l'albero sintattico, sappiamo adesso che il nostro obbiettivo è estrapolare un predicato $amare(Paolo, Francesca)$ che ci indica il collegamento tra i due nomi e il verbo

$\rightarrow$





In questo modo, partendo dalle foglie, mi rendo conto che *Paolo* e *Francesca* sono due costanti,
mentre *ama* rappresenta il predicato $love(?,?)$ dove i punti interrogativi indicano per ora che il
predicato *ama* ha bisogno di due costanti per essere completato

Essendo il verbo *amare* asimmetrico, ci aspettiamo che *Francesca* venga mappata sul "*paziente"*, (cioè sul secondo punto interrogativo) ma non sempre il paziente è il secondo a comparire nella frase (nelle frasi passive ad esempio).

A questo punto dobbiamo quindi

- definire in maniera rigorosa termini come $love(?, Francesca)$ o $love(Francesca, ?)$
- capire come combinare i pezzi

Lo scopo, per risolvere questo problema, è segnalare per ogni relazione chi è il paziente e per far questo abbiamo bisogno di un altro ingrediente il **lambda calcolo**

$\lambda - calcolo$

λ è un operatore che ci consete di rendere dimanica la FOL (che è statica) infatti ci consente di indicare/segnare le variabili che non hanno ancora un argomento, cioè l'informazione mancante

La funzione lambda è fatta cosi $\rightarrow$

$\lambda x.love\ (x, Mary)\ \ (john)$

Per far cio si utilizza la Beta reduction, che funziona in 3 passaggi

1. elimino il λ cosi ottengo $love\ (x, Mary)\ \ (john)$
2. rimuovo l'argomento a destra cosi ottengo $love\ (x, Mary)$

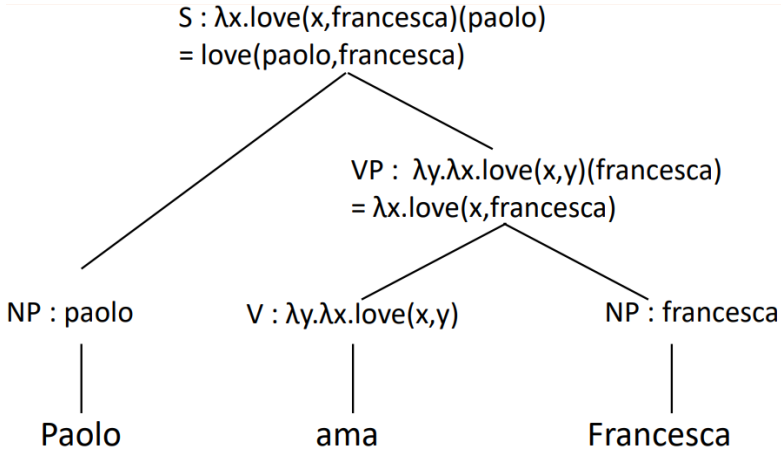
3. rimpiazzo tutte le occorrenze della variabile che prima era legata a λ, con l'argomento e quindi ottengo $love\ (john, Mary)$

Cosi facendo stiamo continuando ad usare la FOL la la rendiamo più potente unendola al lambda calcolo Lambda-FOL anche chiamata

> **Semantica di Montague**
>
> - che ci permette di rappresentare parole e sintagmi mancanti, e si basa sull'idea che non esista una gran differenza teoretica tra il linguaggio naturale ed il linguaggio artificiale logico. Possiamo usare principi matematici e regole logiche per
> analizzare qualsiasi lingua.
> - Si basa sul principio di composizionalità

Riprendendo l'esempio di prima, ora possiamo vedere come risolverlo correttamente con questa nuova tecnica perchè il nostro albero verrà rappresentato diversamente

Ora funziona bene perchè *Francesca* viene correttamente messa nella *y* (perchè con lambda rispettiamo l'ordine)



> **Importante**
>
> L'ordine delle lambda è fondamentale, poiché permette di sapere quale è il *paziente* nella frase attiva piuttosto che nella frase passiva.

Come fa il sistema a capire se il valore è a destra o a sinistra?

Nell'approccio di Montague lo andiamo a scrivere nelle regole di **composizione** (ecco perchè l'approcio segue il principio di composizionalità), cosi possiamo capire quale sia la *funzione* e
quale sia *argomento*

In quest'immagine

- per un VP, la funzione viene da sinistra e l'argomento invece da destra.
- Mentre nell'altra regola quello che viene da NP è l'argomento e quello che viene da VP è la funzione

Regole di composizione

VP : f(a) -> V : f NP : a

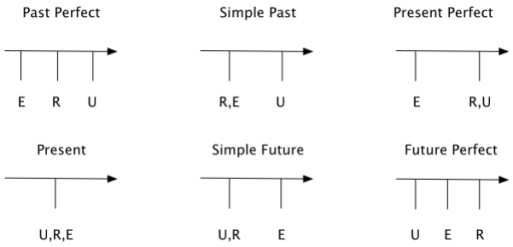
S : f(a) -> NP : a VP : f

queste regole sono grammatiche a cui è stata aggiunta quest'informazione in più (dette anche **augmented CFG** o grammatiche ad attributi).

Ma comunque, stiamo considerando solo il tempo verbale presente, e gli altri tempi verbali?

Si dovrebbe ricorrere ad una struttura temporale, rappresentando ogni tempo verbale con tre tempi

- **U** utterance (quando viene detta la frase)
- **R** reference time
- **E** time of the event



Quindi se avessi solo soggetto verbo e complemento oggetto potremmo aver finito qui con quanto appena descritto, ma il linguaggio ha altre cose: articoli, proposizioni, avverbi ecc..

Come gestiamo gli articoli?

Gestire gli articoli per esempio è assai più complesso rispetto ai verbi o nomi. Questo è dovuto al fatto che nomi e verbi sono vocaboli *"di contenuto"* e dunque si spiegano da soli indipendentemente dal contesto. Questo, invece, non si può dire per gli articoli invece, che dunque hanno una semantica più complicata.

Proviamo a vedere una frase con l'articolo *UN*

> Esempio *Un uomo ama francesca*

Ragioniamo su due aspetti

▼ Astrazione

come prima idea ci viene quella di utilizzare l'approccio degli operatori esistenziali $(\exists)$. Solo che "Un uomo" non risulta essere una formula ben formata, dato che il significato dell'operatore
esistenziale è diverso da quello espresso dalla frase cioè *"Un uomo"*, infatti, significherebbe più *"esiste un uomo"* o *"c'è un uomo".* Per risolvere ciò, l'idea è di partire dalla fine anziché dall'inizio (**reverse engineering**).

Partendo dalla fine quindi, cerco di capire come devono essere fatti tutti i componenti e facendo **astrazione** sui predicati otteniamo la formula dell'articolo *un*

$$\lambda P.\lambda Q.\exists z(P(z) \wedge Q(z)) \rightarrow un$$

è come se stessi dicendo che quando c'è l'articolo *un* allora ci sarà un NP alla destra (al 99%) quindi stiamo già prevedendo un predicato binario in sostanza.

▼ Type raising

A questo devo devo aggiongere anche **type raising** (che inverte la funzionalità di chi sta a sx con chi sta a dx) dicendo che Paolo è un predicato che arriva da sinistra dall'albero ma si applica come argomento nella funzione ama che arriva da destra

Per cui grazie a questi due passaggi, alla fine avrò modellato tutti i componenti, quindi gli articoli avranno una propria definizione, i nomi propri un'altra, i verbi un'altra ancora, ecc..

uomo	λx.man(x)
Paolo	λP.P(Paolo)
corre	λx.run(x)
ama	λR.λx.R(λy.love(x, y))
un	λP.λQ.∃z(P(z) ∧ Q(z))

💡 La proprietà **importante** che voglio mantenere (oltre alla composizionalità) è la **sistematicità** ovvero che il nostro modello funzioni su tutte le possibili frasi, e non solo su una ad-hoc (ovviamente)

Qual è la differenza tra ambiguita sintattica e ambiguità semantica?

> **Esempio**
>
> **Frase 1**: *Tutti gli uomini amano una donna* (*ambiguità semantica*)
>
> **Frase 2**: *Mangio la pizza con le acciughe* (*ambiguità sintattica*)

La seconda frase è sintatticamente ambigua e anche semanticamente ambigua, ma per ogni albero sintattico io ho una sola lettura semantica.

Invece nella prima ho ambiguità solo a livello semantico (e non sintattico) però ho due possibili letturedi FOL

- $\forall x(man(x) \rightarrow \exists y(woman(y) \land love(x,y)))$
- $\exists y(woman(y) \forall x(man(x) \rightarrow love(x,y)))$

▼ **Ambiguità sintattica**

abbiamo alberi sintattici diversi quindi non da problemi a livello semantico

▼ **Ambiguità semantica:**

abbiamo lo stesso stesso albero sintattico, quindi significato ambiguo

Come gestiamo gli avverbi?

> **Esempio** *Paolo ama Francesca dolcemente*

Potremmo definirlo cosi $love(P, F, sweetly)$ ma se ho più avverbi?

Allora la soluzione definitiva è la reificazione dell'evento ovvero definire una variabile che descrive l'evento. Quindi diciamo che esiste un certo evento e poi lo mettiamo in fondo in *and* in questo modo

$\exists e \;\; love(e, P, F) \land sweetly(e)$

più preciso sarebbe indicare l'agente e il paziente in questo modo

$\exists e \;\; love(e) \land agent(e, P) \land patient(e, F) \land sweetly(e)$

(questo si chiama **stile neo-Davidsonian**)

▼ Livello Pragmatico e del discorso

Introduzione

Fino ad ora abbiamo sempre e solo parlato di analsisi del linguaggio, perchè la parte di analisi di una frase è sempre stata più studiata, in quanto più ambigua. Invece ora parliamo di generazione del linguaggio (NLG).

NLG

Processo di costruzione deliberativa di testo in linguaggio naturale al fine di perseguire un goal comunicativo

> **Planning** (pianificazione) come concetto fondamentale della NLG

La situazione è questa, abbiamo/dovremmo avere

- **input** una forma di rappresentazione dell'informazione (tabella SQL o FoL), un insieme prefissato di frasi, ecc..
- **output** vogliamo fornire frasi, documenti, spiegazioni, messaggi ecc.
- una **KB** perchè affinché questo sia realizzabile serve una conoscenza
- **dominio** poiché a seconda del dominio utilizzerò slang diversi.

Potremmo dire che il natural language processing si compone di due parti diverse

- **Natural language understanding** con cui comprendiamo quello che ci viene detto
- **Natural language generation** con cui produciamo qualcosa in output in base a quello che
  ci è stato detto

Dall'analisi alla generazione della frase

Come detto prima la NLG ha sempre avuto meno rilevanza rispetto all'analisi del linguaggio, perchè per definizione, l'input che diamo all'NLG è non ambiguo (sono frasi giuste, in teoria). La generazione, essendo più semplice, è quindi meno interessante dal punto di vista scientifico.

Nell’NLG il problema quindi non è più l’ambiguità, ma diventa quello di generare delle frasi come le
genererebbe naturalmente un essere umano, il quale utilizza meccanismi molto complessi pe
farlo.

▼ I Template

All’inizio di è pensato ai **template**, perchè

Pro

- sono facili e veloci da gestire

Contro

- c'è poca formalizzazione
- non facciamo uso di linguistica

Inoltre, la tecnica dei template, ci darebbe come output ripetizioni di frasi simili e ripetitive anziché utilizzare la forma elenco come farebbe l’uomo

> **Esempio** direbbe ho visto L, ho visto M, ho visto P anziché dire ho visto L, M e P).

▼ I sistemi NLG

rispetto ai Template

Pro

- sono è più mantenibili
- hanno una qualità testuale molto superiore
  - grazie all’utilizzo di ricorsività, quindi allla vicinanza al vero linguaggio naturale).

Contro

- sicuramente più lenti

Nella storia ci sono stati molti approcci ibridi.

> **Esempio** *“BabyTalk"*. L'idea è che, a fronte dei dati medici provenienti da dei macchinari, il sistema di NLG produca in automatico tre documenti diversi pensati ognuno per un soggetto diverso
>
> - uno per i medici
> - uno per le infermiere

> - uno per i familiari del paziente
>
> Per realizzarlo è stato chiesto ai medici come avrebbero scritto una serie di dati provenienti dai macchinari e in base a queste considerazioni venivano prodotti i documenti.
> Per fare ciò, è stato chiesto a dei dottori di verbalizzare valori dati dai sensori, quindi di creare dei corpus, allo stesso modo è stato chiesto alle altre categorie. In base a questi corpus, BabyTalk è in grado di rendersi conto come deve descrivere le informazioni sulla base appunto dei corpus assegnati. In questo modo si ha un testo adatto a chi dovrà leggerlo.

Quali sono i task dell'NLG?

Nella suddivisione dei task intrapresi si va dal significato al testo (da 1 → 7)

▼ 1. Content Determination

siamo molto vicini al significato e lontani dal linguaggio. Decido cosa dire, il messaggio che voglio esprimere. Sono basati su entità, concetti e relazioni sul dominio. Creo una struttura dati, come fosse il risultato di una query ed esprime le parole che io voglio comunicare.

> **Esempio** *treno, destinazione e orario di partenza.*

▼ 2. Discourse Planning

devo capire quale relazione causale c'è tra i messaggi scelti al passo precedente, queste relazioni possono essere

- **Concettuali** le due frasi sono affiancate poiché parlano dello stesso argomento, ma
  non c'è un legame fra loro.

  > **Esempio** *Il prossimo treno è il treno A e parte alle 10*.

- **Retoriche** una delle due frasi è elaborazione dell'altra.

  > **Esempio** *Il treno A parte alle 10 ed è uno dei 20 treni per Glasgow*

▼ 3. Sentence Aggregation

decide quanto compatto deve essere il testo, ovvero decido quante frasi usare per comunicare i miei messaggi. I messaggi devono essere combinati per poter produrre frasi più complesse ma anche più simili alle frasi umane/naturali

> **Esempio**
>
> - senza aggregazione
>   - *The next train is the Caledonian Express.*
>   - *It leaves Aberdeen at 10 am*
> - con aggregazione
>   - *The next train, which leaves at 10 am, is the Caledonian Express*

▼ 4. Lexicalisation

permette di decide quali parole utilizzare per esprimere i concetti del dominio e quali strutture sintattiche utilizzare per collegare le varie parole.

> **Esempio**
>
> nel contesto dei treni, la scelta delle parole è importante infatti “*partire”* va bene sia per aerei che treni ma *“decollare”* va bene solo per gli aerei.

Per le strutture sintattiche invece, dovrò decidere se utilizzare una frase attiva o una passiva.

> **Esempio**
>
> “I*l cane morde l'uomo”* o “*L'uomo è morso dal cane”*

▼ 5. Referring expression generator

decidiamo come riferirci ai nomi propri. È infatti il momento in cui dobbiamo cominciare a riferirci non solo alle classi ma anche alle istanze delle stesse (nomi propri per l'appunto)

> **Esempio**

> *"Alessandro Mazzei"* e *"Il professore che tiene la lezione di TLN"* si riferiscono alla stessa persona. Inoltre, invece di ripetere il soggetto, potremmo utilizzare dei pronomi per rendere la frase più naturale.

▼ 6. Syntactic and morphological realisation

dato come input l'albero sintattico a dipendenze genero la corretta sequenza morfologica rispettando la grammatica della lingua naturale selezionata

> **Esempio**
>
> sulle regole morfologiche dato che devo dare il passato alla frase, metto *-ed* dopo *walk* per fare *walked*.
>
> sulle regole sintattiche il soggetto va messo prima del verbo. Quindi la frase *“John walked”* è meglio di *“Walked John”*

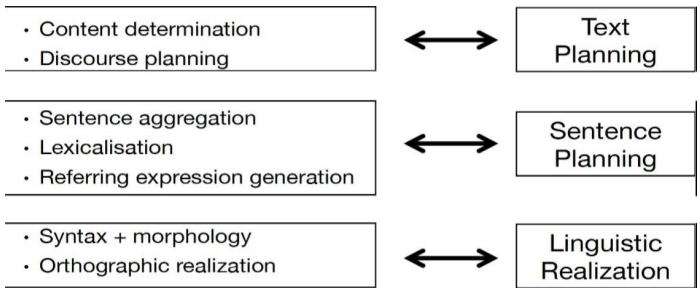
▼ 7. Orthographic Realization

scelgo l’ortografia (punti, spazi, ecc..). Quindi seguo le regole ortografiche

> **Esempio**
>
> le frasi cominciano con la lettera grade, le dichiarative finiscono con un punto, per le domande ci va il punto interrogativo, ecc..

Come è strutturata la NLG?

Ognuno dei 7 task descritti prima segue una struttura dell’NLG che forma una pipeline (l’output del precedente è l’unput del successivo)



Perchè abbiamo queste 3 macro fasi?

Perchè quando genero un testo con NLG uso 2 fonti di conoscenza (con ruoli completamente diversi)

- fonti sul dominio
- fonti sul linguaggio

Infatti nel

▼ Text Planning

- Il dominio in questione è importante (parlo di calcio, scienza, ecc..)
- la conoscenza sul linguaggio no, non gioca nessun ruolo, per cui non ci interessa il linguaggio (se parlo in inglese o in italiano non mi interessa, in questa macro-fase)

▼ Sentence Planning

dipende sia dal dominio che dalla lingua, perchè per decidere che parole usare devo capire che lingua sto trattando e per capire che termini devo usare mi serve anche il dominio in questione

▼ Linguistic Realization

Al contrario della prima macro-fase, qui

- il dominio non c'entra
- il linguaggio da usare si

Fase	Task	Struttura Dati	Output
Text Planning	1 | 2	Logica, Formule	Text Plan
Sentence Planning	3 | 4 | 5	Albero	Sentence Plan(s)
Linguistic Realization	6 | 7	Array, Sequenza, Frase	Frase

Sistemi di dialogo

Fino ad ora abbiamo visto tutta la pipeline di analisi fino alla generazione del linguaggio

Mentre ora parliamo di sistemi di dialogo che sono molti più complessi dei sistemi di machine translation, perchè devono ascoltare e rispondere in maneira consistente in base a ciò che si è ascoltato

Cos'è un dialogo?

è un'attività **collaborativa**, dove ci sono vari elementi che entrano in gioco quindi non solo l'aspetto linguistico, ma anche gli aspetti temporali (in che momento, in quanto tempo dico una cosa), spaziali,ecc..

Può essere tra

- **uomo-uomo** chi ascolta riceve il messaggio, ragiona e risponde (quindi da noi umani avvengo le fasi invertite di pragmatica, semantica, sintassi e morfologia.
- **Uomo-macchina [ChatBot]** si usano *"cheat"* tipo: quando trovo la parola *"ama"*, allora rispondo *"ah, l'amore!"* (tipo Eliza)

Cosa caratterizza un dialogo?

▼ Turni

Capire quando tocca a te a parlare, ci basiamo quindi su dei segnali

- silenzio
- segni di esitazione
- intonazione
- body language

Sono fenomeni molto complessi, che solo un umano per ora può veramente capire

▼ Atti del dialogo

i turni si organizzano in unità basiche di comunicazione chiamate **atti di dialogo,** che hanno una natura totalmente diversa a seconda del contesto, gli atti possono essere di tipo

- Assertivo
- Imperativo
- Espressivo
- Dichiarativo

▼ Contesto conversazionale

gli atti del dialogo sono soggetti al **contesto conversazionale** (di cosa stiamo parlando), ci sono 3 aspetti/situazioni che caratterizzano un contesto, cioè quando abbiamo

- **frasi che non sono frasi**

  > **Esempio** alla domanda *“Quando parrti?”* la risposta *“Domani”* non è una frase ma il contesto mi dice che la frase implicita sarebbe “I*o parto domani*”

- **frasi che implicano**

> **Esempio** alla domanda *“Domani vieni alla cena?”* la risposta *“Domani sarò molto indaffarato”* non ha risposto alla domanda, ma la risposta implica che è come se avesse risposto di no

- **frasi anaforiche**

> **Esempio** *“Lui ha freddo?”* devo capire che mi sto riferendo ad una certa persona (maschio)

▼ Segnali ground

Questo viene a costruire il cosiddetto **common ground,** per capire quallo che è stato detto prima per capire come rispondere adesso o successivamente (per tenere il filo del discorso concordato da entrambe le parti)

ChatBots

Le quatto proprietà prima indicate, sono difcili da mantenere, nei **ChatBot** si possono non usare tutte insieme.

Abbiamo diverse tipologie di ChatBots

▼ Rule-based

▼ ELIZA

Lavora su regole basate su regex dove fa a riconoscere dei pattern.

Se quello che dice l’utente matcha con una di queste regole allora la regola in questione scatta, e usa le parole che hanno matchato restituendo una frase di senso compiuto.

- Se matcha più di una regola, allora ci si basa su un ranking delle keyword dove per ogni regola prediligo quella più specifica.
- Inoltre ho un meccanismo di memoria dove se nella prossima frase non matcho nulla, allora uso una delle frasi (nel ranking) che prima era una potenziale candidata ma non l’avevo usata

▼ PARRY

Fa la stessa cosa di ELIZA, però ha un **mental model** quindi riesce a rispondere creando ua forma di umore sulla risposta (incazzato, felice, ecc..). Per impersonare per esempio un Professore che ti deve interrogare

▼ Corpus-based

che recuperano info da un corpus o sul Web, e sono di tipo

- Information Retrival
- Neural network encoder-decoder

Caratteriscihe di un ChatBot

Pro

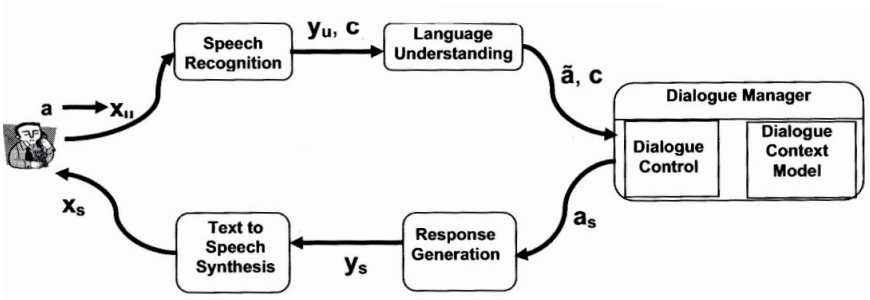
utili per

- contesti di intrattenimento
- contesti commerciali (assistenza clienti)

Contro

- un chatbot **non capisce realmente,** dopo la conversazione non c'è un report che mi dice l'argomento della conversazione
- **non scalano**, rimangono fermi nel contesto per cui sono stati creati

Architettura di un ChatBot



- dall'utente parte la frase/messaggio
- viene analizzato il messaggio tramite il modulo di **speech recognition** (qui basta un piccolo errore per sfanculare poi tutto il resto)
- viene quindi interpretato dal modulo di **NLU** quello che è stato capito al passo precedente
- poi il messaggio viene passato ad un sistema di dialogo (**Dialog Manager**)
- il DM genera con il modulo **NLG** una risposta (la sua interpretazione)
- la risposta viene data all'utente tramite il modulo di **text to speech**

Analizziamo i moduli

▼ Speech recogniton

oggigiorno è sempre più performante per via di

- grande vastitià di vocaboli aggiunti
- **indipendenza** ovvero che ora capisce il testo a prescindere da chi arriva la voce
- microfoni più precisi e performanti

▼ NLU

Natural language understanding, ci sono due approcci

- classico quello di **Montague**, sintassi + semantica + pragmatica
- moderno basato su GUS semantics una struttura chiave valore con dati tipati di tipo (Slot,Type) + un Ontologia di dominio oppure utilizzo di sistemi statistici + ML su dialoghi già annotati

Slot	Type
ORIGIN CITY	city
DESTINATION CITY	city
DEPARTURE TIME	time
DEPARTURE DATE	date
ARRIVAL TIME	time
ARRIVAL DATE	date

▼ Dialog Manager

lavora due concetti importanti

- **contextual model** quali informazioni che sono state dette sono importanti per ragiungere il task
- **modello di decisione**, si chiede come procedere. Rispondere subito, oppure salvare quello che è stato detto e aspettare altro input, ecc..

Nei DM abbiamo due approcci

- **di traduzione,** dove traduce da un linguaggio umano ad un linguaggio macchina (promt dei comandi)
- **di proattività,** che si comporta come un'agente che oltre a risponere ad un comando può prendere iniziativa.

Strutuuturalmente sono divisi in un

- **dialog control,** l'algoritmo che governa il dialogo che può essere basato su
  - grafi anche se in un contesto complesso è complicato perchè dovrò avere tanti stati (nodi) allocati
  - frame vuol dire che se devo comprare un volo devo riempire il frame con i suoi campi, per cui posso fare domande specifiche all'utente se mancano

dei campi nel frame. Il **problema** è che nei frame non ho ricorsività, non resco a catturare elementi ricorsivi, per cui si è introdotto l'**information state** dove aggiungiamo al frame altri elementi (pila, stack) per rendolo più similmente ricorsivo.

    - modelli statistici

- **dialog contex model**, che gonverna è la struttura dati, mantiene il contesto del dialogo, cronologia (il common ground praticamente)

`La valutazione di un DM`

si basa sulla **Trindi Ticklist**, che sono domande a cui posso fare al mio DM per fare una prima valutazione su cosa sia riuscito a fare e cosa non è riuscito

> **Esempio** L'intepretazione della frase dell'utente era corretta?

▼ NLG

a questo punto useremo un sistema di generazione del linguaggio come abbiamo visto nelle lezioni precedenti

▼ Text to Speech

dove adesso possiamo anche settare l'enfasi, gli accenti delle lingue, ecc..