

Tecnologie del Linguaggio Naturale

Parte 1 – prof. Mazzei

Mazzone Giuseppe

Sgaramella Davide

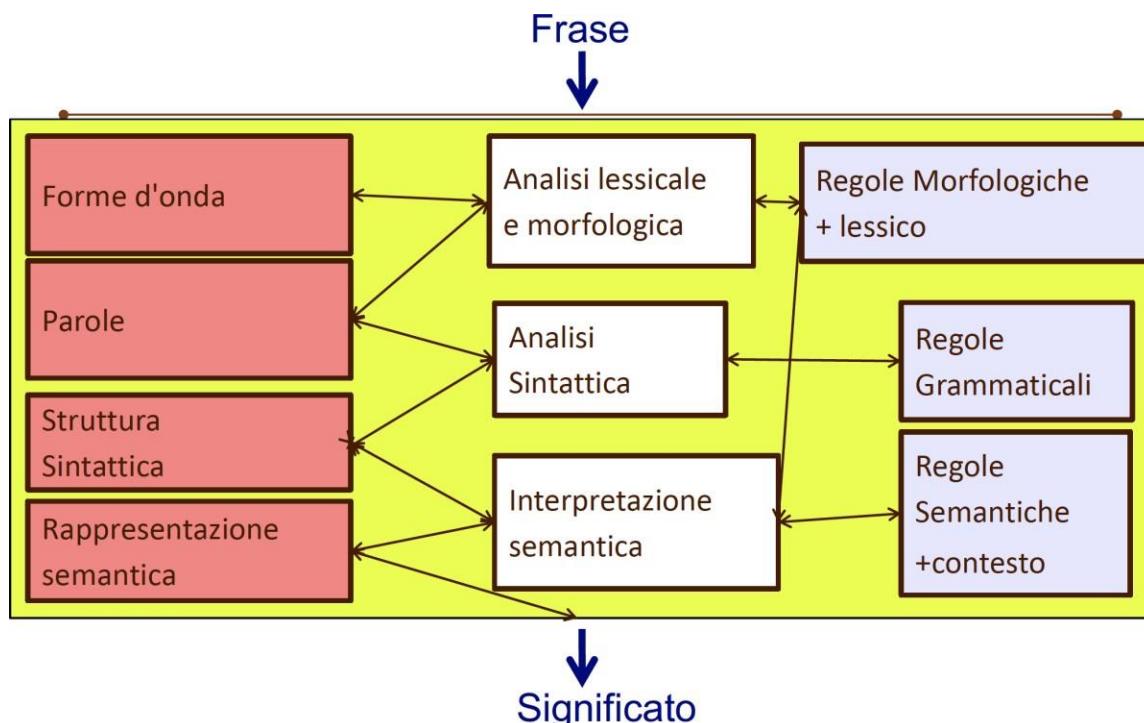
Appunti aggiornati a Ottobre 2020

LEZIONE 2

In questa lezione il concetto fondamentale è il livello linguistico.

Questo ci porta dalla superficie del linguaggio ad un livello più profondo proprio perché sotto una frase ci sono molti livelli fondamentali. Il seguente diagramma rappresenta una possibile architettura funzionale e modulare di un sistema computazionale che porta da una frase al suo significato, i 3 colori stanno a rappresentare 3 diversi moduli. **Analisi lessicale e morfologia**, **analisi sintattica**, e l'**interpretazione semantica**.

Questa è solo una rappresentazione dei moduli che studieremo di più nel nostro corso. Le frecce indicano come si utilizzano tra di loro per calcolare l'output finale. Ogni diverso modulo utilizza una diversa fonte di informazione.



Ad esempio, per fare **analisi sintattica**, abbiamo bisogno di **parole contenute nella frase e regole grammaticali**. L'**interpretazione semantica** invece utilizza **la struttura sintattica e le regole semantiche più il contesto** per creare la rappresentazione semantica della mia frase.

In che modo questi moduli comunicano con le basi di conoscenza ??

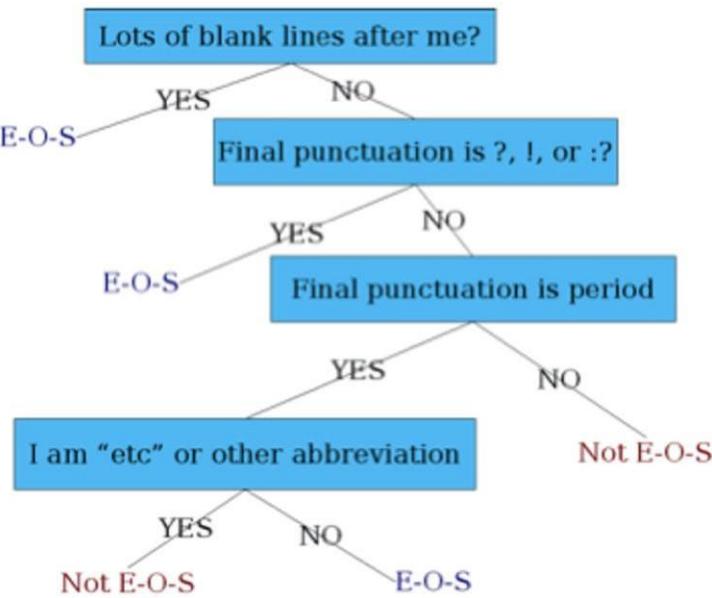
Prima di tutto, chiariamo un grande errore comune, dobbiamo suddividere i sistemi a regole e i sistemi statistici. Molte volte sono entrambe le cose, quindi non sono proprio separati, però a livello culturale, i due sistemi sono storicamente suddivisi.

Un difficile task è quello di capire quando finisce una frase.

Tale task di **Sentence Segmentation** risulta semplice se ad esempio se incontriamo un ! o ?. La situazione cambia quando abbiamo il . perché non sempre indica che la frase finisce, può indicare anche una abbreviazione (dott. Prof. 4.5).

Quindi il nostro task si può definire come se fosse un classificatore che ci dica "si" o "no" se il carattere è un fine frase (EoS) o meno. Una possibilità è fare un'introspezione della lingua italiana e provare a scrivere a mano delle regole in un linguaggio formale che ci permettono di definire dei pattern in grado di scrivere espressioni regolari che ci portano all'obiettivo (si pensi agli automi finiti).

Un'alernativa è utilizzare un corpus di frasi indicando per ognuna quale è il carattere che permette di indicare il fine frase (machine learning).



Bisogna poi scegliere quale tecnica di machine learning utilizzare nell'ultimo caso proposto, una tecnica molto utilizzata è quella degli alberi di decisione dove ad ogni passo sul carattere corrente si chiede se è fine frase o no.

Come è possibile vedere in questo esempio abbiamo un albero di decisione che su nuovi esempi si chiede, attraverso delle domande opportunamente create, se siamo a fine frase o no. Questo non è altro che un albero di decisione basato su delle caratteristiche/features. Tali features possono essere di diverso tipo, potrebbero anche essere più complesse.

In ogni caso le features ci permettono di capire se ci si trova di fronte ad una EOS oppure no.

Ragionando su cosa è un albero di decisione, possiamo fare una riflessione su cosa esso rappresenta, ovvero un insieme di regole incapsulate che vengono apprese in maniera autonoma e si basano sul valore di features linguistiche che vengono scelte dal programmatore.

Il concetto di **features** è comune a diverse tecniche di ML, non solo agli alberi decisionali, quindi rimane un punto cruciale per molti modelli. Nel caso della linguistica computazionale risultano fondamentali quindi le features linguistiche.

Per alcuni task della linguistica computazionale posso dimenticarmi delle features e creare un sistema input-output quindi permetto al sistema di capire/costruire autonomamente le features stesse. Si potrebbe pensare di non aver più bisogno della linguistica dato che le reti fanno tutto autonomamente, ma in realtà un punto fondamentale è proprio l'architettura della rete che, se ben modellata, permette di modellare le features linguistiche al meglio. Manning ci dice che non solo le performance sono importanti ma anche che la forma dell'architettura sia linguisticamente motivata.

LIVELLO MORFOLOGICO E L'ANALISI LESSICALE

Si basa sul concetto di parola, elemento costituito da elementi atomi (morfemi), rappresenta il primo livello della pipeline. **Parola**: sequenza di caratteri delimitata da spazi o punteggiatura. Una definizione così semplice è abbastanza generale. Ad esempio, in certi casi una parola ne esplica diverse e in altri casi si verifica il viceversa:

- "passamela": che sta per "passa a me essa". Abbiamo una parola che ne esplica quattro.
- "by the way": viceversa tre parole che nel loro insieme formano un unico concetto logico.

Inoltre, insiemi di parole mostrano un significato unitario dal punto di vista della sintassi lessicale, ma ci sono altre parole il cui significato non è composito (Es: "di corsa"). Questo problema in altre lingue può essere anche più evidente.

Inoltre, abbiamo un'ambiguità dovuta alla presenza di suffissi, ad esempio:

- Capitano (forma non declinabile)
- Capitan+o (nome o aggettivo o forma del verbo capitanare)
- Capit+ano (forma del verbo capitare)

La parola capitano può indicare "Tutti sono i capitani" oppure "io capitano questa nave" oppure "le cose capitano", assume 3 significati diversi.

Per questi motivi si usa un **analizzatore morfologico**: è un algoritmo che, presa in input una parola, restituisce in output tutte le possibili forme morfologiche (morfemi) di tale parola. Si basa su un dizionario (insieme di lemmi/morfemi, "capitare" per "capitano") e una serie di regole che indicano come possono essere combinati questi morfemi. Le regole possono essere scritte a mano o statistiche, ad esempio ottenute da un classificatore. Può funzionare anche su parole nuove (Es: twittare), basandosi sulle regole morfologiche.

L'analizzatore morfologico deve tenere conto di:

- **Forme composte**, cioè quelle parole formate da una parola contenuto più una (o più) parole funzione.
 - STAMPAMELO: STAMP+A+ME+LO (radice verbale + suffisso verbale + forme pronominali).
- **Forme multiple** in cui le diverse componenti sono nel dizionario, ma la semantica non è composito:
 - Più o meno: composto da tre parole, ma il significato non è composito e può essere considerato come un'unica parola. Viste separate avrebbero un significato diverso.
 - Prendere un abbaglio.

La maggior parte di questi sistemi è basato sugli automi finiti e sulle espressioni regolari. Alcuni analizzatori sono: Foma, Morph-it, TULE.

L'analizzatore morfologico si basa su due operazioni che sono la lemmatizzazione e lo stemming:

- **Lematizzazione**: riduzione dei vocaboli ad una forma standard, ad esempio si trasformano tutti i verbi all'infinito: Amaron → Amare
- **Stemming**: si riduce una parola alla propria radice. È utilizzato in contesti di information extraction. Amare → Amar

L'analizzatore morfologico è seguito da un **disambiguatore morfologico**, che prende in input le varie possibilità prodotte dall'analizzatore e un contesto e fornisce in output la giusta analisi morfologica sulla base del contesto. In pratica contestualizza gli output dell'analizzatore morfologico.



Part of Speech (PoS)

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 potrebbero alternarsi ad essi: se ad una frase cambio una parola, dal punto di vista semantico potrebbe non essere corretta, ma da quello grammaticale si. Es: Il cane corre – La sedia corre.
- **Sintagmatico**: relazioni che rendono la frase grammaticale e legano le sequenze di parole. in "paolo ama francesca" parole ha un legato sintagmatico con

Le parti del discorso possono essere aperte o chiuse:

- **Aperte (open)**: 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 (closed)**: le aggiunte alle categorie sono molto rare e cambiano nell'arco dei secoli. In questa categoria rientrano le **function word**, quindi preposizioni, articoli, pronomi, congiunzioni, interiezioni. Sono dell'ordine delle centinaia.

Tullio De Mauro ha fatto una distinzione tra le parole:

- **Parole ad alto uso**: sono quelle presenti nei testi. Circa mille.
- **Parole ad alta disponibilità**: parole che tutti conoscono, ma che sono infrequenti nei testi

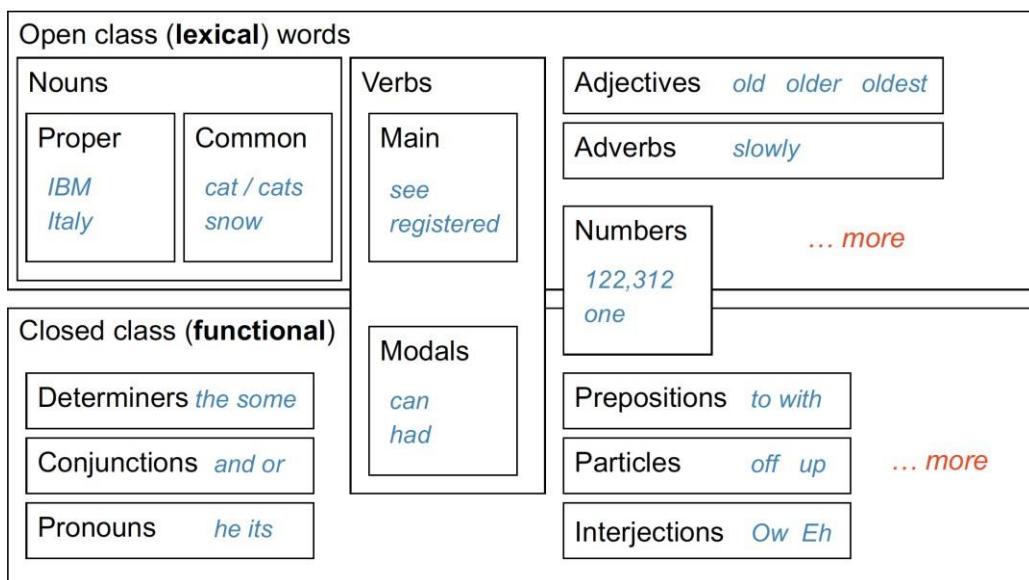
Se consideriamo i **sub-languages**, cioè linguaggi specifici della materia, ad esempio informatica, medicina ecc. arriviamo all'ordine delle centinaia di migliaia → Zellings.

Le classi grammaticali più importanti sono nome e il verbo perché dal punto di vista semantico indicano gli oggetti e l'azione:

- **Nomi:** indicano persone, oggetti, luoghi.
- **Verbi:** indicano eventi, azioni, processi. Abbiamo molte forme morfologiche, come ad esempio il modo, il tempo e il numero e tante categorie: ausiliari, modali, copula, ecc.

Abbiamo gli aggettivi che descrivono delle proprietà e gli avverbi che invece modificano qualcosa, spesso verbi, ma anche altri avverbi o intere frasi.

Di seguito una schematizzazione di quanto detto in precedenza. I verbi possono essere sia open che closed, in quanto possiamo inventare nuovi verbi (googlare, twittare), ma non possiamo inventarci nuovi modali; così come i numeri, in quanto i numerali sono in numero finito, ma possiamo inventare sempre un numero più grande.



Esistono diverse tecniche di PoS come Penn PoS, ISDT PoS, TUT PoS e il Google universal PoS che contiene 12 diversi PoS in grado di funzionare contemporaneamente su diverse lingue.

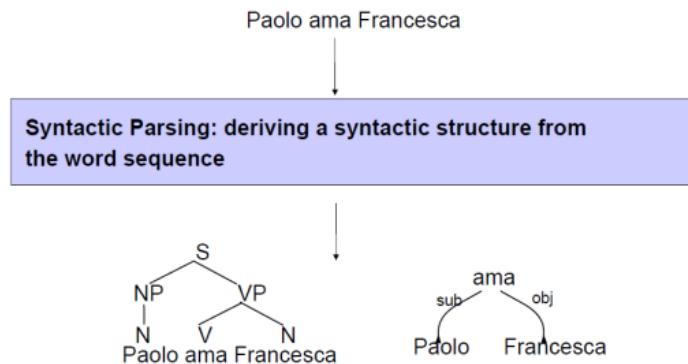
PoS tagging

Il PoS tagging consiste nell'assegnare una parola ad una delle possibili categorie.

Penn tree bank: è corpus di frasi annotate morfologicamente e sintatticamente. È composto da 40 PoS. Google propone 12 PoS, e in particolare vengono aggiunti NUM (numerals), che individuano i numeri, PRT (particles), “.” Per i simboli di punteggiatura e X per abbreviazioni, link, ecc.

LIVELLO SINTATTICO

Questo livello prende come input l'output del livello morfologico. Il task è l'**analisi sintattica** che fornisce una struttura dati in grado di codificare le **relazioni sintattiche**. Queste relazioni possono essere rappresentate in diversi modi. L'analisi sintattica prende in input una sequenza di parole e restituisce una struttura sintattica, che può essere una **struttura a costituenti (sx)** o una **struttura a dipendenze (dx)**.



Struttura a costituenti: orizzontale, rappresentano esplicitamente la relazione di gruppo tra le parole. Questo permette che alcune parole si comportino come un'unità (VP), si hanno delle strutture annidate ad albero. **L'ordine delle foglie ci dà l'ordine delle parole nella frase.**

Utilizzato negli Stati Uniti. Le parole sono organizzate in costituenti annidati. Un costituente si comporta come un'unità che può apparire in differenti parti della frase. In particolare, un **costituente è un gruppo di parole contigue che si comportano come un'unità e che hanno delle proprietà sintattiche.**

Per capire se un elemento è un costituente o meno Chomsky propone un test: se un elemento è sostituibile con un altro elemento della stessa categoria e se a quell'elemento possiamo concatenarne un altro della stessa categoria senza invalidare la frase, allora l'elemento è un costituente.

Gli alberi tendono ad essere **right-branched**, tipica della sintassi delle lingue europee. **VP: ama Francesca** si comporta un gruppo che può essere chiamato sentence. "Ama Francesca" è ordinato, le parole non possono essere scambiate. Inoltre "ama Francesca" è un costituente perché possiamo sostituirlo con un altro VP e la frase continua ad essere corretta, ad esempio: "ama Chiara".

Sarà possibile sostituire dei gruppi con delle strutture equivalenti, creando delle frasi grammaticalmente giuste. Ma ci chiediamo, il concetto di costituente è effettivamente utilizzato dal nostro cervello? Alcuni studi dimostrano di sì, ma sono molto discussi. Alcuni studi dimostrano che le persone creano frasi in base alle strutture che conoscono. **Le grammatiche costituenti sono anche utili per capire ad esempio chi fa cosa in una frase.**

Struttura a dipendenze: verticale, si rappresentano le relazioni di dipendenza tra le parole, quindi l'ordine delle foglie non rispecchia l'ordine delle parole nella frase. **Il verbo "comanda", il soggetto viene "comandato". È utilizzato nell'est-Europa.** Il verbo prende il nome di head ed è la parola dominante; il soggetto, che è la parola dominata, è il dependent.

Esempio: Paolo ama Francesca → ama = head – Paolo/Francesca = dependent.

La relazione tra head e dependent può essere di due tipi:

- **Argomento:** quando la parola dipendente è un argomento dell'head. Ad esempio, in "ama Francesca", Francesca è argomento dell'head ama. È quell'elemento sintattico indispensabile per considerare una frase sintatticamente corretta.
- **Modificatore:** quando la dipendente va a modificare in qualche modo l'head, ad esempio: "Paolo corre velocemente", velocemente ci dà informazioni in più su corre. Elemento opzionale.

L'ipotesi fondamentale linguistica computazionale quando si parla di sintassi a costituenti è la seguente: la generazione dell'albero è rappresentata con una CFG e i simboli non terminali della CFG sono proprio i costituenti.

LIVELLO SEMANTICO

L'obiettivo è capire quale sia il significato di una frase. Si trasforma la frase in un qualche modo, attraverso una **interpretazione semantica**, tale da poter utilizzare il significato contenuto nella frase per ragionare. L'ipotesi di Frede (Chomsky) dice che la semantica della frase si costruisce a partire dalla semantica delle parole (foglie) salendo nell'albero, fino ad arrivare alla radice, che è la semantica della frase.

Tale interpretazione semantica può essere divisa in due fasi:

1. **semantica lessicale**, trova il significato delle parole;
2. **semantica formale**, compone il significato trovato attraverso l'uso della struttura sintattica.

Fase 1 – Semantica Lessicale

Definito lessema un concetto come forma-significato, cioè un elemento del lessico, allora fare semantica lessicale vuol dire associare il giusto significato ad ogni parola della frase; per **forma** si intende proprio la forma ortografica e fonologica mentre il **significato** è il senso ovvero l'unità elementare di significato nascosta nella forma ontologica.

Nei **vocabolari** è esplicitata per ogni parola il significato, solo che in alcuni vocabolari ci sono degli errori che il sistema non riuscirebbe a sopportare, ad esempio se si prende la parola rosso il vocabolario ci dirà che è data dal colore del sangue e la definizione di sangue invece come liquido rosso. Si crea così un problema di circolarità, infatti per la definizione di rosso mi ritrovo a valutare il sangue e per il sangue valuto nuovamente il rosso. Non si riescono a definire i concetti di base primitivi sul quale poter definire dei concetti più complessi, infatti la teoria del **vocabolario** non è ammissibile per definire la semantica lessicale in maniera computazionalmente accettabile.

Costruire i lessemi in maniera automatica è complicato, in quanto abbiamo un problema di ricorsione circolare.

Una possibile alternativa è una tecnica che si basa sul concetto di **relazione tra lessemi**, ovvero si costruiscono delle relazioni di tipo lessicale che raggruppano tra di loro in maniera equivalente le parole. I lessici computazionali funzionano così, quindi sulla sinistra ho il lessema da descrivere mentre sulla destra trovo un gruppo di parole che mi rappresentano un concetto individuale individuabile da quei 4 tipi di relazione. Si parla di:

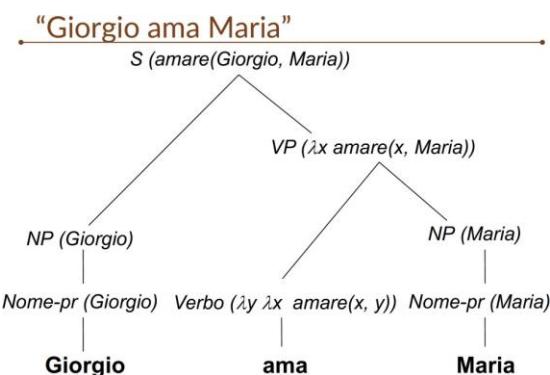
- **Omonimia**: intendiamo due lessemi che hanno la stessa forma ortografica, ma due significati diversi in base alla frase e al contesto. Es: pesca.
- **Polisemia**: stesso lessema, ma due sensi diversi che sono in qualche relazione tra di loro. In questo caso si ha una etimologia comune dei sensi che hanno una qualche relazione comune. Es: banca monetaria. banca del sangue.
- **Sinonimia**: permette di indicare lessemi con forma ortografica diversa ma che hanno lo stesso significato o un significato simile. A volte tale sinonimia è solo locale, ovvero anche se due parole sono sinonimi non è detto che sostituendo la prima parola con il sinonimo il significato della frase sia lo stesso. Es: How big is that plane? How large is that plane?
- **Iponimia**: lega tra di loro due lessemi di cui uno denota una sottoclasse dell'altro. Es: automobile è un **iponimo** di veicolo. Al contrario si può dire che veicolo è **iperonimo** di automobile.

Fase 2 – Semantica formale (composizionale)

Supponendo di essere riusciti a codificare la semantica lessicale nel concetto di lessema, allora ci impegniamo a capire quale è la seconda fase ovvero quella di semantica formale, cioè riuscire a ragionare e rispondere magari a domande che non per forza utilizzano la stessa forma della frase originale. Ad esempio, sapendo che “Giovanni ha acquistato un’auto” deve essere possibile rispondere a domande del tipo “Giovanni ha comprato un veicolo?”. Come è possibile notare nell’esempio, si usano dei termini leggermente diversi. Si necessita di un ragionamento che non si basi sulla singola parola ma sul significato di tutte le parole comprese in una frase. Per fare ciò posso basarmi ad esempio ai linguaggi formali come X+Y.

La semantica di un sintagma è funzione della semantica dei sintagmi componenti; non dipende da altri sintagmi esterni al sintagma stesso. Quindi con la semantica computazionale potremmo dire che un numero finito di regole potrebbe controllare un numero infinito di soluzioni poiché una volta definita la semantica della composizione e una volta definiti quali sono i costituenti possibili allora si potrebbe calcolare la semantica di qualsiasi frase di qualsiasi lunghezza.

Nel 1974 ci fu una proposta che intendeva capire quali rappresentazioni semantiche si associano con quali sintagmi. Quindi è stata fatta un’analisi sistematica della sintassi a costituenti per cercare di capire come passare dalla frase “Giorgio ama Maria” alla formula logica del primo ordine ad esempio amare(Giorgio, Maria) ovvero la forma di predicato che esprime la relazione tra i due parametri.



Attraverso questa rappresentazione è possibile ragionare in 3 modi:

- **Deduzione:** se le premesse sono vere, le conclusioni sono vere. È sempre vero. Se tutti gli uomini amano Maria e Giorgio è un uomo, allora anche Giorgio ama Maria.
 $\forall X \text{ uomo}(X) \rightarrow \text{amare}(X, \text{Maria}) \quad \text{uomo}(\text{Giorgio}) \rightarrow \text{amare}(\text{Giorgio}, \text{Maria})$
- **Induzione:** cerchiamo di ricavare delle informazioni a partire da ciò che vediamo. Non sempre vero. Se tutti gli uomini amano Maria e Giorgio ama Maria, allora Giorgio è un uomo. Non è detto che sia vero, Giorgio potrebbe essere anche un gatto!
 $\forall X \text{ uomo}(X) \rightarrow \text{amare}(X, \text{Maria}) \quad \text{amare}(\text{Giorgio}, \text{Maria}) \rightarrow \text{uomo}(\text{Giorgio})$
- **Abduzione:** partiamo da una premessa certa e una incerta e proviamo a ricavare una conclusione probabili. Non sempre vero. Sapendo che esiste una donna che Giorgio ama. Se giorno ama Maria, allora Maria è una donna.
 $\exists X \text{ amare}(\text{Giorgio}, X) \quad \text{amare}(\text{Giorgio}, \text{Maria}) \rightarrow \text{donna}(X)$

La **semantica distribuzionale** indica che il significato di una parola è legato alla distribuzione di parole intorno a sé. Come nella poesia del Lonfo che ci permette di capire il significato senza che effettivamente esista.

Il concetto di semantica lessicale può portarci da una rappresentazione simbolica delle parole ad una rappresentazione numerica ovvero un embedding, dove vado a considerare il contesto delle parole in base ai documenti in cui queste parole compaiono e magari quante volte compare e vicino a quali parole compare, in questo modo è possibile costruire una rappresentazione statistica delle parole sottoforma di vettore numerico. Il problema è la dimensione di questi vettori.

Si distinguono i vettori classici che sono dei vettori lunghi, molto ampi e molto sparsi in cui la rappresentazione è più una one hot representation poiché si tratta di vettori lunghi quanto il testo e con solo una casella avvalorata in direzione di ogni parola. Una alternativa più moderna consiste in vettori di dimensione più piccola, molto più densi, dove la maggior parte degli elementi non sono nulli. Il concetto di **word embedding** fa più riferimento a quest'ultima tipologia di vettore ed è nata in vicinanza con le reti neurali, poiché di solito vengono specializzati da queste ultime. Questi vettori si sono dimostrati molto versatili e facili da usare; in pratica si parte da un vettore casuale e in maniera iterativa si shifta un embedding in modo tale che sia più simile agli embedding delle parole vicine e meno simile agli embedding delle parole lontane. In questo modo otteniamo vettori di vocaboli correlati tra loro. Il significato può essere rappresentato in forma vettoriale:

$$\underline{\text{vector}(\text{"king"}) - \text{vector}(\text{"man"}) + \text{vector}(\text{"woman"}) \approx \text{vector}(\text{"queen"})}$$

Es: in pratica se consideriamo il vettore relativo alla parola “king”, gli sottraiamo il vettore “man” e gli aggiungiamo il vettore “woman”, otteniamo un vettore simile a quello relativo alla parola “queen”.

Tuttavia, non si è ancora capito se questi word embedding possono essere utilizzati in una semantica compositiva per rappresentare una semantica che non sia semplicemente l’analoga tra parole, ma qualcosa di più complesso come la descrizione di un evento.

LIVELLO PRAGMATICO

Intende l’integrazione dell’informazione che viene dal canale linguistico con altre informazioni che non vengono dal canale linguistico. Come ad esempio la frase “Oggi sono ad Alessandria” deve essere intesa come la persona che parla si trova nella città di Alessandria. Ovvero il linguistic meaning di una frase che può essere legato a diversi aspetti. Come ad esempio l’utilizzo delle Anfore ovvero sintagmi che si riferiscono a oggetti precedentemente menzionati. Non affronteremo tale livello in questo corso.

Analisi morfologica

PoS TAGGING

Tale task fa riferimento al livello morfologico e sarà l'unico in questo ambito a essere visto nello specifico.

| | | | | |
|-------------------|-----------|-------------|---------|------------|
| <u>Input:</u> | Plays | well | with | others |
| <u>Ambiguity:</u> | NNS/VBZ | UH/JJ/NN/RB | IN | NNS |
| <u>Output:</u> | Plays/VBZ | well/RB | with/IN | others/NNS |

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

Il **Part of Speech Tagging** è una tecnica che permette di rimuovere le ambiguità delle parole e restituisce come output, per ogni parola, il suo ruolo disambiguato.

Si tratta di un problema di Sequence tagging: data una frase bisogna capire, per ogni parola che la compone, il ruolo (PoS Tag). Tale tecnica può avere diverse applicazioni: per la lingua inglese può indicare come pronunciare una parola oppure può farci capire se è un nome o un aggettivo ecc. 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, o anche dal livello semantico che può utilizzarlo per disambiguazione semantica delle parole.

Il PoS tagging è complicato a causa dell'ambiguità del linguaggio, poiché abbiamo diversi tag a seconda del contesto per ciascuna parola della frase. Utilità del PoS tagging:

- Il PoS serve a diversi task pratici, in quanto la conoscenza sui PoS associati ad ogni componente della frase permette ad esempio di determinare quale sia la giusta pronuncia (to lead, lead).
- Conoscendo la sequenza di PoS è possibile fare **chunk parsing**, ossia utilizzare automi finiti per riconoscere i gruppi che stanno attorno ai nomi.
- Nel nostro caso i tag rappresentano l'input del livello di parsing sintattico.
- Può essere utilizzato anche come feature aggiuntiva per la sentiment analysis.

Un modo per misurare l'accuratezza del sistema in questo task è quello di verificare la Tag Accuracy ovvero la accuratezza nel tag di ogni parola, in genere si aggira intorno al 97%. Non basta però tale numero, ci si basa infatti sulle Baseline ovvero dei sistemi di confronto molto semplici (come ad esempio in questo caso l'utilizzo di un algoritmo molto semplice, che tagga ogni parola alla sua occorrenza con il tag più frequente per quella parola recuperato da un corpus precedentemente annotato). Possono esserci casi in cui delle parole non compaiono nel corpus, si decide quindi di taggare le parole sconosciute come nomi poiché sono quelli che più spesso vengono creati.

Il concetto di Baseline implica sempre un confronto, quindi avremo sempre bisogno di un sistema a cui paragonarci. In tale task sembra semplice l'utilizzo delle baseline ottenendo infatti una precisione del 90%.

In questa tabella vengono riportate le ambiguità sia nel caso dei tag set originali, con un 10%, e sia nel nuovo tag set con un 20% di ambiguità circa. Esistono diversi tipi di ambiguità.

| Tag | Description | Example | Tag | Description | Example | | |
|-------|-----------------------|------------------------|------|-----------------------|--------------------|--|--|
| CC | coordin. conjunction | <i>and, but, or</i> | SYM | symbol | <i>+, %, &</i> | | |
| CD | cardinal number | <i>one, two, three</i> | TO | "to" | <i>to</i> | | |
| DT | determiner | <i>a, the</i> | UH | interjection | <i>ah, oops</i> | | |
| EX | existential 'there' | <i>there</i> | VB | verb, base form | <i>eat</i> | | |
| FW | foreign word | <i>mea culpa</i> | VBD | verb, past tense | <i>ate</i> | | |
| IN | preposition/sub-conj | <i>of, in, by</i> | VBG | verb, gerund | <i>eating</i> | | |
| JJ | adjective | <i>yellow</i> | VBN | verb, past participle | <i>eaten</i> | | |
| JJR | adj., comparative | <i>bigger</i> | VBP | verb, non-3sg pres | <i>eat</i> | | |
| JJS | adj., superlative | <i>wildest</i> | VBZ | verb, 3sg pres | <i>eats</i> | | |
| LS | list item marker | <i>1, 2, One</i> | WDT | wh-determiner | <i>which, that</i> | | |
| MD | modal | <i>can, should</i> | WP | wh-pronoun | <i>what, who</i> | | |
| NN | noun, sing. or mass | <i>llama</i> | WP\$ | possessive wh- | <i>whose</i> | | |
| NNS | noun, plural | <i>llamas</i> | WRB | wh-adverb | <i>how, where</i> | | |
| NNP | proper noun, singular | <i>IBM</i> | \$ | dollar sign | <i>\$</i> | | |
| NNPS | proper noun, plural | <i>Carolinas</i> | # | pound sign | <i>#</i> | | |
| PDT | predeterminer | <i>all, both</i> | " | left quote | <i>' or "</i> | | |
| POS | possessive ending | <i>'s</i> | " | right quote | <i>' or "</i> | | |
| PRP | personal pronoun | <i>I, you, he</i> | (| left parenthesis | <i>[, {, <</i> | | |
| PRP\$ | possessive pronoun | <i>your, one's</i> |) | right parenthesis | <i>], }, ></i> | | |
| RB | adverb | <i>quickly, never</i> | , | comma | <i>,</i> | | |
| RBR | adverb, comparative | <i>faster</i> | . | sentence-final punc | <i>! ?</i> | | |
| RBS | adverb, superlative | <i>fastest</i> | : | mid-sentence punc | <i>: ; ... --</i> | | |
| RP | particle | <i>up, off</i> | | | | | |

| | 87-tag Original Brown | 45-tag Treebank Brown |
|-----------------------------|--------------------------|--|
| Unambiguous (1 tag) | 44,019 | 38,857 |
| Ambiguous (2–7 tags) | 5,490 | 8844 |
| Details: | | |
| 2 tags | 4,967 | 6,731 |
| 3 tags | 411 | 1621 |
| 4 tags | 91 | 357 |
| 5 tags | 17 | 90 |
| 6 tags | 2 (<i>well, beat</i>) | 32 |
| 7 tags | 2 (<i>still, down</i>) | 6 (<i>well, set, round, open, fit, down</i>) |
| | | |
| | | 4 ('s, half, back, a) |
| | | 3 (<i>that, more, in</i>) |

Questo ci indica che il PoS tagging non è un problema banale. Le parole ambigue infatti compaiono spesso tanto che il 40 % dei word token sono ambigi. Una delle parole più ambigue in inglese ad esempio è that.

Ci sono diversi algoritmi per effettuare PoS tagging, questi possono essere:

- **Basati su regole:**
 - ENGTWOL (ENGLISH TWO Level analysis)
- **Stocastici:**
 - HMM (Hidden Markov Model): utilizzato nell'analisi delle sequenze
 - MEMM (Maximum Entropy Markov Model)

RULE-BASED TAGGER ENGTWOL (ENGLISH TWO Level analysis)

Abbiamo in input una frase, quindi una sequenza di parole; l'output è una sequenza di tag.

L'idea di base è la seguente:

- **Step 1:** utilizzo di un analizzatore morfologico, per ogni parola presente in una frase restituisce il PoS associato in maniera decontestualizzata.
- **Step 2:** Fase di Remove, elimina tutti quei tag che non soddisfano determinati criteri: rimuove l'ambiguità per una parola in base a regole (ad esempio se la parola precedente non è un verbo e la successiva è un aggettivo allora possiamo eliminare tutti gli avverbi tra i tag possibili).

Entrambe queste fasi utilizzano delle regole che possono essere apprese in maniera automatica oppure scritte manualmente.

Questo esempio di lessico per l'inglese ci indica per ogni parola il proprio PoS e altre feature. Applicando tale tabella per la frase "Pavlov had shown that salivation" utilizzando un analizzatore morfologico ci permetterebbe di dire che pavlov viene classificato come nome o nome proprio, had può essere sia un ausiliario che un verbo autonomo, shown può essere il participio passato di show, that ci dice che può essere avverbio, pronome o altro, salivation invece può essere semplicemente un nome.

| Word | POS | Additional POS features |
|-----------|------|---------------------------------|
| smaller | ADJ | COMPARATIVE |
| entire | ADJ | ABSOLUTE ATTRIBUTIVE |
| fast | ADV | SUPERLATIVE |
| that | DET | CENTRAL DEMONSTRATIVE SG |
| all | DET | PREDETERMINER SG/PL QUANTIFIER |
| dog's | N | GENITIVE SG |
| furniture | N | NOMINATIVE SG NOINDEFDETERMINER |
| one-third | NUM | SG |
| she | PRON | PERSONAL FEMININE NOMINATIVE SG |
| show | V | IMPERATIVE VFIN |
| show | V | PRESENT -SG3 VFIN |
| show | N | NOMINATIVE SG |
| shown | PCP2 | SVOO SVO SV |
| occurred | PCP2 | SV |
| occurred | V | PAST VFIN SV |

Nella seconda fase invece, fissando delle regole come la seguente, i vincoli ci permettono di eliminare le ambiguità della frase. Infatti, ci dice che se la parola ha determinate regole allora possiamo eliminare dalle possibili analisi della parola that, i tag che non lo considerano come avverbio, al contrario elimino tutti i tag che lo considerano come avverbio.

```

IF (and
    (+1 A/ADV/QUANT)          /* if next word is adj, adverb, or quantifier */
    (+2 SENT-LIM)              /* and following which is a sentence boundary, */
    (NOT -1 SVOC/A) )         /* and the previous word is not a verb like
                               'consider' which allows adjs as object
                               complements */
THEN
    eliminate non-ADV tags
ELSE
    eliminate ADV

```

STOCHASTIC TAGGER

Il PoS tagging è basato sulla statistica, cioè sulle osservazioni che si possono fare per una parola basandosi su un corpus annotato con il PoS corretto fatto a mano o in maniera semi-automatica in cui siamo sicuri che non ci sono errori.

HIDDEN MARKOV MODEL

Si considera il problema del PoS tagging come un problema di **Sequence Labeling**. La sequenza di parole viene vista come una sequenza di osservazioni, ad ogni parola della sequenza si vuole associare un tag. Alla base di questa operazione c'è la probabilità: associeremo i tag alla sequenza di parole in base alle osservazioni che forniranno la probabilità più alta. La migliore sequenza è quella che ha la probabilità più alta. **Si ha quindi un problema di machine learning.** Seguendo lo standard del machine learning definiamo 3 fasi:

1. **Modelling:** viene fornito un modello formale del problema;
2. **Learning:** invece si cerca di capire come, dato un corpus, sia possibile settare i parametri in grado di generare il modello in grado di apprendere da un corpus;
3. **Decoding:** si cerca di capire qual è l'algoritmo che mi permette di applicare quello che ho imparato per poter recuperare la soluzione ottimale dato un certo input.

Modelling

L'obiettivo di questo modello, data la sequenza di n tag, $t_1 \dots t_n$, è trovare la massima probabilità dati gli osservabili $w_1 \dots w_n$, quindi **un tag per ogni parola che massimizzi la probabilità di un tag rispetto ad altri date tutte le parole in ingresso.** $\hat{t}_1^n = \operatorname{argmax}_{t_1^n} P(t_1^n | w_1^n)$

Per rendere operazionale tale calcolo si ricorre ad un modello bayesiana che ci permette di trasformare questa scrittura in una serie di probabilità più semplici da calcolare. In linea di principio si potrebbero calcolare tali probabilità direttamente da un corpus, però qualsiasi corpus a cui si applica questo approccio darà che i dati risultano troppo sparsi poiché considero la sequenza come un unico blocco e non considero i sottoinsiemi, ovvero le parole che lo compongono, come osservabili elementari, allora anche se il numero di frasi è alto non coprirà mai il vasto numero di frasi possibili.

Per rendere l'approccio più semplice, applico quindi la regola bayesiana.

$$P(x|y) = \frac{P(y|x)P(x)}{P(y)} \quad \hat{t}_1^n = \operatorname{argmax}_{t_1^n} P(w_1^n | t_1^n)P(t_1^n)$$

Ad un primo sguardo è possibile notare che si è aumentata la complessità in quanto adesso ci sono 2 probabilità da massimizzare, ma in realtà non è così perché queste due probabilità sono più semplici da calcolare attraverso delle approssimazioni.

Queste nuove probabilità sono:

- **a priori**: non dipende dagli osservabili, cioè dalle parole in input;
- **verosimiglianza**: dice quale può essere il valore del mio output in base al valore degli osservabili.

$$\begin{aligned}
 \hat{t}_1^n &= \underset{t_1^n}{\operatorname{argmax}} \overbrace{P(w_1^n | t_1^n)}^{\text{likelihood}} \overbrace{P(t_1^n)}^{\text{prior}} \\
 P(w_1^n | t_1^n) &\approx \prod_{i=1}^n P(w_i | t_i) \\
 P(t_1^n) &\approx \prod_{i=1}^n P(t_i | t_{i-1}) \\
 \hat{t}_1^n &= \underset{t_1^n}{\operatorname{argmax}} P(t_1^n | w_1^n) \approx \underset{t_1^n}{\operatorname{argmax}} \prod_{i=1}^n P(w_i | t_i) P(t_i | t_{i-1})
 \end{aligned}$$

Tali probabilità non sono calcolabili direttamente per tutti i possibili input, infatti si crea anche qui una matrice di sparsi in cui ci sono molti valori nulli. Si possono fare delle ipotesi di approssimazione: la prima lavora sulla verosimiglianza ovvero la probabilità $P(w^n | t^n)$ può essere approssimata come prodotto di n probabilità, ovvero delle singole probabilità per ogni i della $P(w_i | t_i)$. Questa è una approssimazione in quanto le parole dipendono dai tag precedenti.₁ Un'approssimazione molto simile si fa nel caso della probabilità a priori in cui la probabilità $P(t^n)$ può essere scritta come il prodotto di n probabilità $P(t_i | t_{i-1})$, quindi la probabilità di un tag considerando il tag precedente. Queste approssimazioni ci permettono di scrivere la sequenza più alta delle probabilità approssimate che permettono di modellare il problema.

Learning

Il problema è quello di trovare come, dato un corpus annotato con la PoS, trovo i parametri che caratterizzano l'equazione del modelling appena vista.

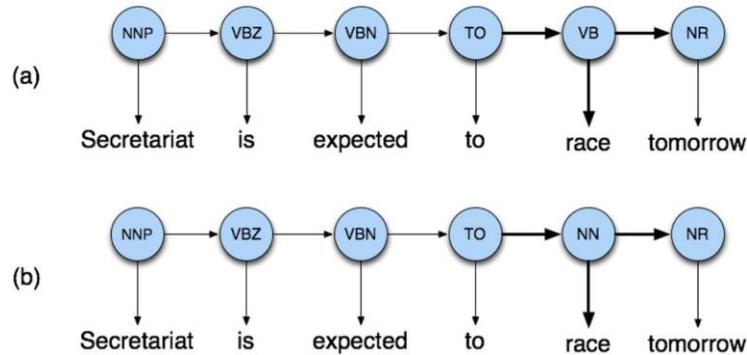
Le due probabilità vengono calcolate in maniera indipendente vedendo il corpus.

Quella a priori, **probabilità di transizione**, è possibile calcolarla semplicemente andando 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 $t-1$, calcolo così una frequenza relativa che è appunto una probabilità compresa tra 0 e 1.

Per la probabilità di verosimiglianza si vuole calcolare la **probabilità di emissione**, ovvero quanto un certo tag sia una certa parola; per farlo conto quante volte una certa parola è contata con quel tag e la divido per il numero di volte che compare quel tag nel corpus. Anche in questo caso il valore è compreso tra 0 e 1.

Decoding

Supponendo di aver finito di calcolare tutte le possibili combinazioni delle probabilità, è terminata la fase di learning ed è possibile passare alla fase di decoding. Rivedendo le probabilità calcolate, posso riguardarle come un sistema di transizioni in cui le mie variabili nascoste, che sono proprio il PoS, si comportano come gli stati di un sistema di markov e le mie parole si comportano come le immissioni di questi stati come ad esempio:

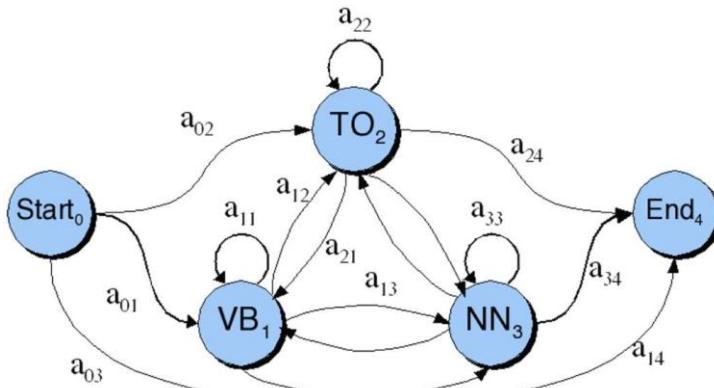


Nella seguente frase, il termine race può assumere 2 diversi significati ed è possibile notare come la probabilità di (a) rispetto alla probabilità di (b) cambia solo in base alle frecce in grassetto tra di loro, inerenti proprio a race, e ci permettono di calcolare quale delle due è più probabile.

Si sceglierà il valore più alto come valore corretto (**Esempio**: 0.5 Verbo. 0.1 Nome, risultato: verbo). Le frecce rappresentano le proprietà che riguardano un certo PoS.

Effettuando i calcoli, fondamentalmente la lettura di race come verbo ha una probabilità molto più alta di quando viene usata come nome, questo poiché i verbi in inglese vengono preceduti dal to.

Chiaramente non è sempre così semplice. È possibile vedere tali algoritmi come degli automi:

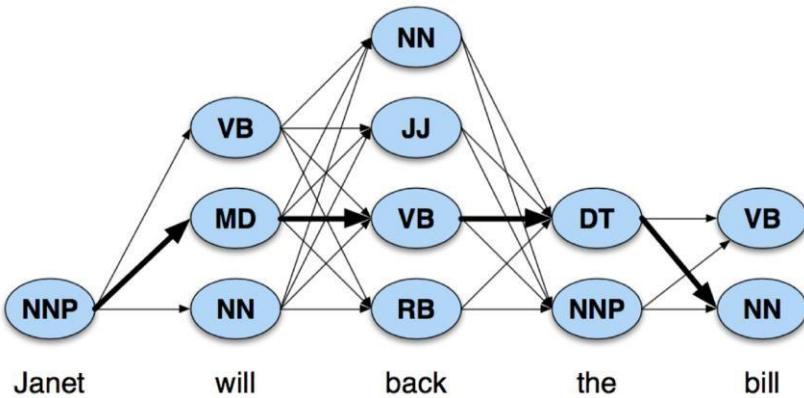


In questo modo otteniamo un automa a stati finiti in cui ogni stato è un tag, le transizioni tra stati sono le probabilità di transizione da un tag all'altro e gli stati terminali che rappresentano le probabilità di emissione delle parole. Visto che vogliamo la sequenza che massimizza la probabilità, dovremmo provare tutte le possibili sequenze di tag associate ad una frase, calcolare la probabilità e scegliere quella massima. Il problema esplode in maniera esponenziale. Tuttavia, non è necessario calcolare tutte le possibili sequenze, in quanto il modello probabilistico è un **modello a stato con assunzione Markoviana** di memoria 1, cioè vede solo lo stato precedente.

Quindi questo approccio è impraticabile poiché ci metteremmo troppo tempo. Questo problema può essere risolto applicando una tecnica di programmazione dinamica.

La **programmazione dinamica** è una tecnica molto diffusa nell'NLP nella fase di decoding e ne vedremo almeno due tipi di applicazione. L'idea è che la probabilità di una sequenza di tag può essere divisa in due parti, quindi posso considerare la probabilità della migliore sequenza che mi porta fino a $j-1$ moltiplicata per la probabilità della transizione da $j-1$ a j . Quindi è possibile calcolare la probabilità di una sequenza come il prodotto della probabilità di transizione per la probabilità di emissione per la probabilità della sequenza calcolata al passo precedente.

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. Si sostituisce alla complessità temporale una complessità spaziale che mi consente di memorizzare i dati parziali cosicché non si ricalcoli un valore già calcolato. Man mano che analizzo la mia sequenza vado solo a memorizzare quale è la sottosequenza con probabilità maggiore, man mano vengono aggiornate, questo ci permette di ridurre la complessità poiché non memorizzo tutte le possibili sequenze. La complessità è lineare nel numero delle parole e quadratico rispetto al numero di tag.



L'algoritmo di Viterbi permette di fare quello appena descritto, per farlo si crea un array con un numero di colonne pari al numero di input e un numero di righe che corrisponde ai possibili stati pari al numero di PoS + 2 (inizio e fine). L'idea è mettere una sorta di finestra che va a riempire una parte della matrice e, considerando solo 2 colonne per volta, riempie la colonna di destra in base ai valori della colonna di sinistra, partendo da uno stato iniziale arrivando ad uno stato finale. All'interno della matrice si inseriscono dei valori chiamati di Viterbi, che stanno a significare la probabilità della sottosequenza trovata sino a quel punto. Il valore di Viterbi si ottiene nel seguente modo:

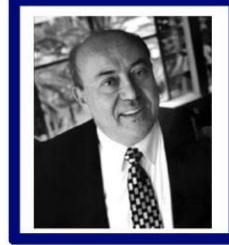
$$v_t(j) = \max_{i=1}^N v_{t-1}(i) a_{ij} b_j(o_t)$$

il massimo del valore di Viterbi precedente per la probabilità di transizione tra i tag i e j (a priori), moltiplicato per la probabilità di emissione per la specifica riga j per la parola o_t ovvero la probabilità che il tag j-esimo emetta la parola t-esima (di verosimiglianza).

function VITERBI(*observations* of len T ,*state-graph* of len N) **returns** *best-path*

```

create a path probability matrix viterbi[ $N+2,T$ ]
for each state  $s$  from 1 to  $N$  do ; initialization step
    viterbi[ $s,1$ ]  $\leftarrow a_{0,s} * b_s(o_1)$ 
    backpointer[ $s,1$ ]  $\leftarrow 0$ 
for each time step  $t$  from 2 to  $T$  do ; recursion step
    for each state  $s$  from 1 to  $N$  do
         $viterbi[s,t] \leftarrow \max_{s'=1}^N viterbi[s',t-1] * a_{s',s} * b_s(o_t)$ 
        backpointer[ $s,t$ ]  $\leftarrow \operatorname{argmax}_{s'=1}^N viterbi[s',t-1] * a_{s',s}$ 
     $viterbi[q_F,T] \leftarrow \max_{s=1}^N viterbi[s,T] * a_{s,q_F}$  ; termination step
    backpointer[ $q_F,T$ ]  $\leftarrow \operatorname{argmax}_{s=1}^N viterbi[s,T] * a_{s,q_F}$  ; termination step
return the backtrace path by following backpointers to states back in time from
backpointer[ $q_F,T$ ]
```

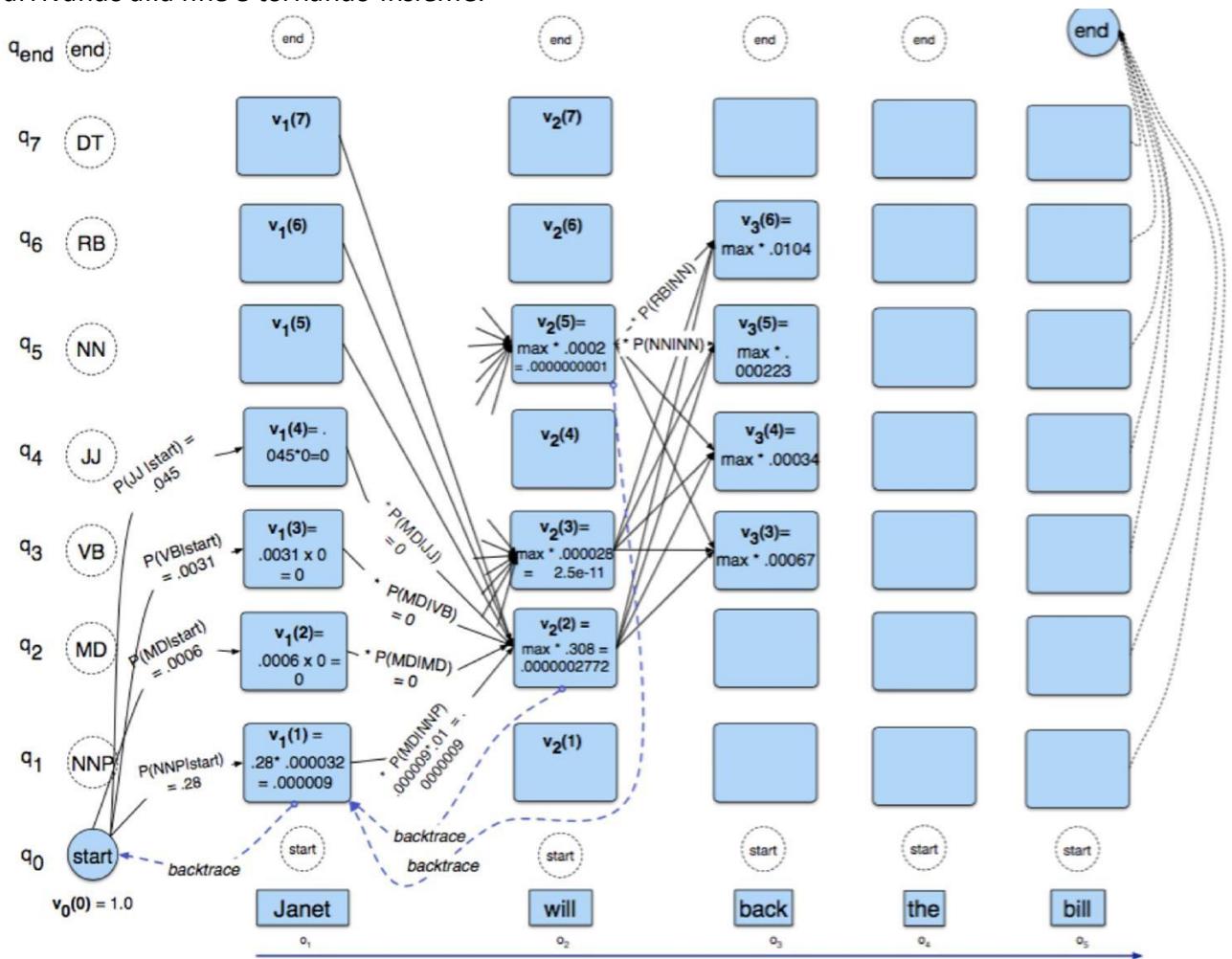


L'algoritmo di Viterbi, data una sequenza di parola di lunghezza t e uno PoS di cardinalità n , restituisce il best path sulla base del modello probabilistico, quindi è un algoritmo di decoding. Ci sono due principali for, nel primo (**initialization step**)

viene riempita la prima colonna e si basa sulla probabilità che un certo stato sia il primo PoS della sequenza, $viterbi[s,1]$ sta a indicare che per ogni PoS si considera la probabilità che tale parola sia la prima emessa con quel PoS e che sia la probabilità a priori che quel tag sia il primo della sequenza, il backpointers serve per poter ricostruire la corretta sequenza di PoS da ripercorrere all'indietro fino al primo elemento della sequenza.

Il secondo for (**recursion step**) serve per tutte le parole partendo dalla seconda fino ad arrivare all'ultima parola. Si calcola il nuovo valore di Viterbi andando a calcolare il massimo visto prima e sostituendo il nuovo valore di Viterbi. Similmente al for di inizializzazione anche in questo for c'è il back-pointer che memorizza quali sono i valori precedenti della transizione. Infine c'è un **termination step** che indica la transizione tra uno dei possibili stati rappresentati nell'ultima colonna, ovvero T , e il fittizio stato finale, in modo tale da calcolare il massimo tra i valori di Viterbi st moltiplicati per le prob di transizione $a_s q$ finale, dove q finale è uno stato fittizio e invece a_{s,q_F} rappresenta la probabilità che un tag sia l'ultimo tag di una frase, quindi anch'essa una probabilità a priori che dipendono dai tag presenti nel corpus. Sempre nel termination step viene utilizzato il backpointer che verrà srotolato così da poter dare il percorso completo che porta dallo stato finale fino allo stato iniziale nella matrice di Viterbi.

Un esempio di applicazione di tale algoritmo lo abbiamo nel seguente esempio, dove abbiamo un numero di colonne pari al numero di parole nella frase mentre il numero di righe è uguale al numero di possibili stati ovvero $7 + 2$ ovvero start ed end. Il primo step, ovvero initialization, calcola i valori per i primi valori della matrice moltiplicando le probabilità che una frase inizi con uno specifico tag a priori per la probabilità di emissione. Ad esempio, le prime due sono pari a 0 perché Janet non è un verbo, il terzo invece non è nullo perché si valuta se è un nome proprio infatti si ha un valore abbastanza alto. Un punto importante è che per poter riempire una colonna vado a vedere solo la colonna precedente. Però la sequenza di probabilità più alta la ho solo arrivando alla fine e tornando insieme.



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 e solo l'ultima cosa che succede prima a condizionare quello che succede ora.

L'algoritmo di Viterbi e il modello HMM in generale si chiamano algoritmi di memoria 1 proprio perché solo il tag precedente influenza. È possibile migliorare tale approssimazione aumentando tale memoria, ad esempio di memoria 2 che quindi dipende, nella probabilità a priori non solo dal tag precedente ma anche dal precedente ancora, ottenendo una probabilità a trigrammi, la fase di learning subirà delle leggere modifiche che sono le seguenti:

semplicemente si aggiunge anche t_{-2} come parola da considerare. Il problema che può capitare è il problema di sparseness ovvero che potrebbero esserci dei trigrammi che non compaiono mai nel nostro corpus.

$$P(t_1^n) \approx \prod_{i=1}^n P(t_i | t_{i-1}, t_{i-2})$$

```

function DELETED-INTERPOLATION(corpus) returns  $\lambda_1, \lambda_2, \lambda_3$ 
   $\lambda_1 \leftarrow 0$ 
   $\lambda_2 \leftarrow 0$ 
   $\lambda_3 \leftarrow 0$ 
  foreach trigram  $t_1, t_2, t_3$  with  $C(t_1, t_2, t_3) > 0$ 
    depending on the maximum of the following three values
      case  $\frac{C(t_1, t_2, t_3) - 1}{C(t_1, t_2) - 1}$ : increment  $\lambda_3$  by  $C(t_1, t_2, t_3)$ 
      case  $\frac{C(t_2, t_3) - 1}{C(t_2) - 1}$ : increment  $\lambda_2$  by  $C(t_1, t_2, t_3)$ 
      case  $\frac{C(t_3) - 1}{N - 1}$ : increment  $\lambda_1$  by  $C(t_1, t_2, t_3)$ 
    end
  end
  normalize  $\lambda_1, \lambda_2, \lambda_3$ 
  return  $\lambda_1, \lambda_2, \lambda_3$ 

```

Tale problema si può alleviare utilizzando la tecnica dei LAMBDA ovvero una tecnica di smoothing che al posto dello 0 mi restituisce un valore non nullo, che è dato dal calcolo dei trigrammi dei bi-grammi e degli uni-grammi pesati da dei particolari valori di lambda opportunamente calcolati su base statistica che ci permettono di avere un valore di smoothing ideale e non avere troppe probabilità nulle a priori.

VANTAGGI/SVANTAGGI HMM

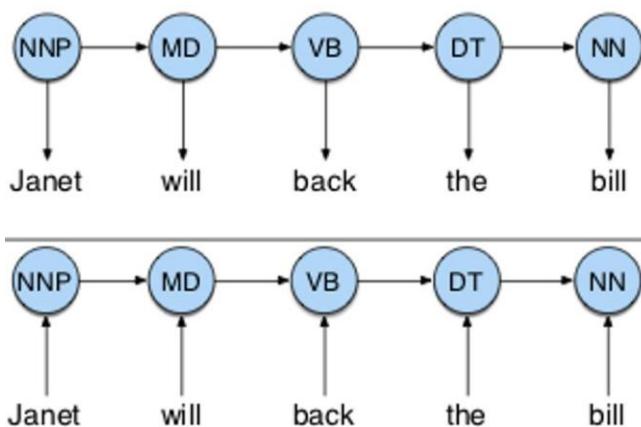
- ✓ estrema semplicità di implementazione, formalizzazione e del learning.
- ✓ Non ci sono feature linguistiche complesse. Le uniche sono il valore del tag precedente e dell'osservabile.
- ✗ È difficile introdurre delle feature di più basso livello come la terminazione di una parola o l'inizio di una parola che identificano alcune regole fondamentali.

MAXIMUM ENTROPY MARKOV MODEL

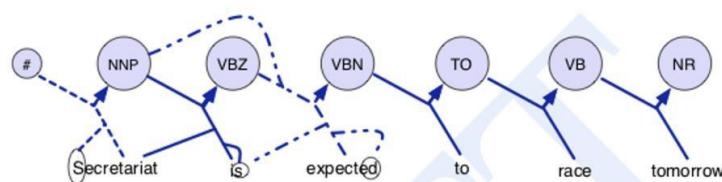
Mentre prima potevo rappresentare

l'apparizione delle parole come osservabili di una serie di stati nascosti che generano parole e che hanno probabilità di transizione nascoste tra stato e un altro stato, posso considerare questo nuovo modello come un sistema a stati, sempre in cui ci sono delle transizioni nascoste tra stati, ma in questo caso 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. È possibile definire tale modello come un modello discriminativo rispetto al modello generativo del HMM dove il modello è allenato per generare un certo dato x dalla classe y , mentre nel modello discriminativo MEMM calcola $P(y|x)$ discriminando tra i differenti possibili valori della classe y piuttosto che calcolare la probabilità di verosimiglianza calcolano direttamente sui valori degli osservabili quale è la classe più probabile.



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.



Si chiama maximum entropy model poiché una volta individuato il set di feature che ci interessa si vuole costruire un sistema che condizioni le probabilità delle label sui valori delle features. I valori di queste feautures possono essere prima di tutto la parola stessa, il prefisso della parola, il suffisso, il caso della prima parola, la forma ovvero se la parola corrisponde ad una specifica espressione regolare. Utilizzando questo approccio diretto e costruisco un modello probabilistico si hanno delle buone percentuali, a volte anche molto superiori alle HMM.

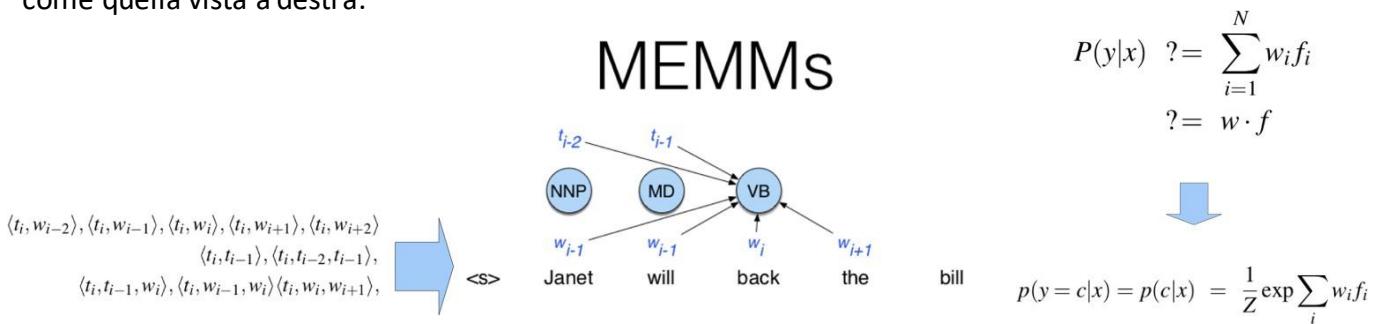
Quindi andando a combinare tale tecnica con i tag precedenti che condizionano la probabilità del tag attuale, creando un sistema MEMM che come il precedente può avere una memoria variabile e più lunga è la memoria più posso avere problemi di parsing allo stesso modo del HMM. Anche questa tecnica sarà suddivisa in modelling, learning e decoding.

Un features template è la scelta di un insieme di feature che poi andremo a stanziare sulla base dei valori presenti nel corpus di addestramento.

La scelta delle feature è data al programmatore e spesso sono delle condizioni booleane.

La fase di modelling permette di capire quale è la distribuzione di probabilità che corrisponde al sistema discriminativo contraddistinto da questa idea di Maximum Entropy model. Quello che vogliamo è che la probabilità di un certo tag sia condizionata da queste features.

Una distribuzione del genere si può dimostrare che è di tipo esponenziale infatti la formula sarà come quella vista a destra.



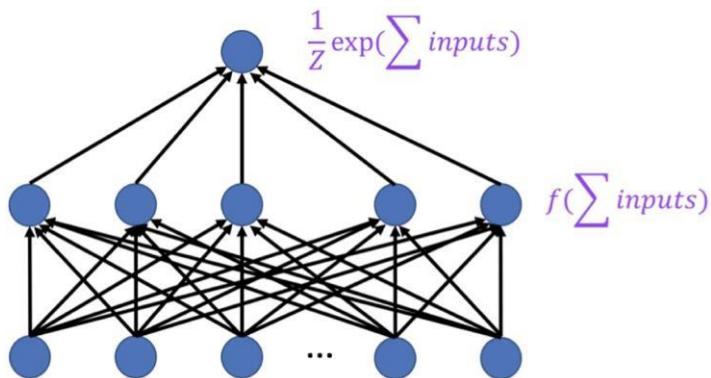
La fase di learning in questa tecnica è molto complicata poiché si tratta di ottimizzare queste funzioni esponenziali, in cui si ha bisogno di una massimizzazione che mi porti a scegliere dei pesi per le features che massimizzino la probabilità della giusta label sul corpus di apprendimento. È un problema di ottimizzazione convessa, come il gradiente.

$$\hat{w} = \operatorname{argmax}_w \sum_j \log P(y^{(j)}|x^{(j)})$$

La fase di decoding, in tale algoritmo, non è banale. Il più semplice è quello di prendere ogni volta la probabilità più alta (greedy) ma si hanno gli stessi problemi visti nella tecnica precedente. 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.

$$v_t(j) = \max_{i=1}^N v_{t-1}(i) P(s_j | s_i, o_t) \quad 1 \leq j \leq N, 1 < t \leq T$$

Il grosso vantaggio di tale tecnica è la possibilità di utilizzare features locali o globali in maniera molto sofisticata, gli svantaggi li abbiamo nel learning che è molto lento ma anche nel decoding che è molto complicato. Viene spesso usato nelle tecniche di pre-ranking, ovvero attraverso un metodo generativo più veloce si ha un insieme di output possibili e con un sistema più lento, appunto MEMM, si effettua una classifica finale dell'output.



Il modello MEMM può essere visto come un percepitrone a logistic regression, o ancor di più come un multi-layer perceptron.

Le tecniche viste per il PoS funzionano abbastanza bene ad esempio un HMM arriva al 90% e un MEMM con grandezza memoria 1 arriva al 97% se si usa invece una tecnica di bidirezionalità supera anche il 97%. Si può dire che in questo caso si è quasi raggiunta l'accuratezza degli umani. Un grande problema, come già accennato, è l'utilizzo di parole nuove o inesistenti che è impossibile riconoscere. Tale problema è notevole e influenza molto l'accuratezza finale del sistema. Esistono delle tecniche che permettono di aggirare tali parole sconosciute. Il primo metodo è di assumere che tutte le parole sconosciute sono nomi, oppure assegnare una delle parole che compaiono poco nel corpus, un terzo metodo è quella di rivolgersi ad informazioni esterne, a regole per capire cosa potrebbe essere (ad esempio se finisce con ed è un verbo) o cercare su vocabolari esterni il significato, un'altra tecnica è quella di assegnare ad ogni parola sconosciuta la stessa distribuzione.

La valutazione viene effettuata attraverso l'accuratezza, la tecnica standard su quale corpus effettuare i calcoli dividendo il corpus iniziale in training data 80%, development data 10% e test data 10% su cui effettivamente si calcola l'accuratezza. Il dataset più comune in questo ambito è il PennTreeBank.

LA SINTASSI

COMPETENCE

CHOMSKY ha fatto la storia della linguistica e dell'informatica. Molti si sono ispirati alle tecniche linguistiche da lui affrontate per far fronte a problemi di diversa natura. Lui si pose il problema della complessità del linguaggio naturale e quindi della sequenza delle parole che compongono una frase e quindi della struttura che caratterizza una frase linguistica (**struttura sintattica**).

Il linguaggio umano, come possibile notare in queste 3 frasi, tende ad essere sempre più complesso e ad avere più incassamenti, anche il concetto di ricorsione rappresenta un elemento fondamentale nella strutturazione delle frasi come ad esempio la canzone alla fiera dell'est che caratterizzano la complessità del linguaggio umano.

| | | |
|---|---|--|
| A man that a woman that a child knows loves | A man that a woman that a child that a bird saw knows loves | A man that a woman that a child that a bird that I heard saw knows loves |
|---|---|--|

SINTASSI E GRAMMATICHE GENERATIVE

La sintassi è quella parte della linguistica che si occupa della formazione delle frasi che intende anche l'ordine delle parole e la formazione dei sintagmi ovvero gruppi di parole che mostrano delle caratteristiche di tipo sintattico.

La sintassi è importante perché indica l'ordine delle parole e condiziona la semantica della frase. Un pezzo centrale e fondamentale della sintassi è dato dalla **competence**, ovvero quella conoscenza sintattica che si sviluppa nell'apprendere una certa lingua (ad esempio nel caso dell'italiano ci direbbe che il soggetto viene prima del verbo). Chomsky dice che nella linguistica bisogna disinteressarsi delle **performance** (algoritmo di parsing) ma dare importanza allo studio delle competenze linguistiche al di là del loro uso, 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, che sembra essere banale ma ha una potenza elaborativa molto elevata. Chomsky inoltre definisce la grammatica generativa, utilizzata poi da parte degli informatici anche per definire i linguaggi di programmazione.

Rewriting rule

$\Psi \rightarrow \Theta$

Tali grammatiche sono caratterizzate da 4 elementi:

1. Alfabeto Σ = simboli terminali, per noi sono le parole;
2. Una serie di simboli non terminali $V = \{A, B, \dots\}$;
3. Un simbolo speciale ($start S \in V$) appartenente ai non terminali e rappresenta l'insieme degli stati iniziali;
4. Regole di produzione (regola di riscrittura). Secondo Chomsky se dimentico che una frase ha un significato e la elaboro come se fosse un linguaggio formale potrei individuare frasi corrette da frasi scorrette, dove però il concetto di correttezza rimane quello puramente sintattico. Permettono di riscrivere una sequenza di simboli con un'altra. Inoltre, esse ci restituiscono l'albero di derivazione, ovvero le varie trasformazioni dallo stato iniziale allo stato finale.

In particolare, vengono equiparati i simboli non terminali ai costituenti ovvero delle strutture sintattiche particolari (come la struttura VP che indica una forma verbale) e grazie alle regole di produzione si può capire la tipologia di un verbo ad esempio.

Es: la seguente grammatica che ha 5 simboli non terminali (con poche regole e un alfabeto piccolo, in realtà si hanno 100000000 parole) ci permette, partendo da S e facendo girare la grammatica attraverso le regole, di generare delle frasi sintatticamente corrette.

$$G4 = (\Sigma_4, \{S, NP, VP, V1, V2\}, S, P4) \rightarrow \text{grammatica}$$

$$\Sigma_4 = \{I, Anna, John, Harry, saw, see, swimming\} \rightarrow \text{simboli terminali}$$

$$P4 = \{S \rightarrow NP\ VP,$$

$$VP \rightarrow V1\ S,$$

$$VP \rightarrow V2,$$

$$NP \rightarrow I | John | Harry | Anna,$$

$$V1 \rightarrow saw | see,$$

$$V2 \rightarrow swimming \} \rightarrow \text{regole di produzione}$$

Applichiamo le seguenti regole e otteniamo l'albero di derivazione di destra

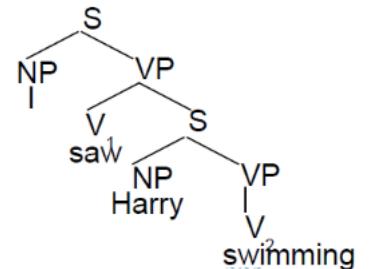
$$S \rightarrow NP\ VP\ S$$

$$VP \rightarrow V1\ S$$

$$VP \rightarrow V2\ NP \rightarrow I | John | Harry | Anna$$

$$V1 \rightarrow saw | see$$

$$V2 \rightarrow swimming$$



Una caratteristica di questa grammatica è la **ricorsione** indiretta, quindi può generare infinite frasi grammaticali, alcune difficili da capire per questioni di limiti di memoria, attenzione ecc... La potenza della grammatica è la **ricorsione**.

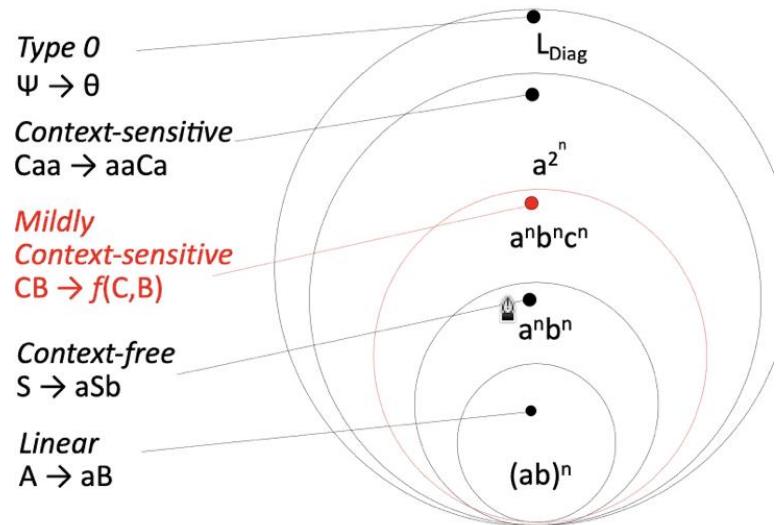
L'albero di derivazione non indica l'ordine di applicazione delle regole, ma ci dice solo quali regole sono state applicate! Un algoritmo di parsing ci dice anche l'ordine di applicazione delle regole.

GERARCHIA DI CHOMSKY

Il quesito principale di Chomsky era capire quanto fosse complesso il linguaggio naturale e le strutture generate dalle grammatiche. Per fare ciò ha stilato la gerarchia di Chomsky. Essa viene usata per definire la complessità dei linguaggi di programmazione.

Nell'immagine vediamo le 4 tipologie di grammatiche con la loro annessa complessità. Ma all'interno di questa gerarchia, dove si trova il linguaggio naturale? Ovvero, quanto risulta essere complesso? Chomsky dice che sicuramente non può essere lineare ma non sa determinare quale livello di complessità effettivamente esso abbia: nessuno fino agli anni 80 è riuscito a rispondere a tale domanda.

Shieber dice che sicuramente non è neanche context-free dimostrandolo con una frase dialettica tedesca che indicava con una frase mista tedesca e svizzera che non era libera da contesto. Nessuno ancora è riuscito a dimostrarlo in maniera formale.



Mildly Context Sensitive Languages (MCS) - Congettura di Joshi

Per Joshi le lingue naturali sono context sensitive ma solo leggermente al di sopra delle context-free (nell'immagine la categoria in rosso, **MCS**) chiamata tiepidamente libere da contesto.

Per lui appunto 1. non sono context-free (ma solo per poco), 2. tutte le lingue naturali mostrano

solo due tipi di dipendenze incrociate (**Nested e cross-serial**)

ovvero il legame sintattico che lega due parole in una frase

come nell'immagine, 3. sono parsificabili in un tempo

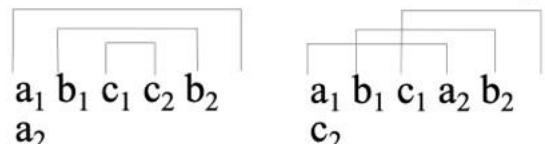
polinomiale e infine dice che 4. tutte le frasi che costituiscono

una lingua, hanno la proprietà di crescita lineare ovvero ci dice

come le frasi corrette di una lingua crescono linearmente in lunghezza. data una frase lunga n, ci

sarà sempre una frase lunga $n+\epsilon$ che è comunque grammaticale. È legato alla ricorsività delle

lingue naturali, che garantiscono questa proprietà



Negli anni successivi, in pura linguistica computazionale, ci si pose il problema di quali grammatiche utilizzare per generare delle MCS. Vennero presentati una serie di formalismi che cercavano di rispondere a questa esigenza. Un primo esempio fu presentato proprio da Joshi, le **Tree Adjoining Grammars**, rappresentano il formalismo più famoso e costituiscono una leggera evoluzione delle context-free. Altri esempi sono le **Head Grammars**, **Linear Indexed Grammars** e **Combinatory Categorical Grammars**. Questi 4 formalismi si sono dimostrati equivalenti e tutti sono adatti a rappresentare il linguaggio naturale. Tutti fanno una cosa simile ovvero potenziano leggermente le context-free cambiando leggermente le strutture elementari e le regole di combinazione.

Tree Adjoining Grammars (TAG - Joshi)

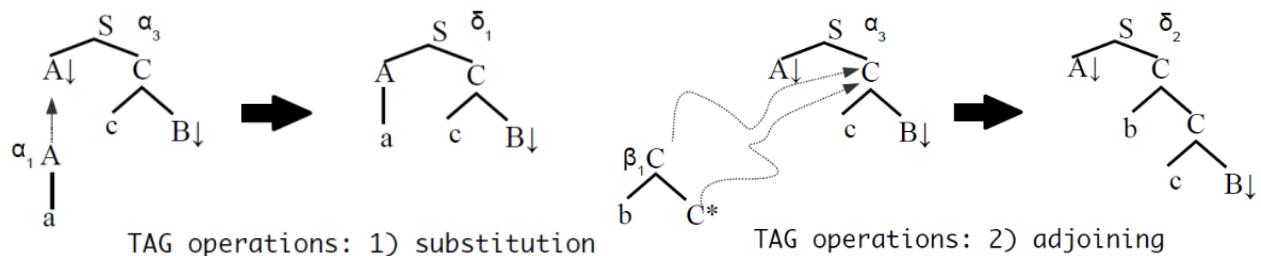


Sono Midly context sensitive e modellano bene la linguistica.

Il modello **Tree Adjoining Grammars** va a potenziare le strutture elementari delle context free permettendo l'utilizzo di strutture multilivello basate sugli alberi (**Tree**). Non viene più usata la struttura a stringhe delle context-free, ma si passa all'utilizzo di alberi che permettono di costruire strutture con più di un piano, quindi più complesse.

Invece di avere una sola operazione di sostituzione, ne abbiamo due:

- **Tree substitution:** innesta un albero in un altro per ottenere un nuovo albero, lo innesta a livello delle foglie, se l'albero ha come radice lo stesso simbolo non terminale della foglia rimossa. Solitamente utilizzata dai nomi. La regola di sostituzione è simile a quella usata nelle grammatiche generative, quindi non estende il potere espressivo delle CF.
- **Adjoining:** permette di innestare un albero tra due alberi. L'innesto è a livello di nodi, purché il nodo rimpiazzato abbia la stessa radice dell'albero ausiliario. Ciò cambia il potere generativo e siamo in grado di generare il tedesco svizzero. Solitamente utilizzata da verbi e avverbi. Ha una complessità di $O(n^6)$.



Alcuni studi hanno dimostrato che le TAG sono un buon strumento per elaborare il linguaggio naturale. In particolare, le TAG hanno delle caratteristiche peculiari in ambito linguistico, in particolare è dimostrabile che hanno 3 proprietà linguistiche che ne permettono l'utilizzo reale:

1. Dominio di località esteso;
2. Fattorizzazione di ricorsione attraverso le operazioni di Adjoining;
3. Lessicalizzazione.

Combinatory categorical Grammar - CCG

Le **CCG** sono un altro formalismo grammaticale che però ha un altro approccio sempre in grado di catturare il linguaggio naturale. Tale formalismo è stato presentato da Mark Steedman, un linguista psicologo informatico che cerca di dare una luce completamente diversa al concetto della sintassi allontanandosi dall'idea tradizionale e dicendo che la sintassi non è una struttura ma un processo, quindi la cosa importante in una frase non è la struttura che ne risulta, ma il modo in cui la costruiamo poiché questo influenza molto il significato semantico. Ci si allontana quindi dall'approccio top-down generativo fino ad ora affrontato e ci si avvicina ad un approccio categoriale e quindi bottom-up; si parte dalle parole, uso regole di combinazione che fanno parte del lessico e che definiscono delle categorie, e a loro volta altre regole che mi permettono di combinare le parole. È una procedura molto diversa da quella top-down. Per i linguisti questa tecnica è in grado di modellare aspetti che le altre non riuscivano a fare, come ad esempio la coordinazione dei non costituenti.

In breve: Ad ogni parola si associano delle categorie che possono combinarsi. Se alla fine della combinazione ci danno la categoria S, allora la frase è grammaticale.

Esempio

Paolo: NP

Francesca: NP

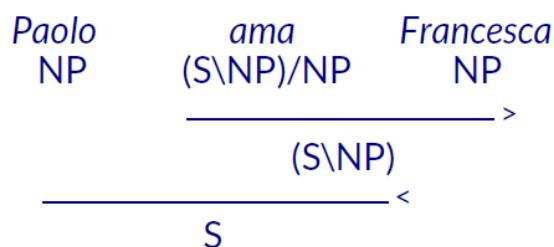
Amare: $(S \setminus NP)/NP$ → categoria complessa che indica le possibili combinazioni con le categorie adiacenti.

Possiamo avere NP a dx e a sx.

Abbiamo delle regole di combinazione che ci dicono come si combinano le categorie.

L'albero è rovesciato. Partiamo dalle categorie complesse (sulle foglie), che si riducono ad S se la frase è grammaticale.

Abbiamo due regole: -> e -<



PERFORMANCE

Algoritmo di Parsing

Bisogna permettere l'elaborazione del significato che si trova sotto alle frasi. Per ora, i chatbot cercano di fare questo.

Per capire il funzionamento di un parser sintattico bisogna prima definire il concetto di **Parser Anatomy** ovvero una maniera per rappresentare le caratteristiche di un parser per il linguaggio naturale inventata da Mark Steedman. Poi si affronterà la differenza tra parser top-down e parser bottom-up, e poi parleremo di uno speciale algoritmo di parsing che prende il nome di CKY che sta per , proposto da questi ultimi e basato sulla programmazione dinamica come l'algoritmo di Viterbi (Possibile domanda d'esame). Si inizierà poi a parlare di Parsing probabilistico e il TB grammar, approcci che mischiano il simbolico con la statistica (che non sono assolutamente antitetici). L'ultimo argomento tratterà il Parsing parziale. Questi algoritmi risultano lenti a real time.

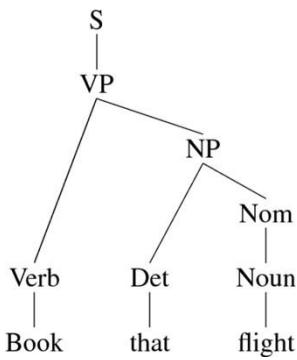
ANATOMIA DI UN PARSER

L'anatomia di un parser è una maniera per definire completamente come è fatto un parser sintattico, distinto in 3 punti:

1. **Grammatica usata dal parser** (competence). Viene rappresentata da una grammatica per i parser che generano alberi sintattici. (CF, TAG, CCG,...)
2. **Algoritmo**: la scelta della strategia di ricerca ovvero che ci indica le direzioni di partenza e di arrivo dell'algoritmo (top-down, bottom-up, left-to-right ecc). Questo riguarda sia la costruzione che il metodo di visualizzazione dell'input. Inoltre, l'algoritmo ha a che fare con la memory organization, ovvero man mano che vado ad attuare l'algoritmo seguendo la strategia di ricerca, avrò una serie di dati potenziali, quello che fa è dirmi come organizzo questi dati parziali. Esploro in diverse maniere le soluzioni, ad esempio una alla volta fino a che posso e se sbaglio faccio backtracking all'ultimo punto di scelta. Un altro approccio è quello della programmazione dinamica, in cui non cerco una soluzione per volta ma esploro contemporaneamente diverse strade, e questa esplorazione in ampiezza mi deve portare ad una soluzione. Il vantaggio della prima tecnica è che occupa meno memoria rispetto alla seconda che invece ne occupa tanta, a discapito però la prima è più lenta della seconda in molti casi un tempo inapplicabile.
3. **Oracolo**: è un altro elemento del parser che deve indicarmi davanti a delle scelte quale scelta è migliore tra le possibili (ad esempio un'euristica nel labirinto). Questo oracolo può avere diverse nature, come il calcolo probabilistico o rule based.

Utilizzando una grammatica context-free (per ridurre la complessità) testeremo le strategie di top-down e bottom-up. La grammatica giocattolo è la seguente:
Le regole in questa grammatica sono 26.

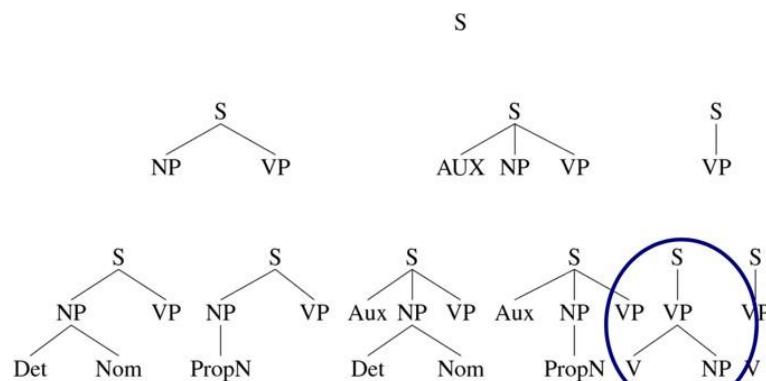
| | |
|------------------------------------|---|
| $S \rightarrow NP VP$ | $Det \rightarrow that this a$ |
| $S \rightarrow Aux NP VP$ | $Noun \rightarrow book flight meal money$ |
| $S \rightarrow VP$ | $Verb \rightarrow book include prefer$ |
| $NP \rightarrow Det Nominal$ | $Aux \rightarrow does$ |
| $Nominal \rightarrow Noun$ | |
| $Nominal \rightarrow Noun Nominal$ | |
| $NP \rightarrow Proper-Noun$ | $Prep \rightarrow from to on$ |
| $VP \rightarrow Verb$ | $Proper-Noun \rightarrow Houston TWA$ |
| $VP \rightarrow Verb NP$ | $Nominal \rightarrow Nominal PP$ |



Supponendo di voler raggiungere tale obiettivo, dando in input “Book that flight” vogliamo il seguente parser. I due approcci sono partendo dalla grammatica o partendo dalla frase. In realtà ci si basa su entrambe le informazioni per arrivare ad una soluzione.

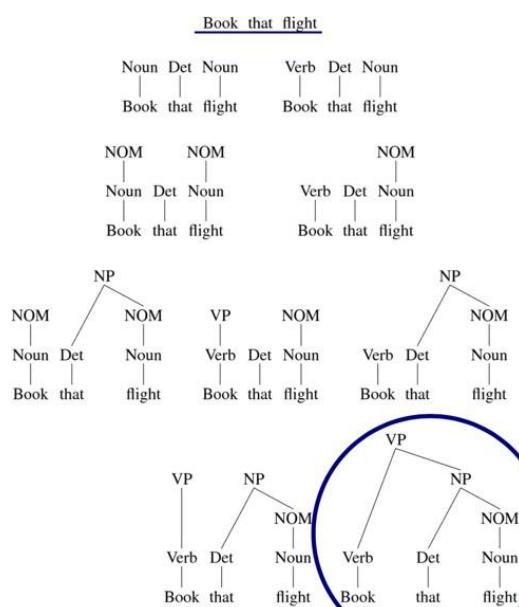
Partendo però dalla grammatica vuol dire partire dall’alto ovvero dal termine di start S (**approccio top-down**), con il vantaggio che qualsiasi soluzione parziale che trovo forma delle frasi che però potrebbero non essere affini alle parole, dei vicoli ciechi che non tengono conto dell’input (ragionamento dei razionalisti), l’oracolo permette solo ad esempio di ordinare le varie strade da scegliere.

Perdo tanto tempo e occupo tanto spazio.

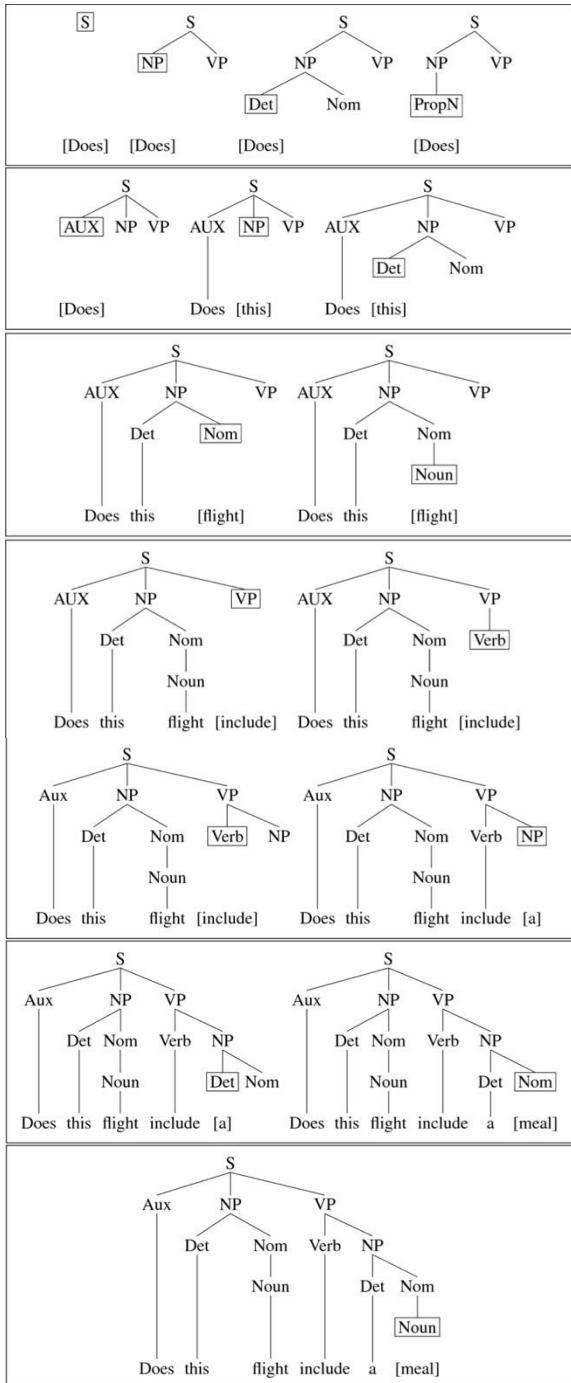


L’altra tecnica è partire dalle foglie, quindi si avranno ricerche solo compatibili con l’input, ma in ogni caso ci sarà la creazione di alberi che non mi portano a nessun risultato effettivo (ragionamento degli empiricisti). Partendo dalle parole, guardando la grammatica, creo le varie possibili strade per le parole, come vediamo nell’esempio sono due inizialmente.

Anche qui ci sono una serie di soluzioni parziali che vengono create e solo una di queste effettivamente mi interesserà. Anche qui allora abbiamo spreco di spazio e tempo per arrivare effettivamente la soluzione.



Queste problematiche le abbiamo utilizzando delle strategie molto semplici, per questo si cerca un approccio più elaborato come ad esempio il Parser A.



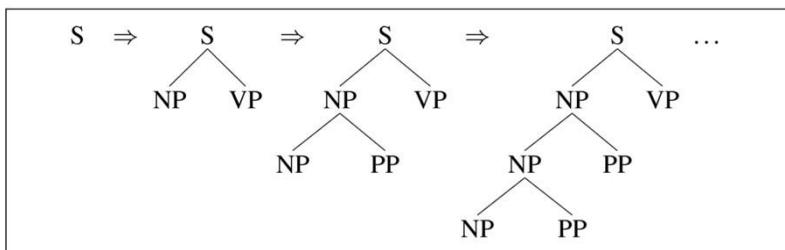
Il Parser A, con una competence context-free, con approccio top-down left-to-right, con organizzazione di memoria con back-tracking con un oracolo rule-based che prende la prima regola presente nella grammatica.

Preso la frase "Does this flight include a meal".

In questo modo, risolviamo i problemi di spazi. Espando applicando una regola alla volta trovando delle soluzioni parziali, si può notare che arrivando a Prop N, arrivo ad un vicolo cieco perché a regola per Prop N includeva altre regole che non sono presenti, quindi faccio backtracking e considero la regola con AUX iniziale e procedo con la verifica di completezza della regola. Si simula la ricerca all'interno del parser per ogni regola. Continuando la risoluzione con le regole della grammatica vediamo che arriviamo ad una parsing corretto, con una soluzione molto bella che ci dà anche informazioni di tipo linguistico che ci permettono nella fase successiva di capire il significato interno. Come ad esempio è possibile capire che è una domanda.

Questo approccio non occupa spazio ma è impraticabile come complessità di tempo e per la mancanza di informazione che parte dalla frase.

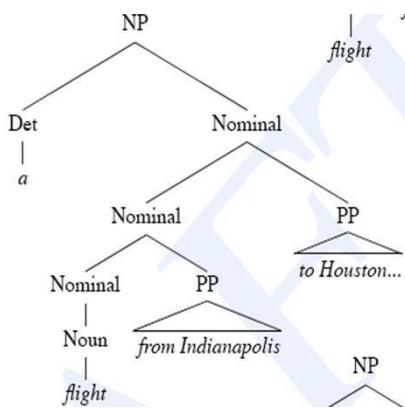
$NP \rightarrow NP\ PP$



Nella grammatica c'è una regola di ricorsione a sinistra che crea una quantità di strutture elevate.

Questo potrebbe mandare il mio sistema in loop perché è sempre possibile applicarla senza sosta, quindi la soluzione non la si troverà mai. Quindi questo parser è debole e non può essere utilizzato.

Inoltre, questo parser nelle regole non direttamente ricorsive, ci porta all'inefficienza anche a causa dell'oracolo. Ad esempio, nella frase "a flight from Indianapolis to Houston on TWA" elaborando parola per parola arriveremo a dire prima che il nominal si divide in nominal e PP dove mettiamo flight e from Indianapolis, ma poi trovando dopo to Houston, mi rendo conto che in realtà andava diviso ulteriormente il primo Nominal, allora faccio back-traking e prendo un nuovo punto di



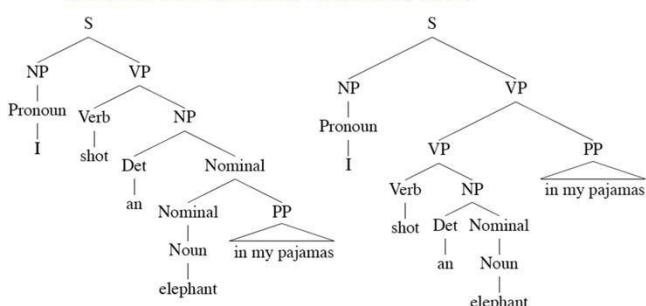
scelta sul primo nominal che mi divide ulteriormente in Nominal e PP. E così via quando incontro on TWS dovrà eseguire back traking e tutte le volte applicare tutte le possibili regole. Quindi fondamentalmente il mio sistema non si ricorda cosa ha fatto nelle strade precedenti e ogni volta effettua operazioni come all'inizio. Non è possibile eliminare tale tipologia di ricorsione che prende il nome di "ricorsione sinistra". Bisogna trovare una soluzione di parser più furba che mi permetta di ovviare a questo problema e al problema computazionale, oltre a quello di ambiguità. Infatti, sappiamo che l'ambiguità non dipende solo dall'algoritmo ma anche dalla competence, proprio perchè il linguaggio naturale è ambiguo. Una frase può avere più tipologie di regole, decontestualizzando la frase, che indicano più percorsi possibili che il parser potrebbe restituire tutti validi.

In particolare, esistono due fenomeni che fanno riferimento all'ambiguità sintattica, che sono l'attachment e la coordinazione.

One morning I shot an elephant in my pajamas.

How he got into my pajamas I don't know.

Groucho Marx, *Animal Crackers*, 1930



In questa frase tratta da un film, c'è l'ambiguità sintattica sul mio pigiama, poiché esso può indicare che ho sparato con il pigiama o che l'elefante si trovava nel il mio pigiama. Questo tipo di ambiguità è quella di PrepositionalPrhase attachment.

... televisioni e radio nazionali ...

Si può mangiare e bere qualcosa

Si può mangiare e pagare qualcosa

Questo fenomeno

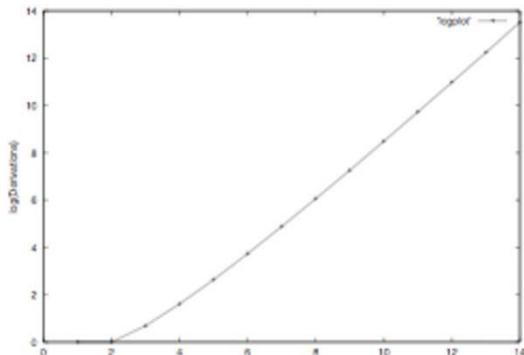
invece è di

coordinazione, ovvero

capire i termini alla fine

qualcosa fa riferimento sia a mangiare che bere o solo bere.

Queste sono dei fenomeni che fanno esplodere il numero di alberi di risoluzione possibili, e bisogna quindi tener conto di queste ambiguità. La maggior parte dello Humor è relato all'ambiguità semantica. Questa ambiguità strutturale è stata studiata da Charc e Patil nell'82 e ci hanno mostrato che questa ambiguità cresce in base al numero di parole contenute nella frase. Questo grafico ci mostra la crescita esponenziale in base al numero di parole, e tal crescita è stata formalizzata con un coefficiente binomiale attraverso dei numeri catalani che governano la complessità massima dovuta alla PP Attachment e alla Coordinazione.



Per gestire tali problemi di esplosione combinatoria (ovvero il fatto che i due fenomeni appena visti, ci creano un numero di sottoalberi potenzialmente corretti in un numero esponenziale in base al numero di parole della frase), similmente a come abbiamo visto quando abbiamo presentato l'algoritmo di Viterbi, passando alla programmazione dinamica. Questa tecnica di programmazione generale ha l'idea di memorizzare delle sottostrutture ottimali (Il nome dinamico non c'entra niente con l'algoritmo ma indica una bella parola) che crea sottoalberi sovrapponibili (nell'algoritmo di Viterbi, è dinamica perché memorizza la soluzione precedente nella matrice di Viterbi che utilizzerà per la prossima esecuzione) indicate da alberi di parsing possibili.

Esistono diversi algoritmi che applicano la programmazione dinamica:

1. il più classico è l'algoritmo di CKY che calcola tutti i possibili parser in tempo $O(n^3)$ (ottimale rispetto alla complessità di una context-free) quindi risulta essere semplice. Il principale difetto di questa tecnica è che il caso peggiore e il caso medio coincidono inoltre esige una forma normale per la grammatica, essa va prima trasformata nella forma normale di grammatica ovvero la forma normale di Chomsky.
2. Altre forme di programmazione dinamica è l'algoritmo Earley e ha il vantaggio di avere una complessità ancora minore di $O(n^2)$ ma è molto più complicato e per questo non lo vedremo.
3. Un'altra alternativa è il Deterministic Parsing, che può essere top-down (LL Parsing) o bottom-up (LR o Shift-reduce parsing) che hanno una complessità lineare e si può applicare ad esempio se si vuole fare un parsing basato su Java.

N.B.: Memoizzazione (senza r): consiste nel restituire un certo output (memorizzato localmente) quando esso è già stato calcolato e ne viene richiesto nuovamente il calcolo. In questo modo si evita una nuova esecuzione della funzione avendo già il risultato.

In cky questo corrisponde a riempire una matrice (simile a viterbi)

Cocke Kasamy Younger - CKY

Questo algoritmo è basato su una grammatica di tipo context-free (applicabile anche alle TAC TCG), utilizza un algoritmo con strategia di ricerca bottom-up e left-to-right con programmazione dinamica e un oracolo basato su regole molto semplici.

Supponendo di avere solo due regole diverse nella parte sinistra ma uguali nella parte destra, allora nel caso di sostituzione avrò che ci sono due sottoalberi possibili per tale risoluzione. Quindi gli stessi sottoalberi utilizzati per A saranno riutilizzati allo stesso modo per ricostruire D.

Quindi memorizzo solo uno dei due e lo rendo disponibile per entrambe le regole. Memorizzando a runtime in una matrice le soluzioni parziali, se mi rendo conto che con A non trovo la soluzione, verrà esclusa anche la regola per D. Per essere applicabile abbiamo accennato che la grammatica deve essere in **CNF (Chomsky Normal Form)**: le regole sono nella formula indicata $A \rightarrow B C$ cioè non terminale in terminale oppure $A \rightarrow d$.

Qualsiasi grammatica context-free può essere portata nella forma normale di Chomsky quindi non è un problema applicare il CKY. Ad esempio, applicando l'algoritmo di trasformazione alla grammatica giocattolo vista prima avremo una nuova grammatica con più regole.

Applicando l'algoritmo CKY alla frase "Book the flight trough Houston" avremo come risultato una matrice come quella in figura, ovvero una matrice riempita solo nella metà superiore, che indica un insieme di alberi impacchettati, in cui considero tutti i possibili alberi costruibili sulla grammatica a partire dalla frase e li impacco in ogni cella. Partendo da 0 riempio la matrice da sinistra a destra. Quindi considerando solo 0 e 1 rappresenta la struttura della frase "Book the", e poi cerco di combinare l'albero precedente, in base alle regole, per la nuova frase "Book the flight". Il vantaggio è che in una stessa cella possiamo avere diverse configurazioni per la cella precedente. In questo modo si sostituisce la complessità temporale in complessità spaziale che aumenta esponenzialmente ad ogni nuova parola. Ad esempio considerando la regola $A \rightarrow BC$ vuol dire che se c'è un A allora c'è un B seguito da un C, e inoltre che se A va da $i \rightarrow j$, c'è un k tale che $i < k < j$, cioè B va da $i \rightarrow k$ e C va da $k \rightarrow j$, allora usando una tabella se B è nella **tabella[i,k]** e C è in **tabella[i,j]**, allora posso aggiungere nella matrice che A è in **tabella[i,j]**. L'obiettivo è riempire la matrice da sinistra a destra seguendo diverse soluzioni riempendo più caselle dal basso verso l'alto.

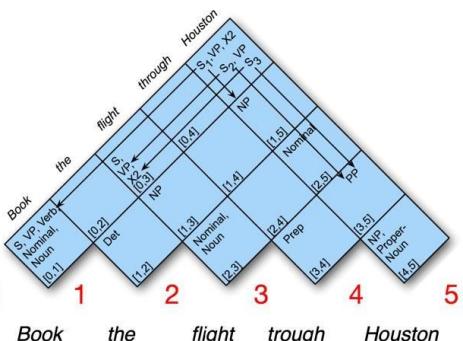
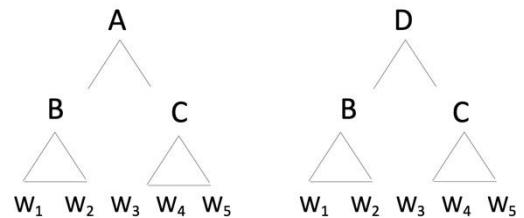
Se si verificano le 3 condizioni immediatamente sulla sinistra, allora si andrà ad inserire il simbolo A all'interno della tabella con indici i e j.

Con il seguente pseudocodice, da alcune parole in ingresso (la frase) e la grammatica, si ha in output una tabella. Questo algoritmo è ottimizzato negli indici. J fa un loop sulle colonne e va a riempire i simboli preterminali (effettivamente è un ciclo che inizializza la parte lessicale e potrebbe anche essere eseguito a parte), il ciclo della i invece, riempie la riga i nella colonna j, e infine il ciclo k, cerca se tra i e j esiste un nuovo albero interno aggiungibile.

invece, riempie la riga i nella colonna j, e infine il ciclo k, cerca se tra i e j esiste un nuovo albero interno aggiungibile.

$A \rightarrow B C$

$D \rightarrow B C$



```

function CKY-PARSE(words, grammar) returns table
  for j ← 1 to LENGTH(words) do Looping over the columns
    table[j - 1, j] ← {A | A → words[j]} Filling the bottom cell
    for i ← j - 2 downto 0 do Filling row i in column j
      for k ← i + 1 to j - 1 do Looping over the possible split locations between i and j.
        table[i, j] ← table[i, j] ∪ {A | A → BC ∈ grammar,
          B ∈ table[i, k],
          C ∈ table[k, j]}
  
```

Check the grammar for rules that link the constituents in $[i, k]$ with those in $[k, j]$. For each rule found store the LHS of the rule in cell $[i, j]$.

[j : _{1..n}] [i : _{j-2..0}] [k : _{i+1..j-1}]]]

| Book | the | flight | through | Houston |
|-------------------------------------|-------|--------------------|---------|---------|
| S, VP, Verb, Nominal, Noun [0,1] | | S, VP, X2 [0,3] | [0,4] | [0,5] |
| [0,2] | | [1,3] | [1,4] | [1,5] |
| Det | NP | | | |
| [1,2] | [1,3] | [2,3] | [2,4] | [2,5] |
| Nominal, Noun | | | | |
| [2,3] | | | | |
| Prep | PP | | | |
| [3,4] | | | | |
| [3,5] | | | | |
| NP, Proper-Noun [4,5] | | | | |

j=5

i=3

k=4

Consideriamo solo la colonna 5

Il K va a tentativi e quindi cerca ogni possibile configurazione tra I e J. Se esiste aggiunge il nodo tra i nodi di I e J. La vera parte fondamentale quindi è proprio quella del loop K. In quest'ultimo ciclo si attua il principio Considerando il momento in cui gli indici hanno i valori in figura, lasciando j fisso, Supponendo che nella posizione con il pallino rosso ho NP e nella posizione con bollino verde ho Prep, vedendo le regole ho PP Prep NP, sono abilitato a riempire nella casella [3,5] PP.

Andando avanti con la i, posso avere una k = 3 e noto che con le caselle in questione ho una regola all'interno della grammatica sostituibile ovvero Nominal.

Sostituendo ancora con i=1 e quindi k=2 potrò ancora una volta attraverso le regole aggiungere NP nella casella [1,5].

N.B.: Quando ho più simboli all'interno della stessa cella vuol dire che posso avere più alberi sintattici

[j : _{1..n}] [i : _{j-2..0}] [k : _{i+1..j-1}]]]

| Book | the | flight | through | Houston |
|-------------------------------------|-------|--------------------|---------|---------|
| S, VP, Verb, Nominal, Noun [0,1] | | S, VP, X2 [0,3] | [0,4] | [0,5] |
| [0,2] | | [1,3] | [1,4] | [1,5] |
| Det | NP | | | |
| [1,2] | [1,3] | [2,3] | [2,4] | [2,5] |
| Nominal, Noun | | | | |
| [2,3] | | | | |
| Prep | PP | | | |
| [3,4] | | | | |
| [3,5] | | | | |
| NP, Proper-Noun [4,5] | | | | |

j=5

i=2

k=3

[j : _{1..n}] [i : _{j-2..0}] [k : _{i+1..j-1}]]]

| Book | the | flight | through | Houston |
|-------------------------------------|-------|--------------------|---------|--------------------|
| S, VP, Verb, Nominal, Noun [0,1] | | S, VP, X2 [0,3] | | S, VP, X2 [0,5] |
| [0,2] | | [1,3] | [1,4] | [1,5] |
| Det | NP | | | |
| [1,2] | [1,3] | [2,3] | [2,4] | [2,5] |
| Nominal, Noun | | | | |
| [2,3] | | | | |
| Prep | PP | | | |
| [3,4] | | | | |
| [3,5] | | | | |
| NP, Proper-Noun [4,5] | | | | |

j=5

i=0

k=1,3,3

È importante notare che ogni cella della matrice non è un semplice set, ma un array, una lista, così che sarà possibile inserire tutti i simboli che voglio.

Andando ancora avanti spostando i=0, avrò che k = 1, 3 potrò inserire tutte le regole possibili, in questo caso 2. Quindi bisogna ragionare su quale è la corretta tra le possibili. Interessante è vedere che in questo caso nella casella [0,5] ci sono due possibili S che fanno riferimento ad un problema di attachment: esse sono entrambe corrette ma ci sta dicendo Houston fa riferimento al volo o all'acquisto? La lettura corretta è il VP che si riferisce al

volo ma per la conoscenza che noi abbiamo del mondo, ma chi ci dice che questo è così?

Effettivamente a questo punto sintattico non è possibile distinguere le letture, sarà solo dopo nella fase di analisi semantica. Infatti, la frase è ambigua a livello sintattico. I

Esistono altri casi in cui non c'è ambiguità sintattica ma invece c'è ambiguità semantica.

È importante notare che se nella mia cella [0, n] non ho almeno una S allora la frase non è sintatticamente corretta, solo se S appartiene a [0,n] allora la frase è grammaticale. Le frecce che vediamo nell'immagine, sono proprio le informazioni che devono essere memorizzate per poter riconoscere l'albero.

La complessità è $O(n^3) = k * (n^3)$ dove n è il numero di parole, dovuto ai cicli for che vanno avanti in base al numero di parole. Ma chi sta nella costante k? La grammatica, quindi il numero di regole aumentano in maniera lineare la complessità totale.

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

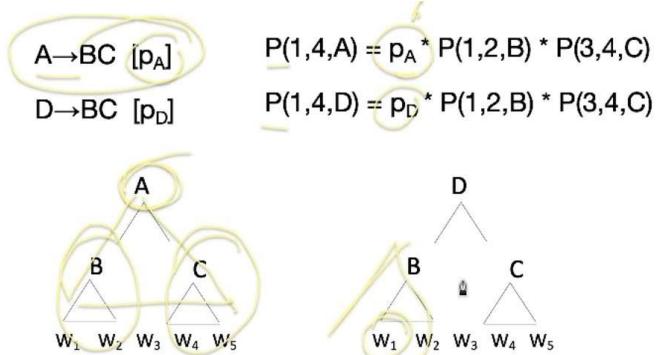
PARSING PROBABILISTICO E TB GRAMMARS

Devo cercare di eliminare possibili strade che non sono potenzialmente promettenti, per fare ciò si implementa un parser con oracolo probabilistico.

CKY Probabilistico

Si calcola la probabilità delle regole, partendo dalle foglie, e tale distribuzione di probabilità, permette di calcolare la probabilità totale degli alberi.

Quindi come nel caso precedente potrò scegliere ancor prima di passare alla semantica quale degli alberi è più frequente nella vita reale.



L'idea per calcolare è che ad ogni regola assegno una probabilità tra 0 e 1 dove però non si definisce mai con 0 o 1. Ogni volta quindi abbiamo un numero che ci indica quanto è probabile scegliere quella regola piuttosto che un'altra.

N.B.: mai con 0 perchè significa che è una regola impossibile, una regola con probabilità 1 sarebbe una regola certa.

Si introducono queste probabilità perchè i parser visti finora sono lenti, e inoltre come già detto, si riesce a distinguere una S rispetto alle altre probabilisticamente, anche se poi l'ultima parola sarà data alla semantica. Infatti, questa tecnica è una che più si avvicina a quella che fa il cervello umano, infatti noi facciamo pruning delle soluzioni.

Basandoci sul seguente esempio è possibile notare che ad ogni regola si ha un numero associato che è proprio la probabilità di frequenza di quella regola. Per ogni regola uguale, la somma è uguale a 1.

| | | |
|-----------------------------------|-------|--|
| $S \rightarrow NP VP$ | [.80] | $Det \rightarrow that [.05] the [.80] a [.15]$ |
| $S \rightarrow Aux NP VP$ | [.15] | $Noun \rightarrow book [.10]$ |
| $S \rightarrow VP$ | [.05] | $Noun \rightarrow flights [.50]$ |
| $NP \rightarrow Det Nom$ | [.20] | $Noun \rightarrow meal [.40]$ |
| $NP \rightarrow Proper-Noun$ | [.35] | $Verb \rightarrow book [.30]$ |
| $NP \rightarrow Nom$ | [.05] | $Verb \rightarrow include [.30]$ |
| $NP \rightarrow Pronoun$ | [.40] | $Verb \rightarrow want [.40]$ |
| $Nom \rightarrow Noun$ | [.75] | $Aux \rightarrow can [.40]$ |
| $Nom \rightarrow Noun Nom$ | [.20] | $Aux \rightarrow does [.30]$ |
| $Nom \rightarrow Proper-Noun Nom$ | [.05] | $Aux \rightarrow do [.30]$ |
| $VP \rightarrow Verb$ | [.55] | $Proper-Noun \rightarrow TWA [.40]$ |
| $VP \rightarrow Verb NP$ | [.40] | $Proper-Noun \rightarrow Denver [.40]$ |
| $VP \rightarrow Verb NP NP$ | [.05] | $Pronoun \rightarrow you [.40] I [.60]$ |

$$A \rightarrow \beta [p] \quad p \in (0,1) \quad \Rightarrow \quad \triangle$$

$$\sum_{\beta} P(A \rightarrow \beta) = 1$$

Questo mi assicura di avere una vera distribuzione di probabilità sulla grammatica e sui miei alberi.

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

Il k oltre a rendere il parser lento, va ad aumentare l'ambiguità della mia grammatica.

Per trovare tali distribuzioni di probabilità è possibile contare dando un corpus e contando il numero di regole che compaiono e contando la distribuzione di esse nell'intero corpus. Ma come si calcolano le probabilità di un albero sintattico?

$$P(T, S) = \prod_{node \in T} P(rule(n)) \quad \text{La probabilità di un albero è il prodotto delle regole usate nelle sue derivazioni.}$$

Con $P(T, S)$ indichiamo la probabilità di un albero partendo da S . Si calcola andando a moltiplicare le

probabilità delle singole regole/nodi che sono presenti nell'albero.

L'annotazione di parole di solito è strutturata a forma di albero quindi prende il nome di Treebanks; il più famoso nella lingua inglese è il Penn TB formato a costituenti. Tali alberi di solito sono costruiti dai linguisti poiché si deve assegnare nel modo più corretto possibile, di solito magari in maniera semiautomatica, in cui si fa elaborare la macchina, con un parser, e poi con una fase di rifinitura da più persone che collaborano insieme per avere un corpus che sia il più giusto possibile.

I TB implicitamente definiscono delle regole di una grammatica CF. Andando a fare reverse engineering dall'albero è possibile ritrovare le regole che lo producono. Da un tree bank posso estrarre anche le probabilità oltre le regole. Le probabilità estratte sono quelle che costruiscono la giusta distribuzione di probabilità tra i miei alberi. $P(\alpha \rightarrow \beta | \alpha)$

Nelle scritture sottostante (Prima formula) vado a contare quante volte una certa regola (alfa --> beta) e lo divido per il conteggio delle regole che hanno la parte sx uguale alla precedente.

$$P(\alpha \rightarrow \beta | \alpha) = \frac{\text{Count}(\alpha \rightarrow \beta)}{\sum_{\gamma} \text{Count}(\alpha \rightarrow \gamma)} = \frac{\text{Count}(\alpha \rightarrow \beta)}{\text{Count}(\alpha)}$$

Andando a vedere una con probabilità, l'algoritmo è molto simile leggermente moltiplicato per rispettare delle minime probabilità e di restituire quindi solo gli alberi più probabili. La table questa volta è tridimensionale, quindi per ogni cella memorizza un dizionario che per ogni simbolo non terminale mi dice la probabilità associata ad un sottoalbero che va da i a j. Ad esempio nell'ultimo if c'è un <, si controlla che la nuova versione che stiamo controllando non sia più bassa di quella possibile al momento. In questo modo memorizzo solo i sottoalberi con probabilità più alta, poiché se c'era una probabilità più alta la sostituisco con quella vecchia che risulta essere più bassa. Quindi adesso non è possibile ottenere in una matrice più soluzioni.

function PROBABILISTIC-CKY(*words, grammar*) **returns** most probable parse
and its probability

```

for j ← from 1 to LENGTH(words) do
    for all { A | A → words[j] ∈ grammar }
        table[j - 1, j, A] ←  $P(A \rightarrow words[j])$ 
    for i ← from j - 2 down to 0 do
        for k ← i + 1 to j - 1 do
            for all { A | A → BC ∈ grammar,
                and table[i, k, B] > 0 and table[k, j, C] > 0 }
                if (table[i, j, A] <  $P(A \rightarrow BC) \times table[i, k, B] \times table[k, j, C]$ ) then
                    table[i, j, A] ←  $P(A \rightarrow BC) \times table[i, k, B] \times table[k, j, C]$ 
                    back[i, j, A] ← {k, B, C}
            return BUILD_TREE(back[1, LENGTH(words), S]), table[1, LENGTH(words), S]

```

N.B.: nel primo if, *table*[*i*, *j*, *A*] indica quanto è probabile la struttura dati indicata da *i,j,A*.

VALUTAZIONE

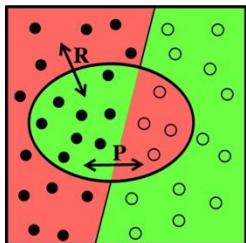


Per la valutazione ci affidiamo alla **Precision** e alla **Recall**. Queste due metriche ci indicano quanto la soluzione trovata è giusta e quanto della soluzione totale giusta viene ritrovata dal mio sistema rispettivamente.

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.

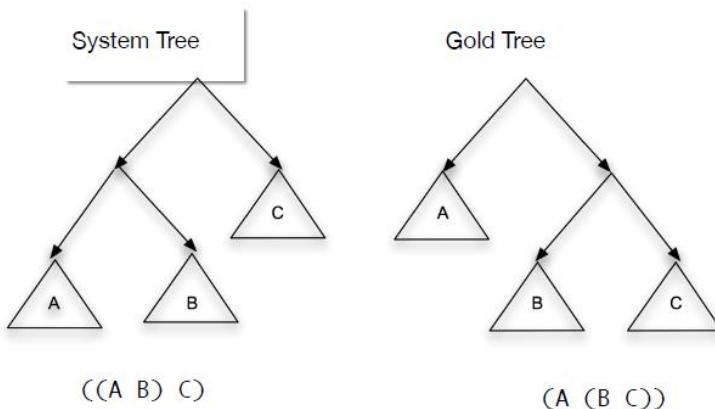
Precision: Quanti sottoalberi del system appartengono anche al gold.

Recall: Quanti subtree del gold appartengono al system.



$$P = \frac{T^\bullet}{T^\bullet + F^\bullet} \quad R = \frac{T^\bullet}{T^\bullet + F^\circ}$$

Definiamo come **system tree** quello prodotto dal parser e **gold tree** quello l'albero obiettivo cioè quello ideale.



In questo caso volendo calcolare la precisione e il recall avremo che:

- precisione = 3/5

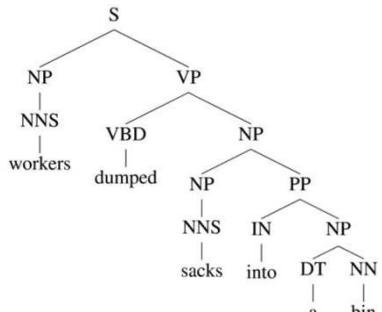
Ipotizzando di avere un'ulteriore diramazione nel gold tree alla radice, il valore di recall cambia

- recall = 3/7

(Lezione 24 marzo)

LEXICALIZED PCFG

È importante notare che però una parte delle probabilità viste nell'albero non dipendono direttamente dalle parole della frase, quindi non dipendono dalla parte lessicale ma dalla parte strutturale delle regole.



Ad esempio, il primo VP dipende da un altro VP e da un PP non direttamente da parole

come ad esempio il VBD che dipende da dumped.

Inoltre, come è possibile notare in questo esempio, abbiamo un caso in cui la stessa frase a due diverse letture ma la parte superiore rimane pressoché la stessa.

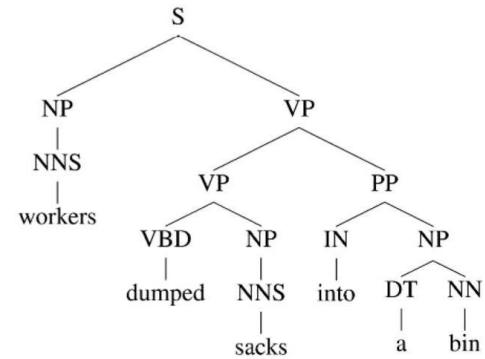
Non ci sono preferenze lessicale che influenzano la struttura totale dell'albero.

Si è provato a modificare leggermente queste PCFG attraverso un processo di Lessicazione ovvero si è cercato di propagare in alto dalle foglie alla radice le informazioni nelle parole.

Sono nate così le Lexicalized PCFG.

$A \rightarrow BC$

$A(\text{head}_A) \rightarrow B(\text{head}_B) C(\text{head}_C)$



Con questa scrittura vogliamo dire che andiamo ad inserire all'interno del simbolo A l'informazione di tipo lessicale che lo riguarda.

Ogni regola CF è potenziata con informazione circa le **heads** dei costituenti coinvolti. Punto di unione tra formalismi sintattici a costituenti e a dipendenza. Questa tecnica ci permette di distinguere in base alle parole il senso della frase, poiché ogni frase influenza sulla probabilità della regola stessa e quindi sulla scelta del giusto sottoalbero (regola). Ad esempio "mangio la pizza con la mozzarella" e "mangio la pizza con la forchetta" all'inizio ci dava lo stesso albero con la stessa probabilità attaccando l'ultima parola con l'NP pizza ma sappiamo benissimo che nel secondo caso la forchetta deve essere attaccata al VP mangio.

Un problema di questa tecnica è che la grammatica diventa enorme a causa delle possibili combinazioni da scrivere all'interno, e inoltre molte di queste regole potrebbero avere una probabilità così bassa da comparire raramente o addirittura mai. Questo è un problema di Sparsing e per risolvere bisogna utilizzare la tecnica di **Smoothing**, ovvero bisogna addolcire la distribuzione di probabilità per allontanare le probabilità dallo 0 e dall'1 per rendere tutto più equale. Questo è un concetto fondamentale di questa tecnica.

PARSING PARZIALE: CHUNK PARSING

Si ha una grammatica regolare di riferimento, quindi ancora più debole e che Chomsky disse non essere adeguata, inoltre abbiamo una tecnica bottom-up e una specie di programmazione dinamica (all-patch) e un oracolo a regole.

Il concetto fondamentale è il Chunk come base sintattica che a differenza dei costituenti non è ricorsivo.

Quando individuo i chunk vado anche ad individuare strutture che non sono ricorsive.

È come se avessi una nuova grammatica senza regole ricorsive. Un chunk non potendo avere ricorsione, rappresenta una struttura che non ha altre strutture al suo interno. Questo sia perché nessun simbolo di sinistra di una regola si rifa ad una regola ma a dei termini terminali e soprattutto perché si usa una grammatica regolare. Però noi sappiamo che il nostro linguaggio è ricorsivo quindi questa tecnica non è adeguata a rappresentare le frasi, però ci accontentiamo sapendo che in alcune tecniche non funziona. Però tale tecnica in alcuni casi è chiaramente più veloce, ha la complessità di un automa finito ovvero una complessità lineare.

Un ulteriore tecnica che si basa sempre sullo stesso tipo di parsing che si basa però su un oracolo probabilistico **IOB tagging** è una tecnica (statistica) di tagging, non proprio di Parsing, che utilizza i chunk e che permette di determinare l'inizio di un chunk con Begin e l'interno con Inside e dopo aggiungo un termine terminale Outside.

In questo modo definisco per ogni parola dei tag che ci indicano quando inizia la struttura e quando finisce.

Questa tecnica è utilizzata ad esempio per il riconoscimento della struttura del DNA.

Dati n chunk si avranno $2n+1$ tags che andranno a rappresentare la dimensione del corpus.

Nell'immagine a dx andiamo ad indicare quando un chunk comincia, quando ci troviamo al suo interno e al di fuori.

In questo modo andiamo a taggare una sequenza e implicitamente indichiamo un albero.

The morning flight from Denver has arrived
B_NP I_NP L_NP B_PP B_NP B_VP I_VP

The morning flight from Denver has arrived.
B_NP I_NP L_NP O B_NP O O

GRAMMATICHE E PARSER A DIPENDENZA

SINTASSI A DIPENDENZE

si basa sulle dipendenze sintattiche

Fino a 10-15 anni fa erano poco usate e rappresentavano un approccio diverso da quello normale. Si ha sempre una sequenza di parole solo che questa volta in output ho un albero in cui non ci sono i nodi non lessicali, infatti il numero di parole corrisponde al numero di nodi dell'albero collegati tra loro da delle frecce che esprimono le relazioni tra le parole, le quali sono rese esplicite dalla nuova struttura che andremo a vedere. L'approccio a dipendenze è detto verticale. È al momento il parsing più utilizzato.

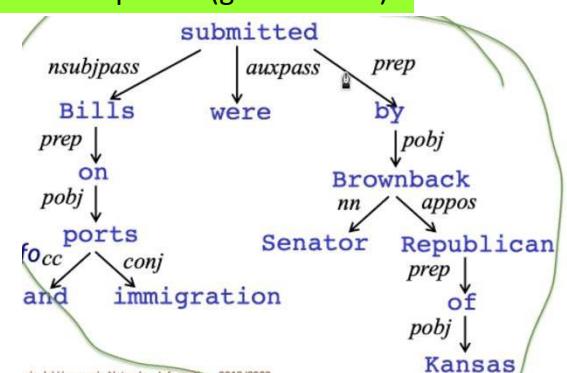
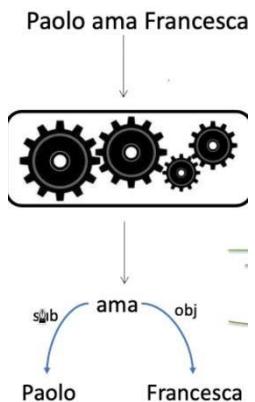
Tesniere, un linguista svizzero, ha per primo ipotizzato l'utilizzo di formalismi a dipendenze per analizzare il linguaggio naturale. Per lui la frase si organizza come un'unità formata da parole, quando viene analizzata, ogni parola non viene vista come singola senza vedere il resto della frase ma dipende da elementi superiori (governatore) e inferiori (subordinati). Tutte le parole insieme ai loro vicini si comportano come un'unità. Ad esempio, nella frase "Paolo corre" corre è il superiore e invece Paolo è inferiore. Ma come è possibile determinare chi sta sopra e chi sta sotto?

Attraverso dei test linguistici analizzati e segnati nel modo giusto da linguisti. Le relazioni sono binarie, come già accennato che quindi possono essere solo 2. Le dipendenze inoltre sono tipate, ovvero ogni arco ogni collegamento ha una descrizione che indica il collegamento tra le parole. Il numero di frecce in un albero è sempre $n - 1$ dove n è appunto il numero di parole.

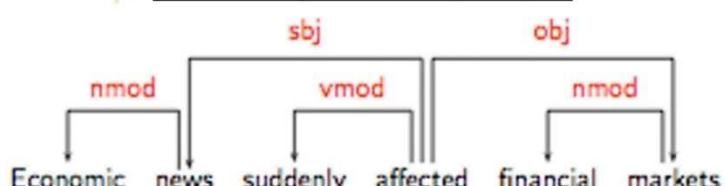
In questa grammatica è importante capire cos'è una testa, dove appunto una testa è un nodo che non ha governatori, e questo nodo determina la categoria sintattica dell'intera costruzione linguistica, inoltre la forma di un dipendente dipende appunto dalla testa ed inoltre la testa decide se le dipendenze devono esistere obbligatoriamente o no. Questi test sono delle prescrizioni che a volte funzionano e a volte no.

Alcuni test sono segnati nella tabella sottostante. Ci possono essere però dei casi in cui questi test tra di loro vanno in contrasto e quindi potrebbero creare dei problemi di simmetria nella risoluzione del parsing. Inoltre, un altro problema di questa tecnica è che in alcune frasi una relazione binaria non basta per distinguere le dipendenze, servirebbero ad esempio delle relazioni ternarie.

Errore: ultima riga è Noun e non Verb.
Si ha che la head è il nome in quanto la frase, affinché sia corretta, necessita del nome, ma può fare a meno dell'aggettivo financial.



| <u>Head</u> | <u>Dependent</u> |
|-------------|---------------------------|
| Verb | Subject (sbj) |
| Verb | Object (obj) |
| Verb | Adverbial (vmod) |
| Verb | Attribute (nmod) |





Questi formalismi grammaticali come sono descritti formalmente? Non esistono, esse sono una specie di schema di analisi sintattica che può essere più o meno formalizzato ma è molto utilizzato poiché permettono a tecniche di machine learning di imparare facilmente le probabilità da un corpus annotato senza una grammatica formale. Alcune lingue come il russo sono più semplici rappresentarli in strutture a dipendenze poiché l'ordine delle parole non è importante. In questo modo, soprattutto grazie al machine learning, si ottengono dei parsing molto veloci. Uno svantaggio è che l'assunzione di dipendenza non è sempre vera. Questa struttura a dipendenze inoltre è omogenea alla struttura semantica che viene dopo, infatti a volte viene vista direttamente come una struttura semantica. Un ulteriore vantaggio è il fatto che si può passare facilmente da una frase in una lingua in un'altra perché non si basa singolarmente sulla struttura della frase ma sulle dipendenze delle parole.

ALGORITMI DI PARSING prende una frase in ingresso e mi costruisce un albero a dipendenze in uscita

Dal punto di vista pratico, nell'applicazione in algoritmi di parsing, con le grammatiche a dipendenze si lavora attraverso schemi di annotazione utilizzati per il machine learning.

Le tecniche principali per il parsing a dipendenze sono 5:

- 1. Programmazione dinamica:** utilizzare algoritmi simili a quello di CKY ma il problema era che erano troppo complessi, fino a che un professore scoprì 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. Algoritmi a grafo: Basata sui grafi, utilizza un Minimum Spanning Tree, prima di valutare le frequenze di occorrenza delle parole in base ad un corpus (formato in un determinato modo) e in base a ciò si costruiscono gli archi fra le parole con associato il costo 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: ha come idea il fatto che prendendo un albero a costituenti qualsiasi e si cerca 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: scelte greedy per la creazione di dipendenze tra parole, guidate dal machine learning per la classificazione

5. Soddisfazione di vincoli (Constraint Satisfaction): Vengono eliminate tutte le dipendenze

PARSING DETERMINISTICO A TRANSIZIONI

Non si chiama deterministico per via del determinismo degli automi, ma è per il fatto che data una qualsiasi sequenza di parole si avrà sempre una soluzione ovvero un albero in uscita, a differenza del CKY ad esempio, in cui il parser può fermarsi non appena si rende conto che non è derivabile.

L'idea di questo parser è nata da un giapponese e poi ampliato da Nivre, un linguista informatico svedese, che ha creato precisamente l'algoritmo di parsing **MALT**.

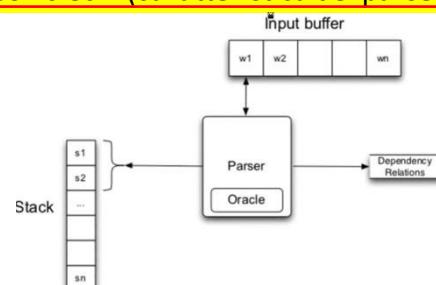
Tale algoritmo di parsing ha fondamentalmente una grammatica rappresentata da uno schema, che può essere visto come un automa, l'algoritmo è bottom-up e con organizzazione della memoria depth-first senza backtracking, noi avremo sempre una soluzione e mai vicoli ciechi (caratteristica del parser).

Oracolo probabilistico basato su machine learning basato sui

TB

Malt cerca di copiare la tecnica di Shift-Reduced della programmazione classica per il linguaggio naturale. Esso è formato fondamentalmente da un buffer di input una lista con la sequenza di parole della frase, dopo ci sarà un stack, ovvero una coda, che permette di memorizzare in ordine di coda le parole apprese dall'input da memorizzare, e infine restituisce un set di relazioni di dipendenza.

Da un punto di vista formale, la proiettività di un albero è la proprietà che gli archi non si incrocino. Sappiamo che in realtà gli alberi generati dal linguaggio naturale non sono proiettivi. Assumiamo che il nostro sistema produca solo alberi che siano proiettivi.



Le 3 strutture dati (stack, lista, insieme) contenute nel parser vengono volta per volta aggiornate, attraverso regole probabilistiche, in base ad alcuni fattori:

- input;
- contenuto dello stack;
- insieme delle dipendenze prodotte fino a quel momento.

Il concetto fondamentale è che le 3 strutture, ad ogni istante, definiscono uno **stato**;

inoltre, creano un numero di stati infinito (perché intendiamo tutte le parole di una

Stato iniziale

lingua messe in una sequenza corretta, ergo ∞) **difficile dal punto di vista**

• [[root], [sentence], ()]

computazionale da affrontare, a meno che non si adotti un approccio probabilistico.

Stato finale

Definiamo quindi i 2 stati fondamentali:

- **Iniziale**: composto da Stack vuoto (root), tutte le parole in ingresso (Sentence), insieme di tutte le relazioni binarie trovate fino a quel momento (inizialmente vuoto);
- **Finale**: composto da Stack vuoto (root), buffer vuoto (tutte le parole sono state analizzate), insieme delle relazioni sicuramente non vuoto (R).

• [[root], [], ()]

R alla fine sarà n-1, con n cardinalità di Sentence.

Data la frase esempio “**I booked a morning flight**”, avremo che rispettivamente stato iniziale e finale saranno **[[root], [I booked a morning flight], ()]** e **[[root], [], ((booked, I) (booked, flight) (flight, a) (flight, morning))]**. Nello stato finale l'insieme sarà valorizzato con le relazioni/dipendenze presenti nella frase. Tuttavia, come arriviamo dallo stato iniziale a quello finale?

Attraverso delle transizioni, vengono man mano modificati buffer e stack, inoltre viene compilato il set con tre principali operatori tra stati: **Shift**, **Left**, **Right**.

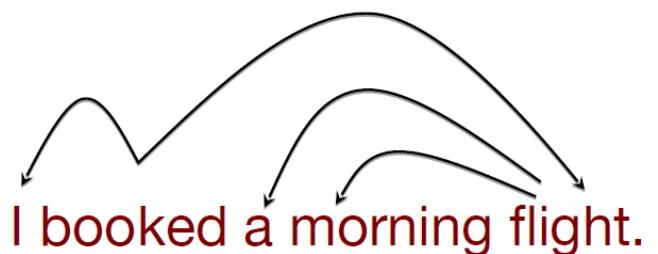
Lo **Shift** effettua una sola operazione: prende una nuova parola dal buffer eliminandola e la inserisce nella coda, il **Left** invece crea una dipendenza tra la prossima parola nella lista e la testa dello stack, subito dopo effettua una pop della coda eliminando l'elemento in testa. Il **Right** invece, crea una dipendenza tra l'elemento in testa alla coda e la prossima parola nella lista, dopo elimina la parola dalla lista ed elimina la parola dallo stack inserendola nella lista. I due operatori left e right sono **asimmetrici** perché il linguaggio umano è asimmetrico.

Chi mi permette di scegliere quali operatori attuare in ogni momento? L'oracolo probabilistico.

Nell'esempio in tabella è stato utilizzato un parser perfetto che fa le giuste scelte, ma non sempre è così. Infatti, l'algoritmo di parsing effettivamente è banale poiché tutto il lavoro è lasciato all'oracolo.



| | |
|-------|--|
| | {[root], [I booked a morning flight], []} |
| Shift | {[root, I], [booked a morning flight], []} |
| Left | {[root], [booked a morning flight], [(booked, I)]} |
| Shift | {[root, booked], [a morning flight], [(booked, I)]} |
| Shift | {[root, booked, a], [morning flight], [(booked, I)]} |
| Shift | {[root, booked, a, morning], [flight], [(booked, I)]} |
| Left | {[root, booked, a], [flight], [(booked, I), (flight, morning)]} |
| Left | {[root, booked], [flight], [(booked, I), (flight, morning), (flight, a)]} |
| Right | {[root], [booked], [(booked, I), (flight, morning), (flight, a), (booked, flight)]} |
| Right | {[], [root], [(booked, I), (flight, morning), (flight, a), (booked, flight), (root, booked)] } |



L'algoritmo, fin quando lo stato non è il finale, applica un operatore chiedendo all'oracolo, il quale darà una risposta su quale operatore applicare in base allo stato attuale. Il nuovo stato si ottiene applicando l'operatore al nuovo stato.

```
depParse(s)
    state = initialState(s)
    while state is not final
        op = oracle(state)
        state = apply op to state
    return relations.state()
```

Provare a fare la tabella di Paolo ama Francesca dolcemente.

A questo punto si presentano 2 problemi:

1. Viene prodotto un albero non tipato;
2. Capire i dettagli dell'oracolo per individuare quale operatore usare ad ogni passo.

Il parser visto finora produce un albero non tipato su ogni arco. Bisogna modificare leggermente questo algoritmo in modo tale che l'albero restituito in output sia tipato.

Per fare ciò semplicemente vado a tipare gli operatori, ovvero creo tanti operatori (Left e Right) per quanti sono le possibili tipazioni per ogni arco, date n dipendenze avremo $2n+1$ (1 è lo Shift che non viene tipato) operatori L e R. Saranno tipati solo L e R perché loro costruiscono le relazioni.

In questo modo le dipendenze contenute nel set di output avranno un'info in più, la tipologia di tipazione da assegnare a quel determinato albero.

| | |
|-----------|--|
| | {[root], [I booked a morning flight], []} |
| Shift | {[root, I], [booked a morning flight], []} |
| Left_Subj | {[root], [booked a morning flight], [(subj, booked, I)]} |

Questo complica ulteriormente il lavoro dell'oracolo. Come già detto, l'oracolo permette di decidere quale operatore scegliere in base alle parole contenute nelle strutture stack e buffer del parser. Supponendo di non voler utilizzare il machine learning, dovrei creare un'infinità di regole che permettono di indicare in base alle parole contenute in ogni istante quale operatore usare ma questo creerebbe un numero enorme di regole e dei conflitti tra di esse.

La soluzione è utilizzare un sistema statistico/probablistico, basato su machine learning, un vero e proprio classificatore che classifica un certo stato assegnandogli una certa etichetta, la quale applicata a tale stato ci porterà in un nuovo stato. La classificazione si effettua per ogni stato. Se il numero di stati è potenzialmente infinito, questo implica una distribuzione di probabilità infinita. Per questo il classificatore è **basato sulle features dello stato**, cosicché non lavora direttamente sugli stati, ma solo su delle caratteristiche/qualità misurabili che cercano di catturare la parte linguistica più importante contenuta in quello stato.

Il problema quindi è quello di apprendere la struttura.

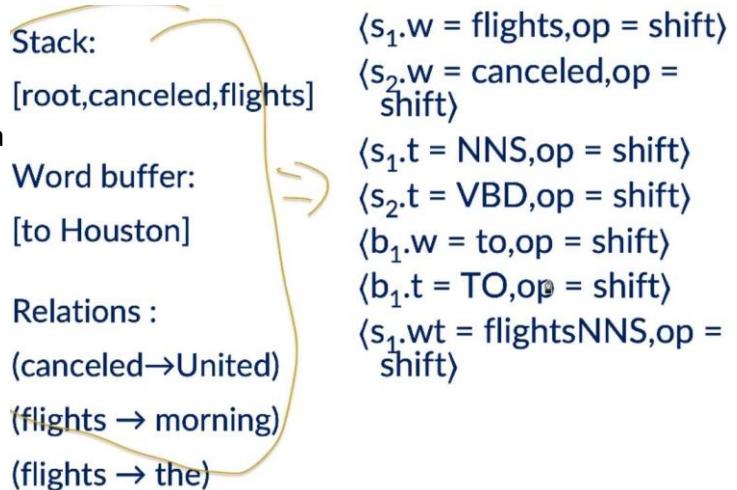
Quindi per un algoritmo di ML, bisogna prima di tutto capire 3 cose:

1. quali devono essere le **strutture delle features**, analizzate grazie ai linguisti che ci indicano quali sono le tipiche dipendenze.
2. Un altro punto importante è avere una **distribuzione di probabilità che si basa sulle feature appena trovate**, per fare ciò utilizzo o il TB a costituenti e lo converto in dipendenze utilizzandolo come training set, oppure vado a costruirmi una risorsa linguistica che permette di descrivere le dipendenze tra le varie parole in un insieme molto grande di frasi. L'esempio più importante di quest'ultimo modello è l'Universal Dependency TB che inoltre permette anche di funzionare su 70 lingue diverse. Permette quindi di rappresentare lingue molto diverse tra loro mediante circa 50 dipendenze.
3. Ultimo punto da capire è **poter apprendere le probabilità delle features trovate nel punto 1 e le associo all'insieme degli operatori**. Per fare ciò devo ricostruire al contrario (reverse engineering, simile al dump SQL) le feature che hanno portato all'utilizzo di quell'operatore all'interno del UDTB, così da ottenere per ogni features la probabilità associata ad ogni operatore. Questo ci permette di creare tante regole a partire dai possibili alberi del TB. Dagli alberi quindi dobbiamo avere le features + operatori che mi hanno portato a costruire l'albero stesso. La distribuzione di probabilità si calcola dall'insieme dei features+operatori che rappresenta l'insieme da cui apprendo, quello che apprendo è la distribuzione di probabilità.

```
# sent_id = isst_tanl-10
# text = Hamad Butt è morto nel 1994 a 32 anni.
1 Hamad Hamad PROPN SP   _      4      nsubj   _      -
2 Butt Butt PROPN SP   _      1      flat:name   _      -
3 è essere AUX VA   Mood=Ind|Number=Sing|Person=3|Tense=Pres|VerbForm=Fin 4
4 morto morire VERB V   Gender=Masc|Number=Sing|Tense=Past|VerbForm=Part 0
5-6 nel _ ADP _ E   _      7      case   _      -
7 in in _ ADP _ E   _      7      case   _      -
8 il il DET RD   Definite=Def|Gender=Masc|Number=Sing|PronType=Art 7
9 1994 1994 NUM N   NumType=Card 4      obl   _      -
10 a a ADP E   _      10     case   _      -
11 32 32 NUM N   NumType=Card 10     nummod _      -
12 anni anno NOUN S   Gender=Masc|Number=Plur 4      obl   _      -
13 . . PUNCT FS   _      4      punct _      -
14                                SpaceAfter=No
```

Ad esempio, in ogni particolare stato avrò un insieme di info assumibili e recuperabili come nella seguente immagine:

in base alle info a sinistra devo calcolare una scoring function che permette di capire a quale regola riferirsi. Quindi lo scopo è apprendere i pesi che massimizzano lo score della transizione corretta per tutte le configurazioni nel training set. Le configurazioni sono la derivazione parziali corrette dell'albero a dipendenze.



PARSING A REGOLE PER DIPENDENZE A VINCOLI

~~Questo parser è stato sviluppato a Torino negli ultimi 30 anni, dal prof Leonardo Nello, è il Turin University Parser.~~

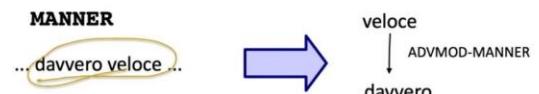
Anatomia:

- Competence: grammatica a dipendenze sottoforma di vincoli (non un automa finito probabilistico come il parser MALT) con regole scritte a mano;
- Algoritmo: circa bottom-up, organizzazione di memoria DFS;
- Oracolo: rule-based.

~~Questo sistema divide il parsing in 3 fasi:~~

~~1. Chunking: la frase in ingresso viene suddivisa in chunk, costruendo piccoli alberi a dipendenze applicando una determinata regola come questa per un Chunking nominale.~~

IF un avverbio di tipo **MANNER** (modo) precede immediatamente un aggettivo qualificativo **THEN** esso dipende da esso attraverso una dipendenza **ADVMOD-MANNER**

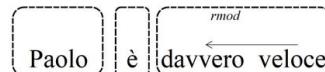


~~2. Coordination: l'importanza della coordinazione è data dal fatto che essa è una delle cose più complesse, infatti, l'ambiguità sintattica nasce anche dalla coordinazione, quindi analizzarla in uno step a sé stante ci permette di analizzare l'ambiguità delle congiunzioni con delle regole.~~

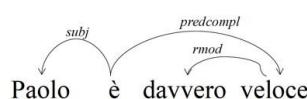
~~3. Verb sub categorization: una volta ottenuti i chunk NP questi vengono collegati ai verbi sfruttando una tassonomia/gerarchia di regole, ossia delle regole che hanno delle priorità su altre.~~

Paolo è davvero veloce

Chunking



Verb-SubCat



~~Domanda esame: differenza tra Chunk e sintagma: la differenza tra un chunk e un sintagma è che un chunk è un'unità di significato non è ricorsiva, quindi indica delle regole più semplici. Questo perché se dovessi individuare un noun phrase sarebbe più difficile in quanto bisognerebbe tenere conto della ricorsione.~~

SEMANTICA FORMALE E COMPUTAZIONALE

Mentre la sintassi è stata sempre affrontata dai linguisti, la semantica in quanto molto complicata è stata sempre snobbata.

Nella storia dell'analisi semantica troviamo SHRDLU e Chat-80. Nel vecchio secolo non c'era il machine learning quindi si scrivevano a mano le regole: entrambe le tecniche accennate sono state scritte a mano per la lingua inglese. SHRDLU è un computer che parla con l'umano eseguendo azioni nel mondo dei blocchi, quindi permette di spostare i blocchi e di interagire con l'umano per eseguire tali scopi. L'approccio è quello di trasformare le frasi create dall'umano in una rappresentazione semantica, basata sulla logica, che permette al computer di capire l'operazione da effettuare.

In Chat-80 invece, sempre a Stanford, costruiscono un database sulla geografia e utilizzano il linguaggio naturale per interrogarlo. Ad esempio, la domanda "Where is Rome?".

Entrambi questi approcci fanno riferimento alla logica per codificare il significato della frase in input, per questo vengono chiamati sistemi di semantica computazionale.

Il giusto compromesso tra espressività e potere computazionale, secondo Blackburn & Bos, è la logica del prim'ordine. Questo perché, oltre che rappresentare bene la logica dei verbi, permette di esprimere anche il determiner.

Anche però la logica del primo ordine ha dei limiti. Infatti, se si vuole rappresentare il significato di frasi incomplete, come la frase formata dalla singola parola "ama", non riesce a esprimere il significato completo con una regola $\text{love}(X, Y)$, perché la logica aspetta i due predicati.

Montague ha proposto un potenziamento della logica del primo ordine utilizzando una Lambda-FOL (First Order Logic), che ci permette di rappresentare parole e sintagmi mancanti. Questo formalismo diventa necessario per rappresentare i frammenti delle frasi, però il problema è che ha una potenza così ampia che è difficile effettuare ragionamento automatico, la cosa positiva è che però, alla fine, la frase sarà sempre una della forma della logica del primo ordine e quindi non avrà una complessità molto più elevata.

Utilizzare la logica del primo ordine, ci permette di effettuare inferenze a partire dalla base di dati e dalle informazioni codificate. Come ad esempio, con la frase "chiunque corre vive meglio" e so che Paolo corre, allora posso facilmente inferire che Paolo vive meglio. L'idea della logica del primo ordine nella linguistica è quella di associare a delle parole, relazioni di qualsiasi tipo.

- FOL simboli

- Costanti: **john, mary**
- Predicati & relazioni: **man, walks, loves**
- Variabili: **x, y**
- Connettivi Logici: $\wedge \wedge \neg \rightarrow$
- Quantificatori: $\forall \exists$
- Altri simboli ausiliari: **() ,**



Content Words

Function Words

- FOL formule

- Formule atomiche: $\text{loves(paolo,francesca)}$
- Formule composte: $\text{man(paolo)} \wedge \text{loves(paolo,francesca)}$
- Formule quantificate: $\exists x (\text{man}(x))$

Questa logica vuole catturare alcune parti del significato della mia frase, e magari lo cattura in maniera frammentata. Una parte importante che cattura è la semantica predato.

Grazie alla logica si può dire in maniera complementare le cose vere e le cose false nel mondo attraverso una struttura dati codificata come nel seguente esempio:



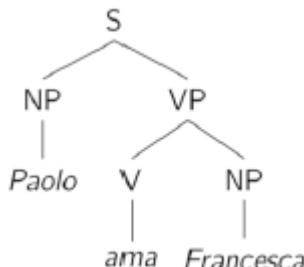
| Domain | $\mathcal{D} = \{a, b, c, d, e, f, g, h, i, j\}$ |
|-------------------------------------|---|
| Matthew, Franco, Katie and Caroline | a, b, c, d |
| Frasca, Med, Rio | e, f, g |
| Italian, Mexican, Eclectic | h, i, j |
| Properties | |
| Noisy | $\text{Noisy} = \{e, f, g\}$ |
| Frasca, Med, and Rio are noisy | |
| Relations | |
| Likes | $\text{Likes} = \{\langle a, f \rangle, \langle c, f \rangle, \langle c, g \rangle, \langle b, e \rangle, \langle d, f \rangle, \langle d, g \rangle\}$ |
| Matthew likes the Med | |
| Katie likes the Med and Rio | |
| Franco likes Frasca | |
| Caroline likes the Med and Rio | |
| Serves | $\text{Serves} = \{\langle e, j \rangle, \langle f, i \rangle, \langle e, h \rangle\}$ |
| Med serves eclectic | |
| Rio serves Mexican | |
| Frasca serves Italian | |

CS Foundamental Algorithm

1. Effettuare il parsing della frase per ottenere un albero sintattico a costituenti (fatto durante l'analisi sintattica);
2. Cercare la semantica di ogni parola nel lessico; 
3. Costruire, con un approccio bottom-up, la semantica per ogni sintagma in maniera ricorsiva seguendo l'albero (Syntax Driven) secondo il **principio composizionalità di Frege**: "Il significato del tutto è determinato dal significato delle parti e dalla maniera in cui sono combinate".

Questo equivale a mettere in pratica il principio di composizionalità di Frege ovvero Il significato del tutto è determinato dal significato delle parti e dalla maniera in cui sono combinate (ovvero la sintassi).

Considerando la frase "Paolo ama Francesca" ed ottenuto l'albero sintattico con un qualsiasi parsing, sappiamo adesso che il nostro significato in questo momento è estrarre un predicato amare(Paolo, Francesca) che ci indica il collegamento tra i due nomi e il verbo.



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

Essendo il verbo amare **asimmetrico**, ci aspettiamo che il VP venga mappato sul paziente, ma non sempre il paziente è il secondo a comparire nella frase (esempio: frasi passive). Lo scopo, per risolvere questo problema, è segnalare per ogni relazione chi è il paziente.

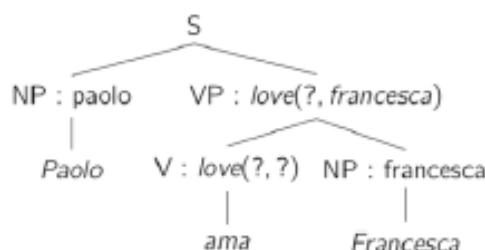
Ulteriori **problemi** sono:

- FoL non accetta variabili libere;
- rappresentare i frammenti della formula.

Per risolvere questi problemi abbiamo bisogno di una tecnica che ci permetta di indicare chi è il paziente nella frase e nel caso fosse assente di riempire in qualche modo.

"love(?, francesca)" oppure "love(francesca, ?)"? Come possiamo specificare come combinare le parti? Come rappresentare parti di formule?

Nell'immagine che segue abbiamo la composizione semantica della frase.





λ -calcolo

La tecnica adottata è quella della λ -FoL.

Attraverso il nuovo simbolo meta-logico λ e la **Lambda abstraction**, posso indicare/segnare, in un predicato anonimo, a variabili che non hanno ancora un argomento, l'informazione mancante.

Non appena si trova l'argomento che andrebbe a sostituire un lambda, si applica la **function application**.

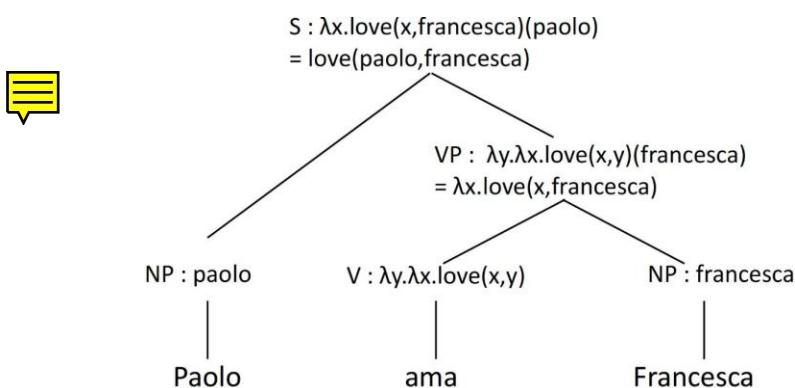
Questo permette di applicare la **Beta Reduction**. Tale operazione viene effettuata mediante tre

passaggi distinti. La beta reduction consiste in una serie di passi elementari che mi permette di fare la function application

Partendo da un qualcosa del tipo $(\lambda x.\text{love}(x; \text{mary}))(john)$:

1. Eliminiamo il λ e otteniamo $(\text{love}(x; \text{mary}))(john)$
2. Rimuoviamo l'argomento e otteniamo $\text{love}(x; \text{mary})$
3. Rimpiazziamo le occorrenze della variabile legata dal λ con l'argomento in tutta la formula ottenendo $\text{love}(\text{john}; \text{mary})$

Allora possiamo provare a riprendere l'esempio di prima e vedere come risolverlo correttamente. Con questa nuova tecnica il nostro albero viene rappresentato diversamente:



Notiamo che in questo caso non abbiamo più "love(?, ?)", ma ha due argomenti x e y, che devono essere scritti per completare il predicato. Questo deriva dal fatto che utilizziamo una logica superiore a quella del primo ordine. Un aspetto importante da considerare è che anche se i livelli intermedi seguono le regole del λ -calcolo, nella radice troveremo sempre un elemento della logica del primo ordine, che è anche quello che ci interessa di più perché illustra il significato della frase.

Possiamo notare che a livello VP elimina il λy sostituendo Francesca ad y (Beta-reduction) e allo stesso modo Paolo viene sostituito a λx .

L'**ordine delle lambda** è fondamentale, poiché permette di risolvere dall'esterno verso l'interno, quindi 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 rispetto al fatto che la funzione è a destra o a sinistra?

Nell'**approccio di Montague** va scritto nelle regole di composizione, ovvero per un VP, la funzione viene da sinistra e il valore invece da destra.

Regole lessicali

NP : paolo \rightarrow Paolo

NP : francesca \rightarrow Francesca

V : $\lambda y. \lambda x. \text{love}(x, y) \rightarrow$ ama

Regole di composizione

VP : f(a) \rightarrow V : f NP : a

S : f(a) \rightarrow NP : a VP : f

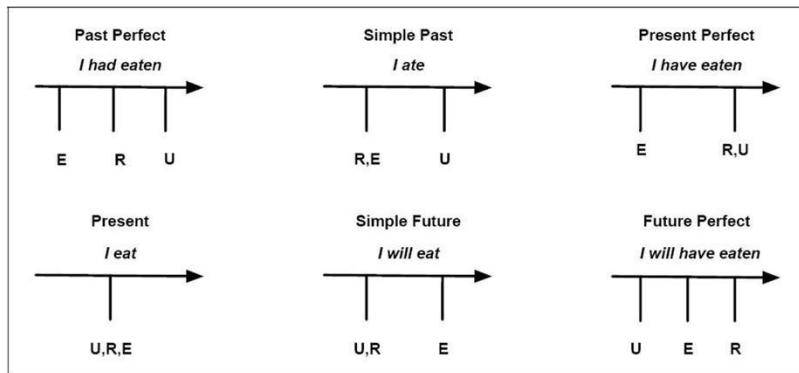
Le regole di composizione funzionano in modo analogo, ma aggiungono le funzioni. Ad esempio, la prima regola indica che la semantica di un VP (indicata come $f(a)$) si ottiene mediante l'applicazione della semantica del verbo (indicata da f) sulla semantica del noun phrase (indicata come a).

Le grammatiche di questo tipo sono dette **augmented CFG** o grammatiche ad attributi. La semantica è inserita nella grammatica.

N.B.: la "parola d'ordine" rimane sempre **sistematicità**, cioè le regole non devono essere valide solo per alcuni casi, ma devono funzionare sempre.

Mentre nella sintassi, la struttura sintattica specificava completamente le relazioni sintattiche nella frase, ciò non è vero per il livello semantico, ovvero la struttura semantica, da sola, non rappresenta tutto il significato che c'è nella frase principale, ma solo la struttura predicato-argomento.

Quindi si considera la semantica di una frase argomentativa in forma di domanda chi sono i predici e quali gli argomenti della frase??



Anche il tempo del verbo gioca un ruolo importante nella semantica, e ha bisogno di una gestione a parte, quindi non basta una struttura predicato-argomento, ma ne serve anche una temporale (immagine). Per gestire i tempi verbali ogni verbo viene rappresentato da tre tempi:

- **Utterance time:** quando è stata detta la frase;
- **Event time:** quando l'evento legato al verbo accade;
- **Reference time:** tempo di riferimento base definito secondo specifici criteri.

Inoltre, ci sono anche altri aspetti che possono essere fondamentali per specifiche forme di frase

COMPUTATIONAL SEMANTIC

Supponiamo di voler analizzare la frase “Un uomo ama Francesca”: come prima idea ci viene quella di utilizzare l’approccio degli operatori esistenziali, essendo nella FoL, che ci permette di definire l’esistenza di un uomo, ovvero un’entità del mondo di cui non conosco il nome, che si lega con il verbo alla persona Francesca.

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 “Un uomo”, infatti, significherebbe più “esiste un uomo” o “c’è un uomo” vista come una frase dichiarativa.

Per risolvere ciò, l’**idea** è di partire dalla fine anziché dall’inizio (reverse engineering).

Reverse engineering

Questo tipo di semantica è migliore e risulta ben formata rispetto alla FoL. Partendo dal finale, cerco di capire come devono essere fatti tutti i componenti affinché sia giusto.

Per ottenere la formula $\exists z (\text{man}(z) \wedge \text{love}(z, \text{francesca}))$ l’ultimo passo di beta-reduction deve essere stato applicato su una formula del genere: $\exists z (\lambda y. \text{man}(y)(z) \wedge \lambda x. \text{love}(x, \text{francesca})(z))$.

Per ottenere ciò dobbiamo sfruttare l’**astrazione sui predicati**:

$$(\lambda Q. \exists z (\lambda y. \text{man}(y)(z) \wedge Q(z))) (\lambda x. \text{love}(x, \text{francesca}))$$

Abbiamo effettuato un’astrazione su un predicato Q che andiamo a rimpiazzare con $\lambda x. \text{love}(x, \text{francesca})$ tramite beta-reduction ottenendo la formula $\exists z (\text{man}(z) \wedge \text{love}(z, \text{francesca}))$.

L’astrazione procede su **man** ottenendo:

$$(\lambda P. \lambda Q. \exists z (P(z) \wedge Q(z))) (\lambda x. \text{love}(x, \text{francesca})) (\lambda y. \text{man}(y))$$

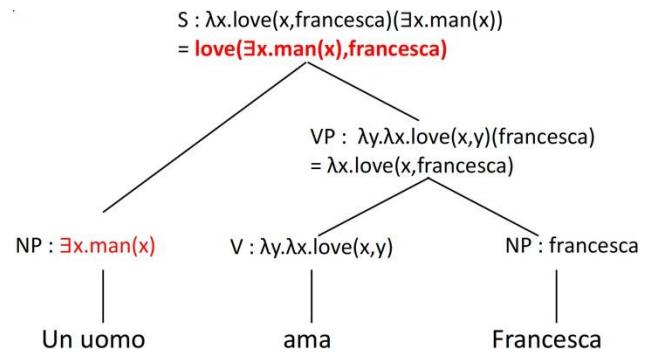
In pratica abbiamo astratto i predicati love e man mediante P e Q in modo che le operazioni di beta-reduction portino alla formula desiderata. Di fatto dopo una serie di passaggi come questo andiamo ad ottenere la semantica dell’articolo (“un”), che è:

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

Applicando lo stesso principio possiamo trovare la semantica per **tutti e nessun**:

$$\text{DT} : \lambda P. \lambda Q. \forall z (P(z) \rightarrow Q(z)) \rightarrow \text{tutti}$$

$$\text{DT} : \lambda P. \lambda Q. \forall z (P(z) \neg Q(z)) \rightarrow \text{nessun}$$



*L’uomo saggio inizia dalla fine e finisce col principio
(cit. ?)*

Vorremmo:

$$\text{Un uomo ama Francesca} \Rightarrow$$

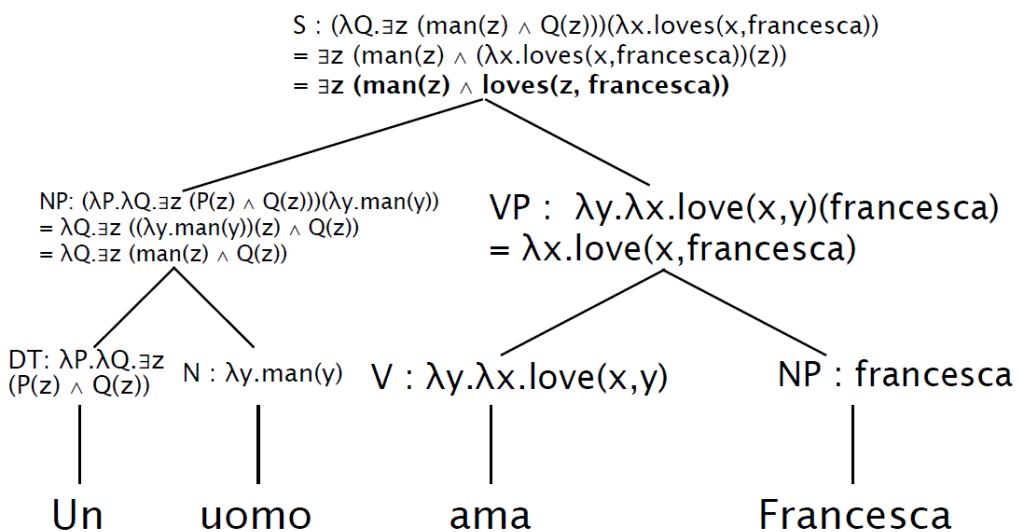
$$\exists z (\text{man}(z) \wedge \text{love}(z, \text{francesca}))$$

$$\exists z (\lambda y. \text{man}(y)(z) \wedge \lambda x. \text{love}(x, \text{francesca})(z))$$

Notiamo come la semantica dell'articolo è assai più complessa di quella dei verbi o dei nomi. Questo è dovuto al fatto che nomi e verbi sono vocaboli "di contenuto" e dunque si spiegano abbastanza da soli indipendentemente dal contesto. Questo, invece, non si può dire per gli articoli invece, che dunque hanno una semantica più complicata.

L'ultimo passo da fare è di cambiare la semantica di composizione facendo diventare NP una funzione. A questo punto possiamo gestire con la nuova semantica la frase "un uomo ama Francesca". Mostriamo l'albero in Figura. Notiamo che otteniamo delle formule abbastanza lunghe ma, ad ogni modo uno dei vantaggi della semantica è la sistematicità della stessa. Non importa quanto complicata possa essere la semantica di un nuovo termine da aggiungere.

Se procediamo al contrario come abbiamo fatto poco fa, possiamo sempre estenderla.



Il nostro nuovo lessico risulterà essere così modificato (\rightarrow):
 Questa mia modifica però non mi permette più il funzionamento per la frase “Paolo ama Francesca” poiché abbiamo cambiato la posizione della funzione e dell’articolo.

Noi però vogliamo **sistematicità**, ovvero che il nostro modello funzioni su tutte le possibili frasi, per poter aggiustare tale problema, devo modificare la forma lessicale di Paolo.

Lessico

DT : $\lambda P. \lambda Q. \exists z (P(z) \wedge Q(z)) \rightarrow \text{un}$

N : $\lambda y. \text{man}(y) \rightarrow \text{uomo}$

NP : francesca \rightarrow Francesca

V : $\lambda y. \lambda x. \text{love}(x, y) \rightarrow \text{ama}$

Regole di composizione

VP : f(a) \rightarrow V : f NP : a

S : f(a) \rightarrow NP : f VP : a

Si effettua il **Type-raising**: scriveremo che Paolo, un nome proprio, è una funzione che cerca un’altra funzione per diventare argomento (è un argomento che sta cercando un predicato per essere argomento), quindi la il nuovo lessico comparirà come nel seguente modo (\rightarrow):

Nella regola di composizione per S l’uso di f ed a è invertito: la semantica dell’NP è funzione, mentre la semantica del VP è argomento; prima era l’opposto.

Tuttavia, anche modificando la semantica di Francesca non si risolve il problema.

Occorre effettuare ancora Type-raising sul verbo transitivo “ama”.

Esiste ancora un ulteriore problema, in riferimento al verbo transitivo, poiché non trova più gli argomenti come prima. Ama nel nostro esempio si modifica, ancora per type-raising.

Lessico

DT : $\lambda P. \lambda Q. \exists z (P(z) \wedge Q(z)) \rightarrow \text{un}$

N : $\lambda y. \text{man}(y) \rightarrow \text{uomo}$

NP : $\lambda P. P(\text{paolo}) \rightarrow \text{Paolo}$

NP : $\lambda P. P(\text{francesca}) \rightarrow \text{Francesca}$

V : $\lambda y. \lambda x. \text{love}(x, y) \rightarrow \text{ama}$

Regole di composizione

VP : f(a) \rightarrow V : f NP : a

S : f(a) \rightarrow NP : f VP : a

$$\text{ama} \leftarrow V : \lambda R. \lambda x. R(\lambda y. \text{love}(x, y))$$

Possibile domanda d’esame!!

Quindi la grammatica risulta essere la seguente:

| | | |
|-------------|-------|---|
| nomi comuni | uomo | $\lambda x. \text{man}(x)$ |
| nomi propri | Paolo | $\lambda P. P(\text{Paolo})$ |
| verbi intr. | corre | $\lambda x. \text{run}(x)$ |
| verbi tr. | ama | $\lambda R. \lambda x. R(\lambda y. \text{love}(x, y))$ |
| articoli | un | $\lambda P. \lambda Q. \exists z (P(z) \wedge Q(z))$ |

Punti chiave

- extra λ per gli NP
- astrazione sui predicati
- inversione di controllo: NP_subj come funzione e VP come argomento

Sistematicità

Queste sono le rappresentazioni semantiche fondamentali di questi oggetti linguistici che mi permettono di rappresentare correttamente la semantica delle 2 frasi in questione.

Ci permette inoltre di ottenere **sistematicità**. RIVEDI LEZIONE 31/03

Problema: scope ambiguity



Consideriamo la frase "In questo paese, ogni 15 minuti una donna mette al mondo un bambino. È nostro compito trovare quella donna e fermarla" [Groucho Marx].

Ovviamente la donna a cui facciamo riferimento non è sempre la stessa, ma per come abbiamo definito la nostra semantica non possiamo cogliere entrambi i significati.

La frase "Ogni uomo ama una donna" può avere due interpretazioni: ogni uomo ama una donna, ma non necessariamente la stessa oppure tutti gli uomini amano la stessa donna. L'ambiguità è semantica.

Per risolvere il problema utilizziamo un oracolo che rappresenta la **pragmatica** per scegliere quali regole applicare.

Avverbi

Per rendere le cose più semplici si può prima di tutto utilizzare una diversa struttura utilizzando una **CCG (Combinatorial Categorial Grammar)** che permette una facile interfaccia tra sintassi e semantica. A differenza delle context-free è più semplice collegare la sintattica alla semantica. Inoltre, cambiano anche la rappresentazione dei predicati (ad esempio *in* ama dolcemente, dolcemente è un modificatore), ma questo indica l'utilizzo di una logica del secondo ordine che però non va bene poiché non ci permette più di eseguire l'inferenza, oltre alla complessità. Utilizzando invece la logica del primo ordine invece, potremmo aggiungere come parametro il modificatore.

Problema: se ne ho più di uno? (domanda esame)

La soluzione è la reificazione dell'evento ovvero definire una variabile che descrive l'evento e utilizzare poi dei predicati sull'evento che modificano.

Quindi risulta:

$$\exists e \text{ love}(e, P, F) \wedge \text{sweetly}(e)$$

sweetly(e) rappresenta che l'evento e avviene con dolcezza. Restiamo nella logica del primo ordine.

È possibile generalizzare quanto visto in precedenza per scrivere anche gli argomenti, cioè gli elementi fondamentali del predicato (Parson).

$$\exists e \text{ love}(e) \wedge \text{agente}(e, P) \wedge \text{patient}(e, F) \wedge \text{sweetly}(e)$$

Agente = Paolo, Paziente = Francesca.

Il vantaggio di tale rappresentazione (detta **neo Davidsoniana**) è che è possibile generalizzare il fatto che alcuni predicati, anche se transitivi o intransitivi, sono uguali dal punto di vista semantico.

Questo mi permette di descrivere meglio semanticamente, anche nel caso in cui ad esempio non c'è il paziente in una frase. Come ad esempio la frase "Paolo ama" come risposta magari ad una domanda, con la logica precedente non potremmo descriverlo.

NLTK

Libreria in python che permette di effettuare tutte le tecniche viste fin'ora.

Questo è un esempio della grammatica sintattica e semantica per la frase Paolo ama Francesca.

```
## Natural Language Toolkit: A simple example for Italian
## Author: Alessandro Mazzei <mazzei@di.unito.it>
## To see more complex example -> simple-sem.fcfg
% start S
#####
# Grammar Rules
#####
S[SEM = <?subj(?vp)>] -> NP[SEM=?subj] VP[SEM=?vp]
NP[SEM=?np] -> PropN[SEM=?np]
VP[SEM=<?v(?obj)>] -> V[SEM=?v] NP[SEM=?obj]
#####
# Lexical Rules
#####
PropN[SEM=<\P.P(paolo)>] -> 'Paolo'
PropN[SEM=<\P.P(francesca)>] -> 'Francesca'
V[SEM=<\X x.X(\y.love(x,y))>] -> 'ama'
```

L'idea è quella di scrivere la grammatica e lasciare al sistema la complessità e l'elaborazione.

ALTRI APPROCCI ALLA SEMANTICA

Fino ad ora abbiamo visto l'approccio più sensato alla semantica computazionale. Rinunciamo ad avere la ricorsione, cosa che permette di rappresentare la logica. Tali approcci sono meno formali. Vediamo 3 diverse tecniche che hanno delle semplificazioni rispetto alle precedenti:

1. **ARM**, non abbiamo più una FoL;
2. **Filler-shot**, rinunciamo alla ricorsione;
3. **Semantic Grammars**: rinuncio alla sintassi.

La **ARM** è una semantica formale non più formale, in cui non c'è la semantica degli articoli, si rinuncia alla rappresentazione della semantica di alcuni oggetti, che ci permettono di concentrarci di maggiormente su oggetti più semplici. L'**idea** è quella di costruire un corpus in cui elimino i quantificatori gli articoli ecc, perché voglio una parte meno complessa i quella della FoL. Quindi avrò una strutturazione molto più semplice delle frasi, e per questo un linguista può scrivere molto più velocemente dei corpus permettendo **poi di applicare machine learning e lavorare di più sulla statistica**. È chiaro che in alcuni casi si perde il significato delle frasi ma rende molto più veloce e meno complessa l'elaborazione e si permette l'utilizzo del ML. Si accelera questo processo, perdendo alcuni significati delle frasi.

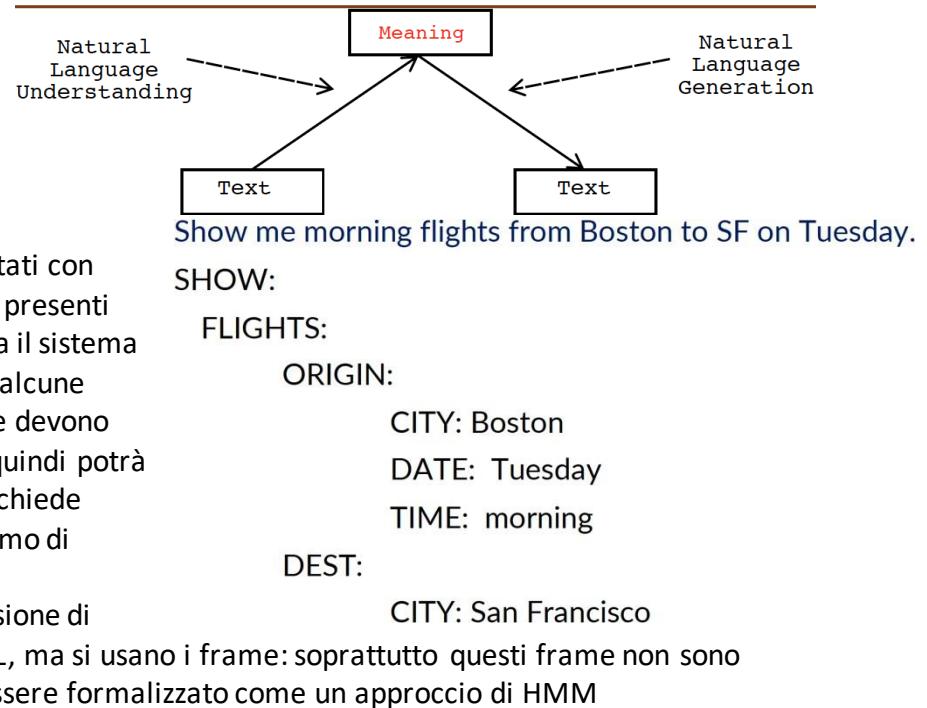
FRAME AND SLOT SEMANTIC

Alexa è un esempio di questa tecnica. Non si usa una logica ma una **Filler/Slot semantic**. Non si ha più una logica che presenta il significato ma si hanno dei frame che devono essere riempiti. Si ha così un formulario in cui ho dei buchi che vanno riempiti.

Si capisce innanzitutto l'**intent** all'interno della frase e subito dopo si utilizzano un insieme di data-structures (frames) con dei buchi, che devono essere completati con determinate regole, con le **entity** presenti all'interno della frase o addirittura il sistema può rendersi conto che mancano alcune entity per completare il frame che devono essere completeate per forza che quindi potrà chiedermi (ad esempio siri che ci chiede l'orario della sveglia se gli chiediamo di svegliarci il giorno dopo).

La semplificazione rispetto alla visione di

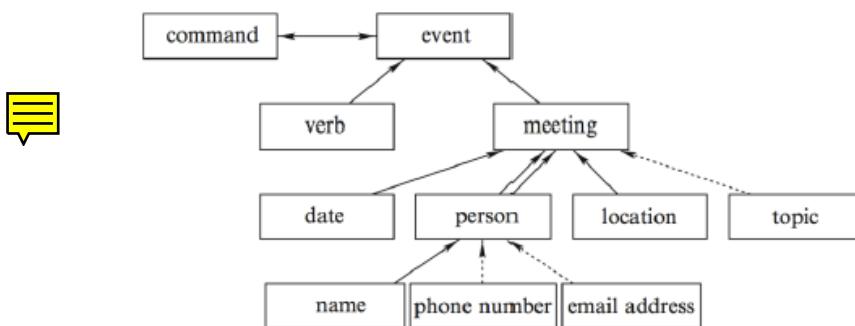
Montague è che non c'è più la FoL, ma si usano i frame: soprattutto questi frame non sono ricorsivi. Questo approccio può essere formalizzato come un approccio di HMM



Per esame: differenza tra semantica Dialog Flow e Montague

Siri

È un multi goal-oriented system. Ogni task avrà un frame diverso con slot che devono essere riempiti. Alla base abbiamo una sorta di ontologia formata dagli slot, alcuni di questi sono mandatori. L'evento "fissare un meeting" ha una data, un luogo, un topic ecc. Siri inizia a riempire gli slot mandatori e se uno di questi slot è vuoto, ad esempio la data, chiede all'utente la data dell'appuntamento.



SEMANTIC GRAMMARS

In questo approccio utilizziamo delle grammatiche in cui rinunciamo alla sintassi. I non-terminali non sono sintagmi della sintassi, ma sono rappresentati tramite variabili semantiche. La testa della regola contiene la semantica:

DEPARTTIME → (after|around|before)

L'approccio non segue l'ipotesi di composizionalità di Fregue in quanto non abbiamo la sintassi.

Domanda esame: Qual è l'approccio alternativo alla semantica formale e compositiva? Frame and Slot semantics!

NATURAL LANGUAGE GENERATION

Una possibile risposta, dopo aver intuito il significato di una frase, oltre che un'azione specifica potrebbe necessitare di una risposta testuale. Per fare ciò si parla di Natural Language Generation. Tale tecnica è vista come la tecnica opposta a quelle viste finora. È il processo che permette di comunicare qualcosa in linguaggio naturale, questo qualcosa può essere anche ad esempio un grafico di crescita in cui si vuole una descrizione o, come abbiamo visto in RNDL, per descrivere foto. Lo **scopo** generale dunque è quello di creare un sistema automatico per produrre testo in linguaggio naturale.

Abbiamo in **input** una forma di rappresentazione dell'informazione (tabella SQL o FoL). In **output** vogliamo fornire documenti, spiegazioni, messaggi ecc.

Affinché questo sia realizzabile serve una conoscenza, prima di tutto, del linguaggio (grammatica, sintassi, pragmatica linguistica), ma anche del **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.

La NLG ha sempre avuto meno rilevanza: il perché si trova nel fatto che “Nella maggior parte dei casi un insieme prefissato di frasi può essere sufficiente per produrre risposte comprensibili e naturali. Per definizione, l'input della generazione è non ambiguo. La difficoltà maggiore dell'interpretazione viene a cadere. La generazione, essendo più semplice, è quindi meno interessante dal punto di vista scientifico.”.

Mentre una frase può avere più significati (**ambiguità**), al contrario, scegliendo il significato, non c'è ambiguità e quindi diventa leggermente più semplice e per questo meno interessante dal punto di vista scientifico. Tutto questo è vero, ma potrebbe capitare che il mio sistema con diversi significati restituisce la stessa frase in output, questa è una performance scarsa del mio 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 per farlo.

Solitamente possiamo o utilizzare dei **template** (ad esempio il comando di printf in C è un template) oppure possiamo costruire un vero e proprio sistema di NLG. I **vantaggi** dei template sono che sono facili e veloci da gestire, c'è poca formalizzazione e non facciamo uso di linguistica. Inoltre, la tecnica dei template prestabiliti, ci darebbe come output ripetizioni di frasi simili e ripetitive anziché utilizzare la forma elenco come farebbe l'uomo (ad esempio ho visto L, ho visto M, ho visto P anziché ho visto L, M e P).

I **vantaggi** dell'**NLG** rispetto ai Template sono molteplici, poiché nella seconda tecnica avrei una formalizzazione limitata (printf(ho visto %f)) che non permette effettivamente la linguistica, sicuramente risulta essere più veloce, ma la NLG è più mantenibile e ha una qualità testuale molto superiore con delle strutture linguistiche che permettono di creare frasi ad hoc, sicuramente più lento. Nella storia ci sono stati molti approcci ibridi.

Esempio: un programma di NLG abbastanza noto è "**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 vengono prodotti i documenti. Per fare ciò inizialmente 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: è inutile fornire ai genitori un testo con linguaggio medico specifico in quanto non sarebbero in grado di comprenderlo.

Spesso i sistemi di NLG vengono valutati con un questionario posto ad un utente e di solito in base anche a specifici task. Inoltre, i dati dei sensori e le azioni conseguenti sono stati mostrati a degli studenti di medicina, i quali ne hanno valutato la genuinità.

L'NLG sembra, in qualche modo un'alternativa all'infografica (grafico che ha lo scopo di informare in modo utile) poiché la seconda funziona meglio rispetto all'NLG se una persona è esperta a leggere i dati attraverso la stessa, per utenti meno esperti il linguaggio naturale sarebbe più ottimale.

N.B.: NLG si avvicina di più al modo in cui gli uomini pensano. Questo per informare, ma per spiegare in un linguaggio grafico spesso è impossibile rispetto invece al testo che per spiegare è naturale. Questo è il valore aggiunto dell'NLG.

Negli ultimi anni grazie agli studi sui big data c'è un maggiore fermento nel campo dell'NLG perché a fronte di tutti questi dati, dobbiamo anche essere in grado di spiegarli. Ad esempio, un'applicazione potrebbe essere di far comprendere come mai una rete neurale ha preso una certa decisione, visto che spesso questo tipo di tecnologie da una risposta ma senza spiegarne i motivi.

Una possibile suddivisione dei **task** intrapresi dal **NLG** è la seguente (si va dal contenuto al testo, 1 → 7): (Considerando un task molto semplice come quello dei treni (treno in transito sul binario.... Treno in partenza dal binario...))

1. **Content determination:** al primo livello siamo molto vicini al significato e lontani dal linguaggio. Decido cosa dire, il messaggio che voglio esprimere, delle capsule di significato rappresentate nel mio database, ad esempio quello ferroviario, da un record. Tale significato viene espresso mediante messaggi, i quali sono aggregazioni di dati che possono essere espressi linguisticamente, con una parola o un sintagma; sono basati su entità, concetti e relazioni sul dominio. Crea una struttura dati, come fosse il risultato di una query ed esprime i messaggi che io voglio comunicare. **Es:** treno, destinazione e orario di partenza.
2. **Discourse planning:** in questa fase indico che le informazioni da comunicare non sono un set di dati. Dobbiamo anche **definire la struttura dell'output**. Crea una relazione tra i messaggi scelti al passo precedente. Crea una struttura che relaziona tra di loro le frasi. Le relazioni possono essere:
 - **Concettuali:** le due frasi sono affiancate poiché parlano dello stesso argomento, ma non c'è un legame fra loro. **Es:** Il prossimo treno è il treno A e parte alle 10.
 - **Retoriche:** una delle due frasi è elaborazione dell'altra (struttura retorica). **Es:** Il treno A parte alle 10 ed è uno dei 20 treni per GlasgowCi dice in che relazione si trovano i messaggi scelti al passo 1. Serve per dare una struttura al content.
3. **Sentence aggregation:** ha già a che fare con il linguaggio, infatti 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 naturali. Decide il mapping tra i messaggi e il numero di frasi da generare. **Es:** Alessandro corre. Alessandro mangia → Alessandro corre e mangia.
La seconda frase ad esempio sembra più creata da un umano.
4. **Lexicalization:** permette di decidere quali parole utilizzare per esprimere i concetti del dominio e quali strutture sintattiche utilizzare per collegare le varie parole. Nella scelta delle parole, ad esempio nel contesto dei treni, è importante decidere il verbo "partire o

decollare". Se poi come obiettivo ho frasi multilingua, bisogna considerare anche i **lexical gap**: concetti unici di una lingua ad esempio "marca da bollo" è un concetto italiano. Nell'ambito di sintassi ad esempio decido se utilizzare una frase attiva o una passiva (punto di vista) e quindi più in generale la sintassi di generazione. DIPENDE dalla lingua scelta, e ha a che fare con le entità (areo o aeroplano?).

5. **Referring expression generation:** in questa fase decidiamo come riferirci ai **nomi propri**. Ha a che fare con le istanze. Generare le referring expressions determina quali parole usare per esprimere le entità del dominio in una maniera comprensibile all'utente. È infatti il momento in cui dobbiamo cominciare a riferirci non solo alle classi ma anche alle istanze delle stesse (nomi per l'appunto). Ad 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.

Domanda esame: differenza tra Lexicalization e referring expression generation.

Differenza Sentence Plan/Dependency Tree:

La prima è tipica della generazione, la seconda dell'analisi. (vedere bene, video lag)

Nel DT ci sono i token, nel SP ci sono i lemmi.

Rappresentano processi diversi: nel DT passo dalla frase al suo albero sintattico, nel SP passo dall'albero che contiene info sintattiche alla frase.

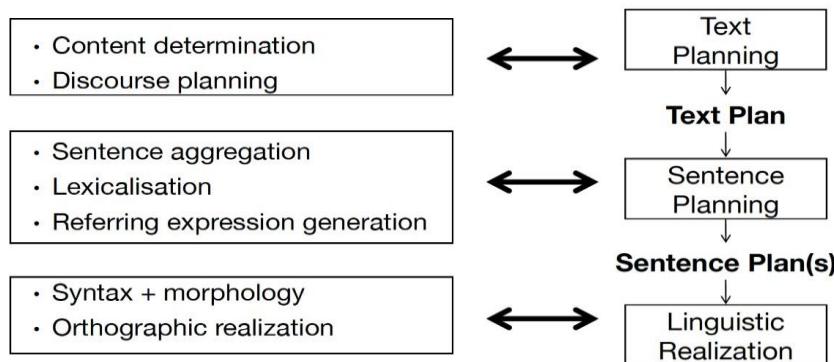
Nel SP non c'è un ordine delle parole, cosa che nel DT è implicita.

Inoltre, nel **DT** ci sono i verbi coniugati.

6. **Syntactic and morphological realization:** rispettando morfologia (come si formano le parole) e sintassi (come si formano le frasi) in base ad una specifica lingua. Quindi prima di tutto dal significato si deve passare attraverso la sintassi e subito dopo bisogna, attraverso la morfologia, flettere le parole (es: amare (Paolo, Francesca), attraverso la sintassi capisco che devo creare Paolo amare Francesca, attraverso la morfologia fletto/coniugo invece il verbo da amare ad ama). L'albero ottenuto in questo momento prende il nome di **Sentence Plan** che è completamente diverso dal **Dependency Tree** visto nell'NLP.
Avremo delle regole morfologiche (verbo al passato: +"ed", plurale +"s" ecc.) e sintattiche (soggetto prima del verbo, soggetto e verbo concordano in numero ecc.). Fino allo step precedente abbiamo utilizzato il lessico. In input abbiamo un albero e tramite la morfologia e la sintassi otteniamo una stringa.
7. **Orthographic realization:** Teniamo conto delle regole specifiche di una lingua che riguardano la punteggiatura e la formattazione, nel caso in cui l'output venga mostrato su schermo. Si applicano delle regole ortografiche. Ad esempio: frasi dichiarative finiscono con un punto, le frasi cominciano con la maiuscola. Bisogna decidere anche il font, se il testo deve essere visualizzato.

I 7 task vengono suddivisi in tre sotto-fasi perché vengono utilizzati due domini di conoscenza: uno che riguarda il linguaggio e uno il dominio specifico applicativo:

- **Text planning:** dipende solo dal dominio applicativo, il linguaggio non gioca alcun ruolo. **Usiamo solo informazioni sul dominio.**
 - Content determination
 - Discourse planning
- **Sentence planning:** dipende dal dominio e dalla lingua di arrivo. **Servono informazioni che dipendono dal dominio e dalla lingua di arrivo.**
 - Sentence aggregation
 - Lexicalisation
 - Referring expression generation
- **Linguistic realization:** non dipende dal dominio applicativo. **Dipende solo dalla lingua di arrivo.**
 - Syntax + morphology
 - Orthographic realization



L'output della fase di text planning è un **text plan** che ha informazioni che riguardano esclusivamente il dominio. È una sorta di albero a dipendenze. Si definiscono dei **satellite** entro i quali si hanno relazioni, concetti ed entità. Il text plan è l'input per la fase di sentence planning.

```

((type textplan)
  (relations ((sequence satellite-01 satellite02)))
  (satellite-01 ((nucleus ((message-id msg091)
                            (process exists)
                            (args ((object train)
                                   (source aberdeen)
                                   (destination glasgow)
                                   (frequency ((number 20) (period day))))))))
  (satellite-02 ((relations ((elaboration nucleus satellite-01)))
                (nucleus ((message-id msg092)
                          (process identity)
                          (args ((arg1 next-train)
                                 (arg2 cal-express))))))
  (satellite-01 ((nucleus ((message-id msg093)
                            (process departure)
                            (args ((object cal-express)
                                   (time 1000)))))))

```

L'output della fase di sentence planning è un **sentence plan(s)** che contiene parole e verbi, quindi ha a che fare con il linguaggio, anche se non è completo. Ad esempio, manca la coniugazione dei verbi

```

(S1/ThereBe
 :object (O1/train
           :cardinality 20
           :relations
             ((R1/period :value daily)
              (R2/source :value Aberdeen)
              (R3/destination :value
Glasgow)))

```

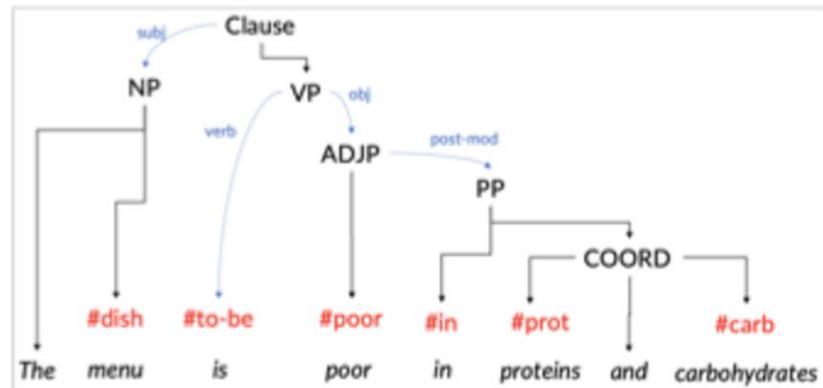
There are 20 trains daily from Aberdeen to Glasgow.

SIMPLE NLG SULLE SLIDE PER ESAME

Simple NLG (libreria Java)

Permette di fare morphological e syntactic realization.

Prende input una sorta di albero a dipendenze e restituisce in output la stringa con function words, parole flesse e ordinate. Molto utilizzato in ambito industriale, sviluppato da Reiter.



Fa solo realization, non si occupa della lessicalizzazione.

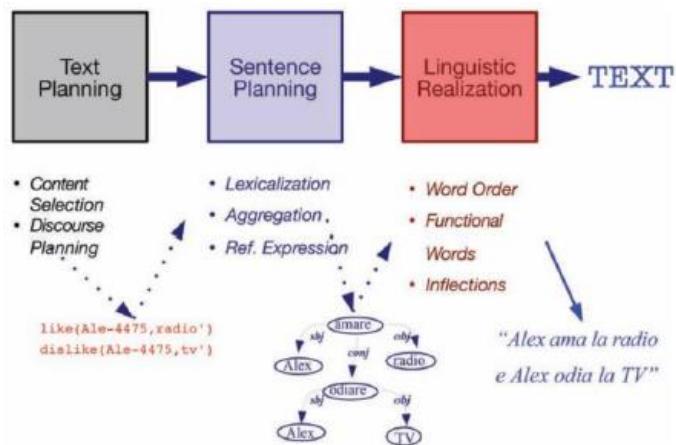
Conclusioni

Il NLG utilizza due domini di conoscenza uno che riguarda il linguaggio e uno il dominio specifico applicativo.

La fase di **text planning** dipende solo dal dominio applicativo. La lingua di arrivo non fa alcuna differenza.

La fase di **sentence planning** dipende sia dal dominio applicativo che dalla lingua.

La fase di **linguistic realization** dipende solo dal linguaggio.



SPEECH RECOGNITION (No esame)

Fondamentalmente è il task che trasforma il parlato in scritto, ed è il primo task che serve per poter comporre un sistema di linguistica computazionale, si pensi ai grandi player dell'informatica e più in generale della linguistica computazionale come Alexa, Google e Siri. Il problema è che tra di loro c'è poca differenza poiché posseggono tutti una grande quantità di dati, a differenza dei piccoli open-source che hanno pochissime informazioni a disposizione.

Mozilla è una organizzazione no-profit con l'obiettivo di valorizzare diversi aspetti di tecnologie internet (come Firefox) molto sicuro. Si è negli ultimi anni interessata molto della speech recognition, poiché ha preso molta importanza nell'ultimo periodo. Lo scopo è quello di creare dei sistemi di linguistica, che quindi portino a NLP, creando una banca dati del parlato che viene associato da del testo e si crea una grossa quantità di dati che poi viene processato per creare un riconoscitore del parlato. L'italiano fa parte di questo progetto.

MACHINE TRANSLATION

Il concetto di traduzione automatica, ci permettono di non rimanere lontani culturalmente ma di essere in grado anche senza conoscere una lingua di capire cosa si intende in altre lingue.

Già nel 1933 ci si impegnò per creare una macchina di traduzione, ovviamente in maniera meccanica in quanto non esisteva l'elettronica, nel 47 all'inizio della guerra fredda, permisero che gli interessi per la traduzione automatica aumentassero di molto, a causa dell'esigenza dei popoli di voler tradurre le infomazioni dei popoli opposti. Nel 56, viene per la prima volta coniato il termine di Intelligenza Artificiale e subito si individua nella machine translation un'importante compito dell'AI. Nel 66, ci si pose la domanda se i soldi spesi fino ad allora fossero spesi bene e se servissero per la machine translation. Per valutare che le macchine traducessero bene, si è provato a tradurre manualmente dall'inglese al russo, e poi provare a tornare allo stesso punto di partenza, ma con qualche delusione poiché si è ottenuta una traduzione poco attinente.

Secondo qualche filosofo la traduzione è un compito molto difficile.

Nel 2000 nasce uno dei sistemi di On-line MT (Babelflic), nel 2007, Google passa ad un sistema statistico di traduzione. Nel 2016, si passa a sistemi concentrati sui sistemi neurali (Seq2Seq). Nel 2017 è stato presentato il primo sistema di traduzione real-time.

Machine Translation classica

La MT classica è basata sulla linguistica

Traduzione letterale

È una traduzione termine a termine. Talvolta è impossibile perché ci sono termini che non sono traducibili fra lingue (**lexical gap**). È impossibile catturare tutte le sfumature di una traduzione con la traduzione letterale.

Traduzione non letterale

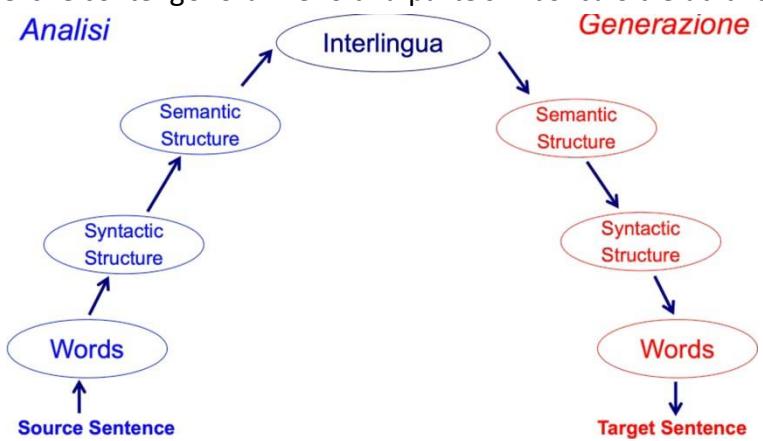
È utile per fare traduzioni approssimative poiché le frasi hanno poca semantica sottointesa, sono abbastanza corte, imperative e senza troppe complicazioni (**Es:** ricette di cucina). È utile nei sistemi **CAT** (Computer-Aided human Translation) in cui la macchina fornisce una traduzione primordiale che viene verificata dall'utente. Utile per previsioni del tempo, basate su uno pseudo-linguaggio.

Esistono anche altri campi in cui la traduzione automatica è fondamentale, come ad esempio nell'unione europea non c'è una lingua sola, quindi quando viene scritto un documento, questo deve essere scritto in più lingue sul quale lavorare in parallelo.

I 3 PARADIGMI DELLA TRADUZIONE AUTOMATICA

I tre paradigmi storici per la traduzione automatica nei modelli simbolici (utilizzata oggi grazie al livello di precisione molto alta) sono così chiamati perché contengono almeno una parte simbolica oltre ad una possibile parte statistica.

Guardando solo la parte in blu, stiamo vedendo la pipeline classica di analisi dei sistemi di NLP, chiamata pipeline di analisi, seguita dalla parte in rosso, di generazione in senso opposto alla precedente. Questa struttura chiamata **triangolo di Vauquois** permette di identificare come gli umani effettuano la traduzione di una frase da una lingua ad un'altra e quindi ci permette di capire quali componenti software potrebbero essere necessarie per effettuare la traduzione.



Ipotesi di Sapir Whorf: pensiamo come parliamo. Il modo in cui parliamo e ragioniamo sono legati. Se nella nostra lingua non abbiamo una parola, non riusciamo a ragionare nei termini di quella parola.

A seconda del punto di partenza e di arrivo nel triangolo di Vauquois abbiamo diversi schemi di MT:

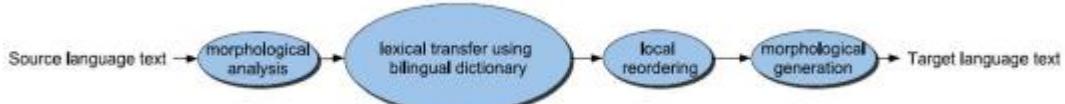
1. **Direct:** traduciamo parola per parola, ignorando la struttura sintattica, si fa reordering e otteniamo la sintassi. Si utilizzano soltanto le parole. Si può fare utilizzando regole o statistica. Funziona bene quando le lingue sono molto simili, come l'italiano e lo spagnolo.

Abbiamo:

- **Analisi morfologica:** lemmatizzazione e riconduzione alle forme normali
- **Transfer lessicale:** si traducono le parole dal linguaggio sorgente al linguaggio destinazione utilizzando un dizionario bilungue o delle regole
- **Riordinamento locale:** le parole vengono riordinate per rispettare la sintassi della lingua target. È la parte più difficile.
- **Generazione morfologica:** si verifica l'ortografia e la morfologia delle parole prodotte affinché siano corrette rispetto alla lingua target.

N.B.: con frasi più lunghe e complesse perde molto di qualità.

N.B.2: Google Translate è un sistema direct perché non usa esplicitamente (però può farlo implicitamente) strutture dati dei livelli superiori del triangolo di Vauquois



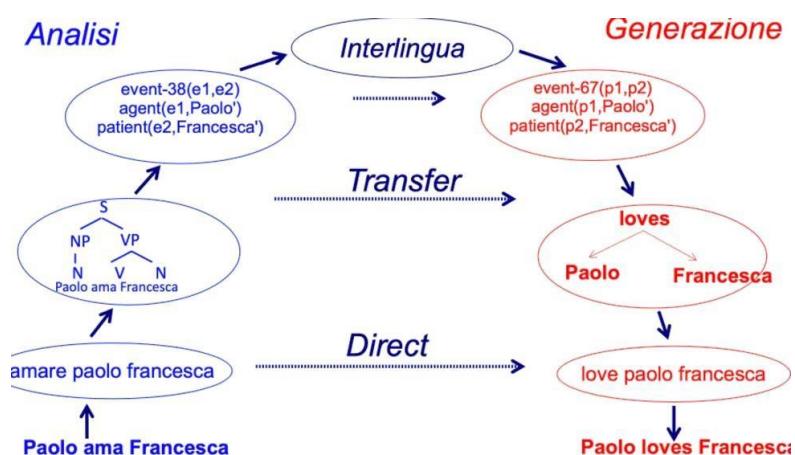
2. Transfer (sintattico o semantico): consideriamo le strutture sintattiche e semantiche. Si lavora con le strutture semantiche, quindi FoL, che derivano dalla lingua. L'idea è quella di trasformare un albero in un altro. **BabelFish** è un esempio di MT. Consiste di tre fasi:

- La prima è di **analisi** in cui facciamo analisi morfologica, PoS tagging, chunking e costruiamo le dipendenze.
- Nella seconda fase, di **transfer**, costruiamo la traduzione degli idiomi ed effettuiamo alcune operazioni di disambiguazione. In questa fase si trasformano gli alberi sintattici. La complessità di questa fase è proporzionale alla distanza tra le lingue.
- Nell'ultima fase, detta di **sintesi(generazione)**, andiamo ad utilizzare un dizionario bilingue per la traduzione dei termini e abbiamo a che fare con il riordinamento per poi andare ad eseguire una generazione morfologica.



A livello pratico le regole di transfer vanno ad operare sugli alberi cambiandone la struttura e operando quel riordinamento che era complicato da fare nel caso della traduzione diretta. Per questo l'operazione di traduzione transfer può essere complessa.

- 3.** Nella tecnica con approccio **interlingua**, si cerca di staccarsi dalle parole capire il significato e produrlo indipendentemente dalle parole utilizzate nella lingua originale. In questa tecnica quindi si prevede un'analisi completa del testo e ci si stacca dal concetto di qualsiasi tipo di rappresentazione. Prende il nome di **transfer semantico** quella tecnica tra interlingua e transfer, in cui la traduzione viene effettuata tra le descrizioni della semantica della frase (nell'immagine descritta attraverso la FoL), che comunque fa riferimento ad una struttura semantica, e di conseguenza ad una lingua, a differenza dell'interlingua pura che invece si sgancia totalmente dalla lingua di partenza (difficile da implementare e da creare). È difficile da applicare perché alcune parole non hanno una corrispondenza nella lingua target, ad esempio il giapponese distingue fratello-minore da fratello-maggiore. Kant utilizza l'interlingua.



Inoltre, ci sono degli approcci puramente statistici alla MT. Le tecniche viste finora possono essere affiancate dalla statistica, ma nei metodi completamente statistici la situazione è differente. L'idea, data da Warren Weaver, è quella di trattare la MT come una specie di problema logico, un puzzle, che deve essere risolto, idea nata per la traduzione dei geroglifici. Si aveva a disposizione 3 testi con lo stesso significato ma scritto in 3 lingue differenti. In questo modo si può iniziare ad inserire delle regole puramente statistiche di sostituzione tra le varie lingue che massimizzano le probabilità di allineamento di frasi e di parole, e sulla base di queste due probabilità arrivo alla traduzione finale. Quindi tecniche di Machine Learning con dei testi allineati in lingue differenti possono imparare le probabilità di allineamento e tradurre nuove frasi in input in basi a tali probabilità. Già dal 1987 si pensava a tali sistemi, ma non si aveva abbastanza potenza di calcolo e abbastanza quantità di dati e infine grazie all'avvento di reti ricorrenti con celle di memoria.

Per valutare le tecniche di traduzione o ci si affida persone fisiche oppure ci si usa una metrica che si basa su n-grammi automatiche

Esame Mazzei: 2 domande + progetto