# TLN - Mazzei

> Appunti di Federico Torrielli (2022)

# Lezione 01: intro

## Programma

- Studio computazionale del linguaggio
- Parsing e Generazione del linguaggio
- Intro: Traduzione automatica e Dialogo

## Libro

- Speech and Language Processing

## Keywords

- NLP: Natural Language Processing
- CL: Computational Linguistics
- Lexicon: dizionario
- Morphology: morfologia
- Syntax: sintassi
- Semantics: semantica
- Conversational Interface: interfaccia di conversazione
- Conversational agent: agente di conversazione
- Parsing
- NLG: Natural Language Generation
- MT: Machine Translation
- Grammar
- Treebank
- NL ambiguity (attachment/coordination)
- BOT

## Elementi caratteristici del linguaggio umano

- **Discretezza**: elementi atomici, fonemi, morfemi e parole per delimitare il senso
- **Ricorsività**: profondità della lingua umana riconosciuta dagli umani
- **Dipendenza dalla struttura**: concetto di sequenza importante per il linguaggio, ma non è completa per la sua complessità generale (forse meglio un grafo o

albero)
- **Località**: la parola dipende dal contesto

# Capacità di un bot

- Speech recognition
- Natural Language understanding
- Natural language generation
- Speech generation

*Ovvero*:

- Riconoscere il suono
- Capire le singole parole
- Raggruppare le parole ed assegnarne una sintassi
- Assegnare semantica lessicale e formale alle parole e alle frasi, contando il contesto (pragmatica)
- Capacità di conversazione, capire le convenzioni del discorso

*Ovvero*:

- Phonetics - Phonology
- Morphology
- Syntax
- Semantics
- Pragmatics
- Discourse

## Prima ipotesi della linguistica computazionale: modellazione a livelli

Posso modellare, studiare ed analizzare il linguaggio a "livelli", che sono quelli che abbiamo appena visto.

## Seconda ipotesi della linguistica computazionale: architettura a cascata

I risultati del primo livello vengono dati al secondo livello, e così via.

## Garden Pathing

Ipotesi di come potrebbe funzionare il nostro cervello: nell'analisi delle frasi facciamo backtracking ad ogni nuova informazione.
Dire, ad esempio, "quando lavai i piatti, caddero", fa cambiare la scena non appena

arriviamo alla parola "caddero", così da far cambiare
il senso totale del discorso, costringendoci ad un backtracking.

## Strutture linguistiche

- Rappresentiamo relazioni con alberi
- La sintassi ci dice che le parole si comportano come unità quando vengono messe insieme
- Ambiguità: una frase come "I made her duck", può avere tantissimi significati!
  Tipo "Le ho cucinato una papera" oppure "L'ho fatta accovacciare" oppure "L'ho trasformata in una papera"...
  noi esseri umani abbiamo imparato a contestualizzare, cosa che non è semplicissima per un computer, a priori di informazione!

## Stato dell'arte

- Traduzioni
- Guessing games (computer che vince a Jeopardy)
- Summarization
- Tecnologie Vocali
- GPT-3

# Lezione 02: Strutture dati per la linguistica

## Prologo: regole vs statistica

Regole e statistica sono due metodi per "potenziare" gli algoritmi NLP.
Sono due prospettive diverse per una singola realtà: molto spesso si utilizzano **approcci ibridi**.

Esempio: *quando finisce una frase?*

**Regole**: !,?,. ← fine frase | Questo sarebbe un sentence splitter a regole, per produrne un decision tree

> *Ma è uguale più o meno a fare una statistica con dati annotati*

**Statistica**: n% con !, m% con ?... etc etc... | Per produrne un feature tree!

> *La natura dei classificatori, alla fine, è la stessa!*

## Modelli di ML utilizzati da NLP

- Decision trees
- Logistic regression
- SVM
- Perceptron
- Neural Networks
- etc...

Punto cruciale: **la scelta delle features**.

# Livello morfologico e analisi lessicale

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

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

Altro esempio: suffissi come *mente* (Avverbi) oppure *one* (Maggiorativo).

## Lemma vs Stemma

- Stemming: radice della parola → Verbo amare ha *amar* come stemma
- Lemma: forma normale della parola

## Parti del discorso (Part Of Speech - POS)

#### Basi

- nome
- verbo
- aggettivo
- avverbio

#### Functions

- preposizione
- articolo
- pronome
- congiunzione
- interiezione

## Paradigmatico vs Sintagmatico

- Paradigmatico: rapporto che intercorre tra gli elementi della frase e gli elementi che virtualmente potrebbero alternarsi con essi nella frase.
- Sintagmatico: gli elementi che si succedono nella frase $\longleftrightarrow$ formano un sintagma: articolo e nome sono in relazione sintagmatica

### Nome

- Approccio semantico: gli oggetti vengono rappresentati con i nomi
- Approccio paradigmatico: posso sostituire il nome, la frase continua ad essere corretta: "mi piace la bicicletta" $\leftarrow$ "mi piace la banana"

Sono oggetti statici, che *durano* nel tempo.

### Verbo

- Caratterizzazioni morfologiche: tempo, modo, numero
- Categorie: ausiliari, modali, copula...

Sono oggetti che stanno nel *divenire*, linguisticamente la parte più importante di una frase.
Dal punto di vista computazionale, il verbo è importante per la logica del prim'ordine: lo vedremo poi meglio più in là.

## Open vs Closed classes

- Open Classes: nome, verbo, aggettivo... Si chiamano open dato che vengono costantemente aggiornate, dato che la lingua cambia.
- Closed Classes: articoli, pronomi, preposizioni. Esse variano, ma molto lentamente.

Verbi modali, numeri sono strutture che sono a metà tra open e closed.

> *Guardare il disegno sul libro*

| Open class words | Closed class words | Other |
|---|---|---|
| ADJ | ADP | PUNCT |
| ADV | AUX | SYM |
| INTJ | CCONJ | X |
| NOUN | DET | |
| PROPN | NUM | |
| VERB | PART | |
| | PRON | |
| | SCONJ | |

# Lezione 03: il Livello Sintattico

Secondo livello dello stack visto inizialmente (*morfologico - sintattico - semantico - pragmatico*).

## Costituente vs Dipendente

- Struttura **Costituente**: rappresenta le relazioni di gruppo tra le parole
- Struttura **Dipendente**: rappresenta le relazioni binarie tra le parole

Gli alberi di costituenti e dipendenze sono equivalenti, ma in due "lingue" diverse.
La **teoria x-barra** deve essere vista come un *traduttore* tra i due mondi.

### Costituenza

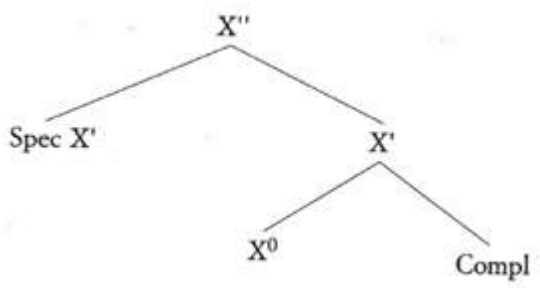
Viene organizzata la struttura ad albero per evidenziare le relazioni di gruppo.
Tecnicamente: **costituente = gruppo di parole continue**.

Esse:

- Si comportano come un'unità
- Che hanno proprietà sintattiche

#### Teoria x-barra

- **A cosa serve**: individuare principi sintattici comuni a tutte le lingue.
- **Principio fondante**: tutte le lingue condividerebbero certe similarities strutturali nel loro modo di formare frasi
  Struttura a costituenti che ci permette di identificare le relazioni grammaticali come se fosse una grammatica context-free.
  Questa struttura viene sovrapposta alla nostra frase per estrarre le relazioni di cui prima.




### Dipendenza

Relazioni tra due parole (binarie):

- **testa**: parola dominante
- **dipendente**: parola dominata

La testa seleziona le sue dipendenti e ne determina le proprietà (ad es. il verbo determina se il nome è singolare o plurale ad esempio)

In *Paolo ama Francesca*, Paolo e Francesca sono argomenti.
In *Paolo corre velocemente*, Paolo è argomento, velocemente è un **modificatore**.

Gli argomenti fanno sì che una frase sia **grammaticale**, i modificatori no.

# Lezione 04: il Livello semantico e pragmatico

## Livello semantico

Semantica vuol dire dare un'**interpretazione** a qualcosa. Questa è la *trasformazione* dal linguaggio naturale in una qualche forma di rappresentazione della conoscenza.
Interpretare vuol dire dare una **considerazione soggettiva** alla parola, parte del discorso etc... etc...
La semantica è organizzata a livelli dato che è *complessa*, più complicata rispetto al livello morfologico.

Sottolivelli:

- Semantica **lessicale**: semantica singolare della parola (il *sintagma*).
    - Semantica **lessicale classica**: significato di *senso* di una parola, a priori del contesto. Il **lessema** ha una *forma* che ha un *senso*.
      La struttura dati adatta per il lessema è il *vocabolario*? No, si formano dipendenze circolari, per il problema delle forme **primitive** (rosso, non si può spiegare se non con esempi).
      Potremmo utilizzare delle relazioni come **omonimia**, **polisemia**, **sinonimia**, **iponimia**, **iperonimia** per dare delle relazioni significative tra le parole *problematiche*.
      La struttura dati adatta è la **tassonomia**, ovvero la costruzione di **synset** (un database lessicale *machine-readable* organizzato per semantica).
      *bank* può avere come synset $S_1 = bank, deposit$ ed un'altro per le altre relazioni.
    - Semantica **distribuzionale**: significato di una parola lo da il contesto, le parole che la circondano.
- Semantica **formale**: detta anche **composizionale**, richiede delle competenze specifiche per capire la *frase*, componendo i significati. Un esempio estremamente semplificato potrebbe essere *X AND Y*.

Nel caso del linguaggio dobbiamo *metterci d'accordo* per fare semantica composizionale, le macchine non possono farla da sole.
Fondamentalmente useremo la **logica dei predicati** per esprimere la semantica composizionale (costruzione di *alberi* + composizione).

### Semantica formale

- **Esempio**: *Giorgio ama Maria* $\rightarrow$ **Ama**(*Giorgio*,*Maria*)
- **Altro Esempio**: $\forall X$ *uomo(X)* $\rightarrow$ *amare(X,Maria)* **&** *amare(Giorgio,X)* $\Rightarrow$ *amare(Giorgio,Maria)*

Non abbiamo bisogno di capire ad esempio il tempo verbale per esprimere la logica del dicorso.
Utilizzeremo il **Lambda calcolo** per comporre il significato: partiremo dalle foglie per produrre una struttura dati per rispondere alla domanda implicita posta dalla frase.

## Livello Pragmatico

- Ultimo livello dello *stack* del linguaggio.

*Oggi mi trovo ad Alessandria* prendiamo questa frase dal livello semantico formale e **contestualizziamo** questo nel discorso (a livello pragmatico).

Un tipico *task* della pragmatica è fare l'anafora-resolution della frase: "La torta era sul tavolo, Giorgio *la* divorò".

## Intenzione ed Entità nel dialogo

- Intenzione: che cosa voglio dire? *"What's the forecast?" == "What's the weather like right now?"*.
- Entità: cosa riusciamo ad estrarre dalla frase? *"What's the temperature going to be in Seattle tomorrow?"* $\rightarrow$ *{location: Seattle,time: tomorrow}*

## Strutture dati per ogni livello

- **Livello morfologico**: *lista*
- **Livello sintattico**: *alberi*
- **Livello semantico**: *insiemi e logica*
- **Livello pragmatico e del discorso**: *frame*

# Lezione 04 (parte 2): Pos Tagging

- **Input**: frase (sequenza di parole)
- **Output**: sequenza di **tag**

- **Why**: ci serve per dare identità alla parola, soprattutto in presenza di ambiguità
- **Difficoltà**: *85%* delle parole (function words, congiunzioni ...) non sono ambigui, ma la restante parte sono parole **usatissime** (vd. *back* parola usatissima ma estremamente ambigua).
- **Performance**: attorno al *90%* minimo (*baseline*), **97%** per sistemi correnti.

Un set di POS si chiama un **tagset**, un esempio famoso (e vecchio) è il **Penn Treebank**, con una buona granularità tra i POS.
L'altro molto famoso è l'**Universal Dependencies** treebank, che è partito da *Google* (duh): è più piccolo del Penn ma più moderno in quanto comprende emoji, simboli strani e unknowns.

# Algoritmi per il POS Tagging

## Supervised machine learning

- Neural sequence Models
- Large Language Models

## Rule based PoS tagger

- ENGTWOL (english two level analysis) $\rightarrow$ rules + statistica

## Stochastic PoS taggers

- HMM (Hidden Markov Models)
- MEMM (Maximum entropy markov models)

# Rule-based tagging

Idea base dell'**algoritmo**:

1. Assegna tutti i possibili tag alle parole (**analisi morfologica**)
2. Rimuovi i tag a seconda del set di regole per tipo (se *word+1* è aggettivo, verbo o quantificatore $\rightarrow$ e *word+2* è un terminatore, e *word* non è "consider" allora elimina *non-adv* altrimenti elimina *adv*)

Queste regole sono solitamente scritte a mano oppure apprese con machine-learning.

# Stochastic PoS taggers

- Tagging come **sequence labeling**
- Visione probabilistica del problema

Idea base dell'**algoritmo** (impossibile):

1. Considera tutte le possibile sequenze di tag
2. Da quell'universo di sequenze, scegli la sequenza di tag che è la più probabile data la sequenza di n words.

Problema: **esplosione combinatoria** di alcune frasi come *time flies like an arrow*.

# Fasi del PoS Tagging stocastico

- Fase di **Modelling**: modellare il problema
- Fase di **Learning**: settiamo i parametri del modello
- Fase di **Decoding**: applicare il modello sullo specifico esempio.

# HMM: Modelling

Hidden Markov Model: algoritmo di machine learning **generativo** (vuol dire che utilizziamo la **regola di Bayes**) in cui cerchiamo la sequenza di stati nascosti (tags) $\hat{t}_1^n$ più probabile.

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

**Dove**:

- t: tags
- w: word

**Come**:

1. Applichiamo la regola bayes (naive) *MAP to likelihood*
2. Facciamo l'assunzione Markoviana di grado 1
3. Risultato da chad, passaggi banali: $\hat{t}_1^n = argmax_{t_1^n} P(w_1^n|t_1^n) \cdot P(t_1^n)$

**Output**: la sequenza di tag più probabile

## Likelihood e Prior (Applichiamo l'assunzione Markoviana)

$\hat{t}_1^n = argmax_{t_1^n} P(w_1^n|t_1^n) \cdot P(t_1^n)$

**Dove**

$P(w_1^n|t_1^n) \approx \prod_{i=1}^{..} P(w_i|t_i)$

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

**Quindi**

$\approx argmax_{t_1^n} \prod_{i=1}^{n} P(w_i|t_i) \cdot P(t_i|t_{i-1})$

# HMM: Learning

Se abbiamo un corpus di PoS annotati a mano etc.. etc.. fare il learning dell'HMM è semplicemente equivalente a **contare**!
Per esempio $P(NN|DT)$ è la probabilità di trovare un nome dopo un articolo, semplicissimo!

$P(NN|DT) = \frac{C(DT,NN)}{C(DT)} = \frac{56,509}{116,454} = 0.49$ Dove *C = count*

> *Più grande è il nostro corpus e migliore avremo questa stima (come in tutto il machine learning con modelli generativi).*

Il numero dei *count* che facciamo è $(n+2)^2$.

$P(w_i|t_i) = \frac{C(t_i,w_i)}{C(t_i)}$

Ora calcoliamo anche la **probabilità di verosimiglianza**, dato che abbiamo calcolato quella a priori proprio ora:
$P(is|VBZ) = \frac{C(VBZ|is)}{C(VBZ)} = \ldots = 0.47$, la likelihood che il verbo VBZ (*3sg Pres Verb*) sia **"is"**.

## HMM: Decoding

Come faccio ad applicare allora il mio modello statistico?

*Secretariat is expected to race tomorrow* (race dipenderà solo da **to** nel sapere se è *VB* oppure *NN*).
Abbiamo però un'esplosione combinatoria delle possibili sequenze (il numero di combinazioni possibili per una singola frase è assurdamente alto).
Se ci sono 30 tag e una sentence media è circa 20 words, allora quante sequenze di tag dobbiamo enumerare nello scenario peggiore? $30^{20}$.

Non è possibile approcciarsi al problema con un **all-path** algorithm.
Con la **programmazione dinamica** possiamo "rompere" questa probabilità in 2 parti:

- Probabilità della miglore sequenza di tag fino a **j-1** (probabilità dei prefissi)
- Moltiplicata per la probabilità di transizione dal tag fino alla fine della sequenza **j-1 fino a T**

Ovvero, calcoliamo la probabilità delle soluzioni parziali, le memorizziamo in una **matrice di Viterbi** e poi andiamo avanti.
La chiave di questa tecnica è che dobbiamo solo memorizzare il path di MAX prob. ad ogni cella, con complessità finale quindi di $O(N^2)$

**Algoritmo di Viterbi (sommario)**:

- **Fase di inizializzazione**: Crea una matrice con *colonne = token* e *righe* corrispondenti ai *POS TAGS* possibili

- **Fase ricorsiva**: Attraversa la matrice in una passata riempiendo le colonne da sinistra a destra considerando solo la colonna precedente ogni volta che andiamo avanti (prefisso da cui sono partito) $v_t(j) = max_{s=1}^{n} v_{t-1}(s') * a_{s',s} * b_s(o_t)$ dove:
  $a_{s,s'}$ *indica la probabilità di transire dal tag s-esimo al tag s'-esimo*
  $b_s(o_t)$ indica la probabilità di osservare il simbolo $o_t$ dato lo stato corrente s (non ha il pedice in s quindi questa quantità è al di fuori del massimo)
  Utilizzo poi un **backpointer** (quale stato precedente ha portato allo stato corrente, parte da 0). Nella seconda ricorsione, ad esempio, abbiamo tutti i valori di Viterbi precedenti (stati nascosti), e teniamo solo quello più alto, immagazzinandolo nel backpointer (alla fine sto backpointer serve a tenere il path dei valori massimi durante il percorso).
  Dobbiamo considerare anche queste due cose: la probabilità iniziale che il primo tag sia il primo della frase e che l'ultimo tag sia proprio l'ultimo della frase.
- **Fase di terminazione**: Proprio in questa frase calcoliamo il backpointer come $backpointer(q_F, T) = argmax_{s=1}^{N} viterbi(s, T) * a_{s,q_F}$ ovvero in questo caso non ci frega di sapere la probabilità di emissione in quanto la abbiamo dal passo precedente, ma dobbiamo sicuramente moltiplicare per la probabilità del singolo tag di essere a fine frase.

```
# probability == p. Tm: the transition matrix. Em: the emission matrix.
function viterbi(O, S, Π, Tm, Em): best_path
    trellis ← matrix(length(S), length(O))  # To hold p. of each state given each
observation.
    pointers ← matrix(length(S), length(O)) # To hold backpointer to best prior
state
    # Determine each hidden state's p. at time 0...
    for s in range(length(S)):
        trellis[s, 0] ← Π[s] * Em[s, O[0]]
    # ...and afterwards, tracking each state's most likely prior state, k.
    for o in range(1, length(O)):
        for s in range(length(S)):
            k ← argmax(k in trellis[k, o-1] * Tm[k, s] * Em[s, o])
            trellis[s, o] ← trellis[k, o-1] * Tm[k, s] * Em[s, o]
            pointers[s, o] ← k
    best_path ← list()
    k ← argmax( k in trellis[k, length(O)-1] )   # Find k of best final state
    for o in range(length(O)-1, -1, -1):  # Backtrack from last observation.
        best_path.insert(0, S[k])         # Insert previous state on most likely
path
        k ← pointers[k, o]                # Use backpointer to find best previous
state
    return best_path
```

## HMM: Problemi

- **Sparseness**: non c'è la possibilità di mettere nuove informazioni su Features nell'hidden Markov model (ad esempio, come gestire lettere maiuscole e minuscole all'inizio della parola... che in questo caso dovrebbero essere trattati come due parole diverse!)

- **Complessità**: modello complesso da calcolare

Per il primo problema, tocca cambiare modello...

# Lezione 05: MEMM

- **Modello (Probabilistico) Discriminativo**
- Feature **pesate**
- **Uniforme**: entropia massima tra tutti i modelli

## MEMM: Modelling

$$
\begin{aligned}
\hat{T} &= \operatorname*{argmax}_T P(T|W) \\
&= \operatorname*{argmax}_T \prod_i P(t_i|w_{i-l}^{i+l}, t_{i-k}^{i-1}) \\
&= \operatorname*{argmax}_T \prod_i \frac{\exp\left(\sum_i w_i f_i(t_i, w_{i-l}^{i+l}, t_{i-k}^{i-1})\right)}{\sum_{t' \in \text{tagset}} \exp\left(\sum_i w_i f_i(t', w_{i-l}^{i+l}, t_{i-k}^{i-1})\right)}
\end{aligned}
$$

> $w_i$ *dentro le parentesi sono parole, fuori invece sono pesi*

- Funziona molto bene quando il corpus è *piccolo*
- Utilizzo feature booleane

## MEMM: Learning

- Utilizziamo il logaritmo
- Non facile da computare

- MaxEnt learning -> Multinomial Logistic Regression

  Learning $\hat{w} = \underset{w}{\operatorname{argmax}} \sum_j \log P(y^{(j)}|x^{(j)})$
- **Conditional maximum likelihood estimation:** we choose the parameters w that maximize the (log) probability of the y labels in the training data given the observations x.
- Convex optimization problem, so we use hill-climbing methods like stochastic gradient ascent, L-BFGS (Nocedal 1980, Byrd et al. 1995)

## MEMM: Decoding

- Possiamo usare Viterbi anche qui
- Formula su Slides

## MEMM: Smoothness Problem

Per trattare parole *mai viste prima* (smoothness problem) possiamo usare la **smoothness** per risolvere questi problemi:

- Assumiamo che siano nouns
- Assumiamo distribuzione uniforme sulla PoS
- Utilizziamo un dizionario esterno (tipo morph-it)
- Utilizziamo dell'informazione morfologica (tipo rules)
- Assumiamo che delle parole sconosciute abbiano una distribuzione di probabilità similare alle parole che occorrono una sola volta nel training set (*funziona molto bene*, ma dipende dal corpus)

## MEMM: Evaluation

Si confronta con un **Gold Standard**, ovvero un corpus taggato *by-hand*.
Molte volte utilizziamo questa divisione:

- Training set: 80%
- Development set / Validation set: 10%
- Test set: 10%

## BIO Tagging

Da internet:

*The BIO / IOB format (short for inside, outside, beginning) is a common tagging format for tagging tokens in a chunking task in computational linguistics (ex. named-entity recognition). The B- prefix before a tag indicates that the tag is the beginning of a chunk, and an I- prefix before a tag indicates that the tag is inside a chunk. The B- tag is used only when a tag is followed by a tag of the same type without O tokens between them. An O tag indicates that a token belongs to no entity / chunk.*

Outline: si da un tag POS per token nella frase, rendendo ogni tag una tupla effettiva

> [PER Jane Villanueva] of [ORG United Airlines Holding], discussed the [LOC Chicago] route.

## NER Tagging

Da internet:

*Named-entity recognition (NER) (also known as (named) entity identification, entity chunking, and entity extraction) is a subtask of information extraction that seeks to locate and classify named entities mentioned in unstructured text into pre-defined categories such as person names, organizations, locations, medical codes, time expressions, quantities, monetary values, percentages, etc.*

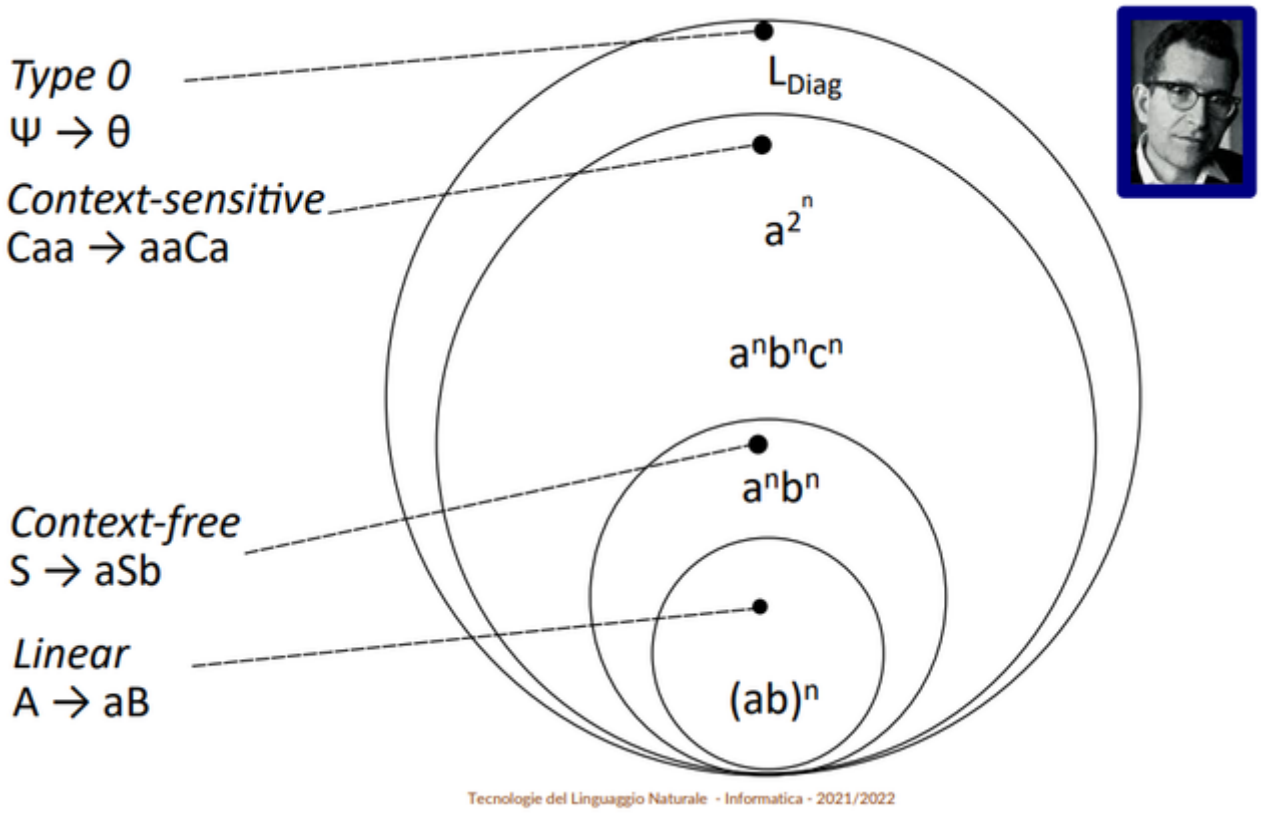
*Most research on NER/NEE systems has been structured as taking an unannotated block of text, such as this one:*

> *Jim bought 300 shares of Acme Corp. in 2006.*

*And producing an annotated block of text that highlights the names of entities:*

> *[Jim]Person bought 300 shares of [Acme Corp.]Organization in [2006]Time.*

# Lezione 06: Livello Sintattico




# Competence/Performance linguistica

- Chomsky osserva che esistono delle proprietà di **ricorsività** del linguaggio (in tutte le lingue)
- Gli umani hanno lingue **con contesto** (lingue naturali), per questo nasce la gerarchia di Chomsky → Type 0 (natural), Context-sensitive, Context-free, Linear.
- La **competence**: la grammatica in generale, il "sapere" questa grammatica. *L'insieme delle proprietà intrinseche della lingua stessa.*
- La **performance**: il ricordarsi effettivamente il discorso con un livello alto di ricorsività (per gli umani), l'implementazione dell'algoritmo (per i computer). *La capacità di un ascoltatore umano di comprendere la semantica di un discorso.*

# Il NL non è context-free!

Esistono delle lingue naturali parlate non context-free (es. dialetto germanico-svizzero o africano scoperti non essere CF) → e allora nella gerarchia di Chomsky dove siamo?

Siamo nel sottoinsieme del **context-sensitive** → *mildly context-sensitive languages*, scoperti da *Joshi* (1985), che includono linguaggi con grammatiche context-free, nested e cross-serial dependencies, **parsabili polinomialmente** e con una **proprietà di crescita lineare**.

- Proprietà di **crescita lineare**: se prendo tutte le parole e le metto in ordine crescente di lunghezza, non avrò mai dei salti ma avrò una crescita lineare.

- **Definition** A language $L$ is constant growth if there is a constants $c_0$ and a finite set of constant $C$ such that for all $w \in L$ where $|w| > c_0$ there is a $w' \in L$ such that $|w| = |w'| + c$ for some $c \in C$.

## Grammatiche nate da Mildly context-sensitive

Sono nate diverse grammatiche basate su:

- **Tree adjoining grammars**
- CCG (**Combinatory Categorial Grammars**)

Esse sono *debolmente equivalenti*.

In una context-free le strutture su cui sto lavorando sono: *lista* $\rightarrow$ *lista*, come regola di manipolazione solo la *sostituzione*.
Questi due formalismi cambiano le strutture oppure le regole di manipolazione (oppure entrambi!).

Ora che sappiamo che le lingue naturali sono midly context-sensitive, possiamo costruire parser per CF pur sapendo che perderemo delle informazioni nel processo.

### Tree Adjoining Grammars

Nelle Tree Adjoining Grammars abbiamo come struttura un'**albero** invece che una lista e come regola di manipolazione abbiamo la sostituzione e l'**adjoining** (l'innesto di un albero dentro un altro albero, ovvero l'utilizzo di strutture dati elementari per unire le parole nel senso tra di loro).

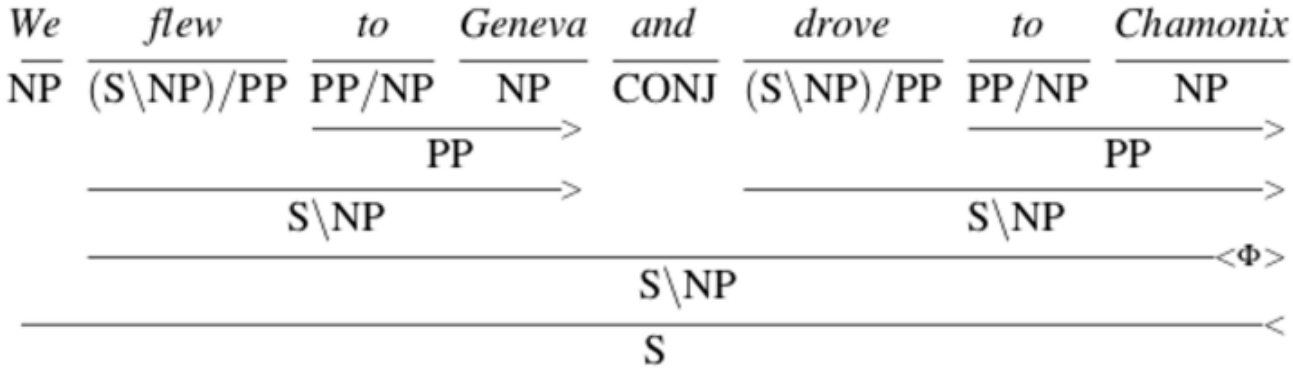
### Grammatiche Categoriali Combinatorie (CCG)

Grammatiche **bottom-up** (le grammatiche generative sono top-down): parto dalle foglie, ovvero le categorie. Ad ogni parola associo una categoria e la singola parola cerca altre categorie per combinarsi. Il verbo non sarà altro che un elemento che sta *cercando* qualcosa alla sua destra (soggetto) e qualcosa alla sua sinistra (complemento oggetto).

In *Paolo ama Francesca*, ama ha come regole:

- $X/Y\ Y \rightarrow X$
- $Y\ X\backslash Y \rightarrow X$

Paolo (*NP*) ama $((S\backslash NP)/NP)$ Francesca $(NP)$




# Parser

## Parser: anatomia

- **Grammatica** (*competence*): può essere CF, CCG, TAG, ...
- **Algorithm** (*performance*): strategia di ricerca (top-down, bottom-up...) + organizzazione della memoria (back-tracking, dynamic programming...)
- **Oracolo** (*oracle*): euristica (probabilistica, rule-based...)

## Parser: Top-down vs. Bottom-up

- **Top-down**: ha lo svantaggio di generare molte possibilità che non servono a niente, dovendo partire dalla radice (tipo una frase *imperativa* dovrebbe essere trattata solo come imperativa, ma per il top-down non va subito bene dato che deve processarle tutte)
- **Bottom-up**: Qui genero cose che non diventeranno mai una frase, ma una radice → (S - sentence)

Il top-down ha un altro problema aggiuntivo: la **ricorsione a sx**. Dato un parser top-down, left-to-right con backtracking e data una frase *Does this flight include a meal?* (vd. slides), ci accorgiamo velocemente che nel fare backtracking avendo la ricorsione a sx (tipico della lingua inglese, ma anche in italiano ne abbiamo tantissime di possibilità, vd. *bottiglia - di vino - di Giovanni*...) fa riempire la memoria con una struttura dati sempre più complessa, se non quasi interminabile. Per evitare questo devo avere una grammatica senza left recursion, certamente non facile da fare e soprattutto poco applicabile alle lingue naturali → **complessità esponenziale rispetto all'input**, soprattutto in caso di ambiguità.

## Ambiguità

- **attachment ambiguity** (Dove la parola sta attaccata nell'albero di parsing?): *il ragazzo vide la ragazza con il telescopio*
- **coordination** (tipo di attachment ambiguity con un *coordinated constituent*): *...televisioni e radio nazionali...*

L'ambiguità ha come vantaggio (per gli umani) che mi **permette di fare frasi molto compatte** nell'articolazione del discorso.

# Lezione 7: algoritmo CKY e CKY-probabilistico

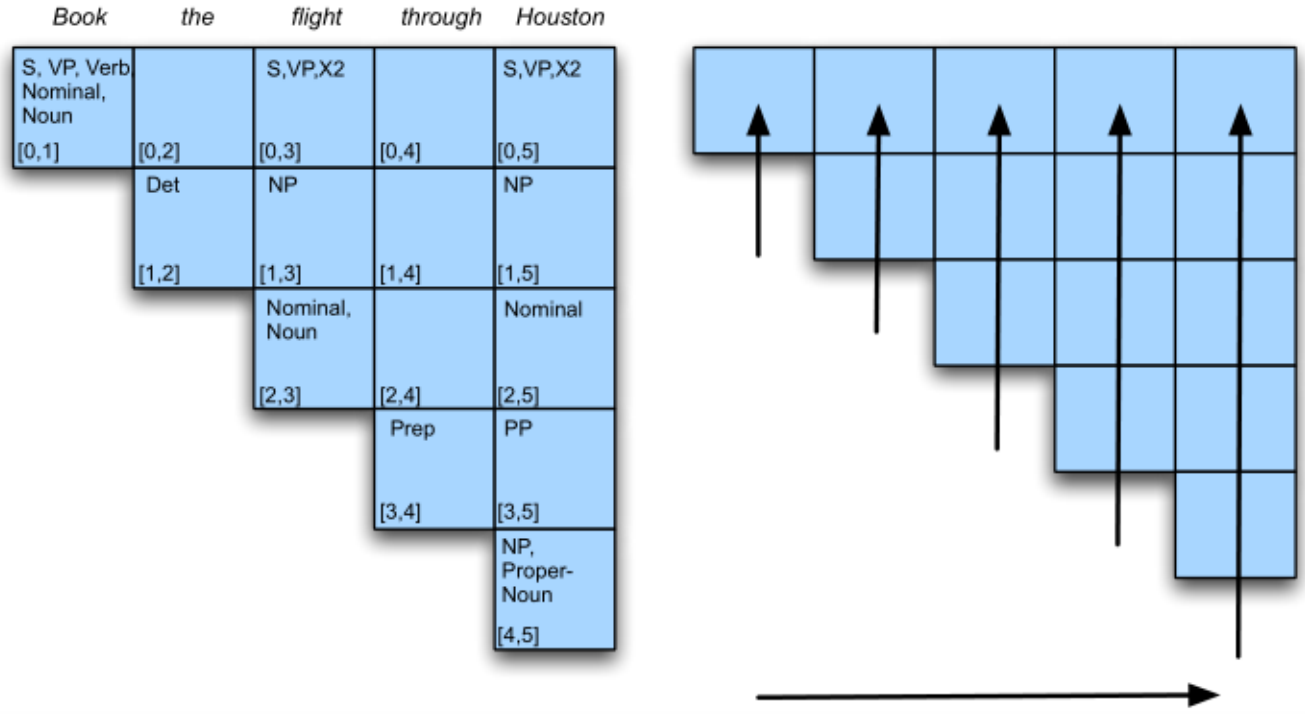
## Parser: algoritmo CKY (Cocke-Kasami-Younger)

- Basato sulla **programmazione dinamica**
- Algoritmo per **context-free**
- **Bottom-up**, left-to-right
- Euristica **rule-based**, ma solo per questo esempio

**IDEA**: Se tu hai delle regole context-free (tipo $A \longrightarrow BC$, $D \longrightarrow BC$) e avendo un sotto-albero con radice B e un sotto-albero con radice C, vanno entrambi bene per costruire un albero totale! Quando la **parte destra** è uguale, allora quello che ho calcolato per uno andrà bene anche per l'altro. Questi alberi pre-calcolati devono essere memorizzati dato che li riutilizzeremo fattorizzando la ricorsività.

**Prerequisito**: dobbiamo convertire una grammatica in **forma normale di Chomsky** (non esiste ad esempio una regola $A \to \epsilon$ ma solo $A \to BC$ e $A \to$ *terminal*)

**Matrice CKY**: tipo di matrice utilizzata per incrociare i sotto-alberi corrispondenti alle regole di derivazione (non dovremo ricalcolare tutte le volte ma solo andare a "trovare" nella matrice i *risultati parziali*)




Data la regola $A \to BC$:

- Se A "domina" dalla parola $i$ alla parola $j$ della nostra frase, allora
  - $\exists B$ per cui B "domina" dal punto $i$ a $k$ e tale per cui
    - $\exists C$ per cui C "domina" da $k$ a $j$.
- Costruiamo dei *cicli* su questi indici ed andiamo a salvare A in posizione $[i, j]$della tabella, B in $[i, k]$ e C in posizione $[k, j]$

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

**for** $j \leftarrow$ **from** 1 **to** LENGTH(*words*) **do**
  **for all** $\{A \mid A \rightarrow words[j] \in grammar\}$
    $table[j-1, j] \leftarrow table[j-1, j] \cup A$
  **for** $i \leftarrow$ **from** $j-2$ **down to** 0 **do**
    **for** $k \leftarrow i+1$ **to** $j-1$ **do**
      **for all** $\{A \mid A \rightarrow BC \in grammar$ **and** $B \in table[i,k]$ **and** $C \in table[k,j]\}$
        $table[i,j] \leftarrow table[i,j] \cup A$

**Notiamo che**:

- Quando l'**AND** delle tre condizioni finali corrisponde, allora abbiamo trovato il match corretto nella table
- Posso anche riempire con più di un oggetto la mia table, ed effettivamente per una cella avere *n* risultati

## Esempio

Dati:

- la stringa $w = baaba$
- la grammatica:
  - $S \rightarrow AB|BC$
  - $A \rightarrow BA|a$
  - $B \rightarrow CC|b$
  - $C \rightarrow AB|a$

Il ciclo sarà del tipo:

1. Partiamo dal fondo, riempiamo una colonna alla volta, dell'ultima riga della matrice: b possiamo ottenerlo solo dalla regola B, quindi mettiamo B. a possiamo ottenerlo dalla regola A,C quindi scriviamo A,C. Riscriviamo nella colonna successiva, e copiamo così via.
2. Guardiamo la sotto-stringa *ba*: $B \times A, C = (BA), (BC)$ (prodotto cartesiano praticamente). Cosa produce allora BA e BC? **A** ed **S**, scriviamo quindi S,A. Andiamo avanti così a coppie di due
3. Poi a coppie di tre nella riga ancora sopra *etc etc*...
4. Otteniamo una tabella:

| **S,A,C** | | | | |
|---|---|---|---|---|
| * | S,A,C | | | |
| * | B | B | | |
| A,S | B | S,C | A,S | |
| B | A,C | A,C | B | A,C |

Con passaggi intermedi:

- b = **B**
- a = **AC**
- ba = B x AC = BA,BC → **AS**
- aa = AC x AC = AA,AC,CA,CC → **B**
- ab = AC x B = AB, CB → **SC**
- baa = **None**
  - (ba)(a) = AS x AC = AA,AC,SA,SC → None
  - (b)(aa) = BB → None
- aab: **B**
  - (a)(ab) = AC x SC = AS, AC, CS, CC → B
  - (aa)(b) = BB → None
- aba: **B**
  - (a)(ba) = AC x AS = AA,AS,CA,CS → None
  - (ab)(a) = SC x AC = SA,SC,CA,CC → B
- baab: **None**
  - (b)(aab) = BB → None
  - (ba)(ab) = AS x SC → None
  - (baa)(b) = None x B → None
- aaba: **SAC**
  - (a)(aba) = AC x B = AB,CB → SC
  - (aa)(ba) = B x AS = BA,BS → A
  - (aab)(a) = B x AC = BA,BC → AS
- baaba: **SAC**
  - (b)(aaba) = B x SAC = BS,BA,BC → SA
  - (ba)(aba) = AS x B = AB,SB → SC
  - (baa)(ba) = None
  - (baab)(a) = None

## Complessità algoritmica

1. $O(n^3)$
2. $O(n^5)$ quando consideri anche le distribuzioni di probabilità

Troppo lento per real-time computation, allora si utilizza:

- Pruning probabilistico
- Parziale
- Dipendenze
- Un altro algoritmo

## Parsers: CKY probabilistico

- **Grammar**: context-free
- **Algorithm**: bottom-up, left-to-right
- **Memory organization**: dynamic programming (+*beam search*)
- **Oracle**: probabilistic

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

$G = (\Sigma, V, S, P)$
$A \to \beta[p] \quad p \in (0,1)$
$\sum_\beta P(A \longrightarrow \beta) = 1$

Nel caso medio è più veloce di $O(n^5)$ *sicuramente*: quando troviamo una probabilità minore di quella che stiamo considerando allora smettiamo di considerare probabilità più alte.

$P(T,S) = \prod_{node \in T} P(rule(N))$ La singola probabilità si calcola con il prodotto delle singole probabilità delle rule

**Da dove prendiamo le probabilità?** $\longrightarrow$ da *banche dati* come il **Penn Treebank**

## CKY probabilistico: training

Dato che abbiamo bisogno di: $P(a \to b|a)$
Quindi, per ogni albero nel TB: $P(a \to b|a) = \frac{Count(a,b)}{Count(a)}$

## Valutazione per le PCFG (Parseval)

- **Precision**: quanti sono i sotto-alberi che effettivamente appartengono anche al gold tree (albero target)? ($\frac{TP}{TP+FP}$)
- **Recall**: quanti sono i sotto-alberi del gold tree che appartengono anche al tuo albero? ($\frac{TP}{TP+FN}$)

## Lessicalizzazione delle context-free

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

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

Non vogliamo tagliare la pizza con la mozzarella, vogliamo condirla! Dovremmo togliere dall'albero la dipendenza della mozzarella dove dovrebbe esserci il senso del "tagliare con".
In questo caso allora ogni regola CF è potenziata con informazione circa le **teste delle regole**!

Come svantaggio avremo *molte più regole prodotte*...

## Parsers: parsing parziale (chunk parsing)

Approssimazione per creare un parser veloce: utilizziamo i **chunks** ed elimino **la ricorsività** per velocizzare il mio processo di parsing (assomiglia al **NER**, posso utlizzare l'algoritmo del BIO tagging per fare questo).

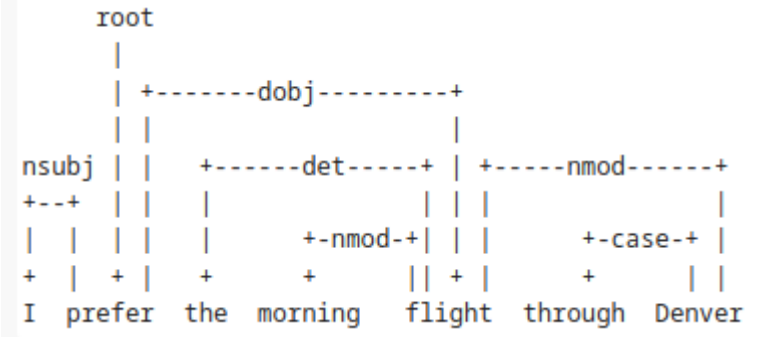
**Differenza chunk/sintagma**: sintagma è ricorsivo (può contenere diversi sotto-sintagmi al suo interno) mentre il chunk nominale non è capace di questa ricorsività, ma devi avere per forza diversi chunk per lo stesso concetto.

# Lezione 8: Parser a dipendenze e MALT

## Parsers: approccio a dipendenze

Termini possono essere **superiori od inferiori,** e quindi avere relazioni **master/slave**. L'output avrà un albero grande come il numero di parole ($n$), con $n-1$ dipendenze binarie (non c'è più il sintagma con tante parole). Anche questa si tratta di una approssimazione: non c'è simmetria con le **coordinazioni**.

Con le dipendenze posso fare information extraction **senza semantica**.
Nel caso delle grammatiche a dipendenze (*De Marneffe, Joakim Nivre*), a differenza delle grammatiche generative che hanno bisogno di competence, hanno uno **schema** più o meno formalizzati. Lo schema viene indotto (vd. *Machine Learning*) direttamente dalla frase, per questo hanno avuto così tanto successo.



## Vantaggi

- Non ho più bisogno della *x-bar teory*
- Può avere dipendenze asimmetriche/simmetriche

# Le 5 tecniche per il dependency parsing

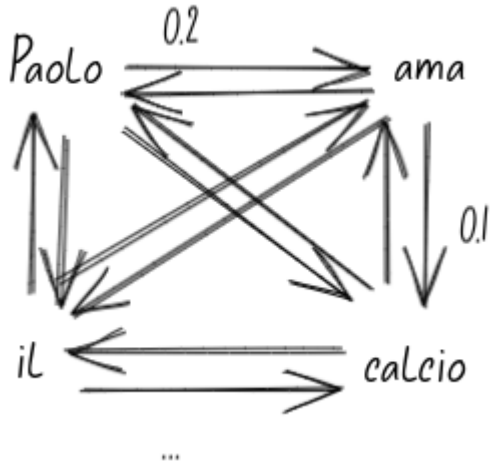
- Dynamic programming
- Graph algorithms
- Constituency parsing + conversion
- Deterministic parsing
- Constraint Satisfaction

## Dynamic Programming (CKY a dipendenze)

Algoritmo simile al parsing delle PCFG: $O(n^5)$ migliorato poi da *Eisner* a $O(n^3)$

## Graph Algorithms

- Creiamo il **minimum spanning tree** per la frase (grafo completamente connesso)
- Posso poi dare un **peso** ad ognuno degli archi




## Constituency Parsing

Parsing con grammatica a costituenti e converto in formato a dipendenze attraverso tabelle di percolazioni.

## Deterministic Parsing

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

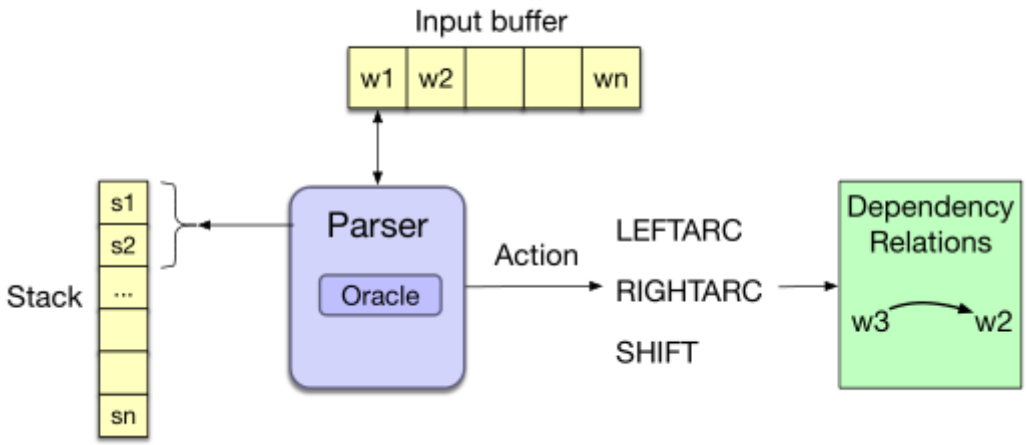
## Constraints Satisfaction

Elimino tutte le possibili dipendenze che non soddisfano certi vincoli.

# Parsers: MALT (Transition-based dependency parsing)

- **Grammar**: dependency grammar (si usa un automa dal TreeBank)
- **Algorithm**: bottom-up
- **Memory organisation**: depth-first (non deve fare *backtracking* dato che è probabilistico)
- **Oracle**: probabilistico

**IDEA**: Dato un *TB*, MALT induce il parser per il linguaggio del TB.




- **Ipotesi di proiettività**: *solo per questi esempi* utilizziamo alberi proiettivi, ovvero alberi con dipendenze **senza incroci**.
- Strutture usate:
  - Uno **stack** per le parole **parzialmente analizzate** (*stack*)
  - Una **lista** contenente le **rimanenti parole** da analizzare (*input buffer* o *word list*)
  - Un **insieme** contenente le **dipendenze** create fino a quel punto dell'analisi (*relations*)

Il parser attraversa la frase *da sinistra a destra*, successivamente sposta gli item dal buffer allo stack. Ad ogni passo analizziamo i 2 elementi in testa allo stack e l'oracolo fa una decisione sulla **transizione** da applicare per costruire il parser. Le possibili transizioni corrispondono alle azioni intuitive che uno dovrebbe fare per creare un dependency tree esaminando le parole in una sola passata sull'input da sinistra a destra:

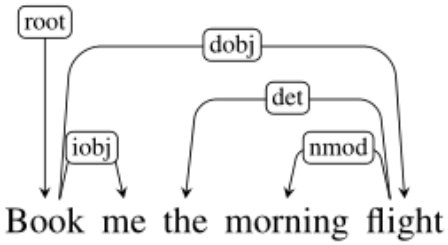
- Assegna la parola corrente alla testa di qualche altra word vista precedentemente
- Assegna qualche parola vista precedentemente come testa della parola corrente
- Postpone l'azione sulla parola corrente, fai storage per processing successivo.

```
function DEPENDENCYPARSE(words) returns dependency tree

  state ← {[root], [words], [] }  ; initial configuration
  while state not final
    t ← ORACLE(state)         ; choose a transition operator to apply
    state ← APPLY(t, state)   ; apply it, creating a new state
  return state
```

Queste operazioni si traducono direttamente in comandi come:

- **Shift**: prende la parola dall'input buffer e fa push sullo stack
- **Left** o **LEFTARC**: crea una dipendenza $(a, b)$ tra la prossima parola della lista $(a)$ e la parola sul top dello stack $(b)$, poi poppa lo stack.
- **Right** o **RIGHTARC**: Il contrario. Crea la dipendenza $(b, a)$ tra la prossima parola della lista $(a)$ e la parola sul top dello stack $(b)$, rimuove $(a)$ dalla lista e poi poppa lo stack e mette la parola poppata all'inizio della lista.




(14.5)

Let's consider the state of the configuration at Step 2, after the word *me* has been pushed onto the stack.

| Stack | Word List | Relations |
|---|---|---|
| [root, book, me] | [the, morning, flight] | |

The correct operator to apply here is RIGHTARC which assigns *book* as the head of *me* and pops *me* from the stack resulting in the following configuration.

| Stack | Word List | Relations |
|---|---|---|
| [root, book] | [the, morning, flight] | (book → me) |

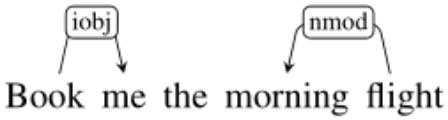
After several subsequent applications of the SHIFT and LEFTARC operators, the configuration in Step 6 looks like the following:

| Stack | Word List | Relations |
|---|---|---|
| [root, book, the, morning, flight] | [] | (book → me) |

Here, all the remaining words have been passed onto the stack and all that is left to do is to apply the appropriate reduce operators. In the current configuration, we employ the LEFTARC operator resulting in the following state.

| Stack | Word List | Relations |
|---|---|---|
| [root, book, the, flight] | [] | (book → me)<br>(morning ← flight) |

At this point, the parse for this sentence consists of the following structure.




(14.6)

C'è **una sola sequenza possibile** per la *costruzione* dell'albero della frase.

## Problema 1: operatori e dipendenza

- $n$ dipendenze $\implies 2n + 1$ operatori
- creiamo allora un $left\_subj$ e $right\_subj$ (ed insomma un right_relazione, left_relazione) per risolvere il problema

## Problema 2: oracolo

Da dove apprendo shift, left e right? Non c'è scritta da nessuna parte la sequenza di operazioni da fare. Io voglio apprendere un classificatore per ogni transizione. Per farlo dal Treebank, utilizzo ML per capire le feature che caratterizzano ogni stato. Ciò comporta:

- Scoprire features linguisticamente significative
- Costruire il training data, attraverso $treebank \implies transitionbank$ (qua solitamente si usa l'universal dependency treebank, poi si fa **reverse engineering**, ovvero data la frase e l'albero finale, vado a trovare le regole che portano la frase nell'albero)
- Utilizzare un buon training algorithm

## Scoring function

Utilizzo una s.f. $S(c,t) = w \cdot f(c,t)$

# Parsers: TUP

- **Grammar**: dependency grammar (Constraints)
- **Algorithm**: bottom-up
- **Memory organisation**: depth-first
- **Oracle**: rule-based

Per valutare prestazioni: **labelled attachment score**

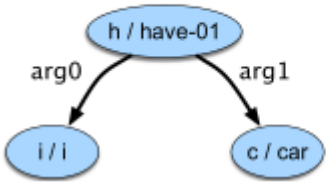
# Parsers: tabella riassuntiva

| Parser | Grammatica (Competence) | Algoritmo (Performance) | Oracolo (Oracle) | Complexity |
|---|---|---|---|---|
| CKY | Context-free | Bottom-up, Left-to-right, dynamic programming | Rule-based | $O(n^5)$ fino a $O(n^3)$ |
| CKY-prob | Context-free | Bottom-up, Left-to-right, dynamic programming | Probabilistic | < $O(n^5)$ |
| MALT | Dependency-grammar (automa dal TB) | Bottom-up, depth-first, no-backtracking | Probabilistic | * |
| TUP | Dependency-grammar | Bottom-up, depth-first, constraint-based | Rule-based | * |

# Lezione 9: intro al livello semantico e formale

In questa lezione tratteremo la rappresentazione logica del significato della frase.

$$\exists e, y\ Having(e) \wedge Haver(e, Speaker) \wedge HadThing(e, y) \wedge Car(y)$$




```
(h / have-01                    Having:
      arg0: (i / i)                   Haver:  Speaker
      arg1: (c / car))                HadThing:  Car
```

Possiamo rappresentare la stessa frase "*I have a car*" in diversi modi possibili: FOL, AMR e frame-based sono solo alcuni dei modi in cui posso rappresentarne il significato. Il nostro linguaggio di rappresentazione in questi esempi sarà sempre la logica del prim'ordine, ovvero "*Paolo ama Francesca*" $\implies love(Paolo, Francesca)$ ovviamente non sapremo cosa vuol dire "amare" oppure $love$ ma in questo senso ci interessa la rappresentazione, non come verrà effettivamente fatto. Per questa motivazione introdurremo il lambda-calcolo, per risolvere il problema del calcolo della semantica. Lo facciamo principalmente perché poi vorremmo fare inferenza sulla nostra KB.

Qua salto la spiegazione di cosa sia la FOL dato che già si sa.

## Semantica di Montague

- **Teoria semantica in stretta relazione con la sintassi**: si basa sull'idea che non esista una gran differenza teoretica tra il linguaggio naturale ed il linguaggio artificiale logico. Possiamo usare principi matematici e regole logiche per analizzare qualsiasi lingua.
- Si basa sul principio di composizionalità

## Computational Semantics Fundamental Algorithm

Algoritmo che si basa sul **principio di composizionalità di Frege**: il significato del tutto è determinato dalle parti e dalla maniera in cui sono combinate.

- Parsifico la frase per ottenere l'albero sintattico (dei costituenti)
- Cercare la semantica di ogni parola nel lessico
- Costruire la semantica per ogni sintagma in maniera bottom-up/syntax-driven

Dobbiamo però:

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

# Lambda calcolo

Per risolvere i due problemi ci serve introdurre il lambda calcolo e le astrazioni lambda. L'operatore $\lambda$ viene usato per legare le variabili libere: $\lambda x.\, love(x, Francesca)$ vuol dire "amare Francesca".

In Python una lambda si rappresenta come:

```
lambda x: x % 2 == 0
```

Come facciamo per legare $\lambda$-FoL e FoL?

## Beta reduction

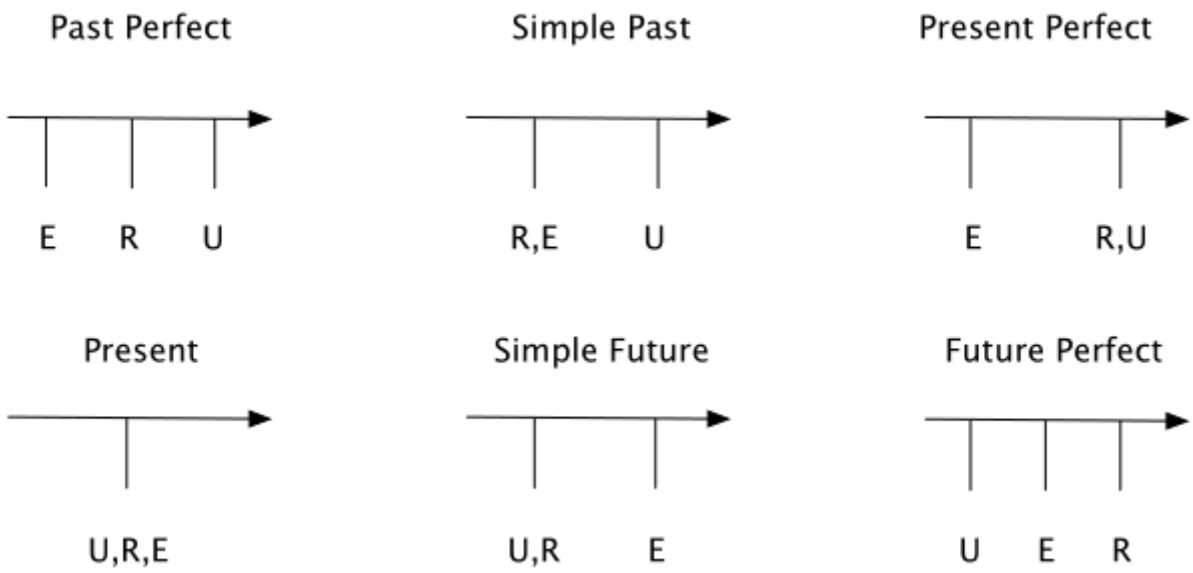
Trasforma una lambda function in una formula FoL
$(\lambda x.\, love(x, mary))(john)$

1. Elimino il lambda: $(love(x, mary))(john)$
2. Rimuovo l'argomento a dx: $love(x, mary)$
3. Rimpiazza tutte le occorrenze della variabile legata dal lambda con l'argomento in tutta la formula: $love(john, mary)$

**L'ordine dei lambda** è fondamentale per capire il senso degli argomenti!
Per fare ciò nell'albero delle dipendenze dobbiamo anche capire quale sia la funzione e quale sia argomento, l'approccio storico è quello di compilare una lista di **regole di composizione**.

## Rappresentazione del tempo

Possiamo anche rappresentare il tempo nel seguente modo:




Dove:

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

Esempio: *I will eat* ha la utterance e la reference ora e l'evento nel futuro.

## FOL: come trattare gli articoli (Reverse Engeneering)

Dato che vogliamo arrivare ad avere sulla radice una formula in logica, dobbiamo anche convertire *un uomo* esplicitando anche la parte "**un**" ( $DT : \lambda P.\, \lambda Q.\, \exists z(P(z) \wedge Q(z))$) e poi faccio una beta reduction sulla formula trasformata.

## FOL: come trattare i nomi propri (Type-Raising)

La semantica di una costante quando si comporta come un soggetto fa un type-raising: ovvero è un predicato p anonimo che si applica alla variabile Paolo. Ciò risolve il problema dei nomi propri come soggetto. Ciò inverte la direzionalità di chi sta a sinistra e chi sta a destra (*pagina 282*).

Regola:
$X \rightarrow T/(T\backslash X)$
$X \rightarrow T\backslash(T/X)$

Nell'esempio:
$paolo \Longrightarrow \lambda P.\, P(paolo)$

## FOL: come trattare i verbi transitivi (Type-Raising)

Una volta che abbiamo fatto type-raising sui nomi propri, non possiamo permetterci di mantenere la forma

$$V : \lambda y.\, \lambda x.\, love(x, y)$$

Perché, sostituendola con $\lambda P.\, P(francesca)$ otterremmo una **ricorsione infinita** di sostituzioni, ovvero

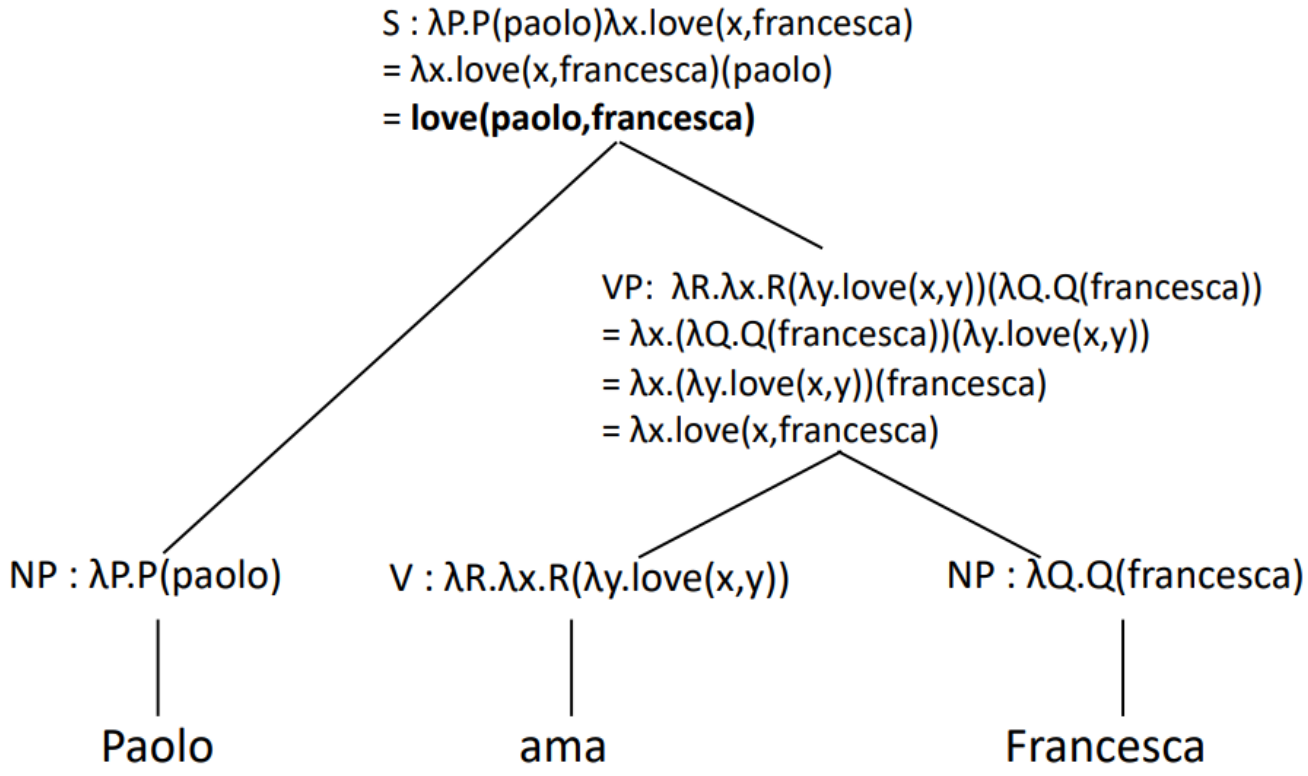
$$(\lambda x.\, love(x, \lambda P.\, P(francesca)))$$

Effettuiamo quindi il type-raising della forma verbale, ottenendo

$$\lambda R.\, \lambda x.\, R(\lambda y.\, love(x, y))$$

## Esempio "Paolo ama Francesca" completo

- Abbiamo fatto type-raising sul verbo ama, per permettere la sostituzione su Francesca, che altrimenti sarebbe errata (problema dei nomi propri come soggetto) $\longrightarrow$ Invertiamo perciò:
  - $\lambda x.\, (\lambda Q.\, Q(francesca))(\lambda y.\, love(x, y))$
  - $\lambda x.\, (francesca)(\lambda y.\, love(x, y))$ (**type raising**, altrimenti *ricorsione infinita*)
  - $\lambda x.\, (\lambda y.\, love(x, y))(francesca)$
- **Proprietà di sistematicità**: Non vogliamo delle regole ad-hoc, ma vogliamo **sistematicità** nelle nostre regole (visto che abbiamo cambiato la forma del nome, dobbiamo cambiare il verbo e così via!)




## Ambiguità logica vs sintattica

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

- **Ambiguità logico-semantica**: stesso albero sintattico, significato *ambiguo*

1. $\forall x(man(x) \rightarrow \exists y(woman(y) \wedge love(x,y)))$
2. $\exists y(woman(y) \forall x(man(x) \rightarrow love(x,y)))$

- **Ambiguità sintattica**: abbiamo possibili alberi sintattici: *mangiare la pizza con le acciughe*!

Esprimiamo **every** come:

$$\lambda Q \lambda P(\forall x(Q(x) \rightarrow P(x))) \rightarrow Every$$

> La CCG funziona molto bene per queste cose, invece.

## Reificazione (Avverbi come eventi)

Come ci comportiamo quando dobbiamo parsificare un **avverbio**? Reifichiamo un evento. Ovvero definiamo una variabile che identifica un evento!

**Esempio**: *"dolcemente"*
$\exists e\ love(e,P,F) \wedge sweetly(e)$ *(stile classico)*
$\exists e\ love(e) \wedge agent(e,P) \wedge patient(e,F) \wedge sweetly(e)$ *(stile neo-Davidsonian)*

## Groningen Meaning Bank

- Risorsa di ampia copertura creata da un'estrazione a mano + classificazione automatica
- Usata per addestrare sistemi statistici
- Link per accedere alla risorsa
- Rappresentazione che si costruisce su CCG

# Lezione 10: Semantic banks e generazione del linguaggio

## Generazione del linguaggio

- Processo di **costruzione deliberativa** di testo in linguaggio naturale al fine di perseguire un goal comunicativo.
- **Planning** (pianificazione) come concetto centrare della NLGeneration
- Nella maggior parte dei casi un insieme prefissato di frasi può essere sufficiente per produrre risposte comprensibili e naturali (Lesmo), ma la verità è che esistono

problemi diversi con vincoli diversi (ad es. NLG su smartphone, su orologio...)! Devo perciò gestire il fatto che ho tante strade per gestire la stessa cosa.
- Esempio utile: *BabyTalk*

## NLG vs. Templates

1. **NLG**:
    1. Mantainability
    2. Text quality (utilizzo di ricorsività, vicinanza al vero linguaggio naturale)
    3. Output multilingue
2. **Templates**:
    1. Poca formalizzazione (es. printf)
    2. Non ho bisogno di linguistica
    3. Veloce!

## Fasi architetturali standard NLG

I numeri *(n)* sono in riferimento ai Task NLG:

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

- Ognuna delle strutture dati è output della fase precedente e input della fase successiva
- Strumenti di valutazione sono più importanti nel NLG piuttosto che in un qualsiasi altro task NLP!

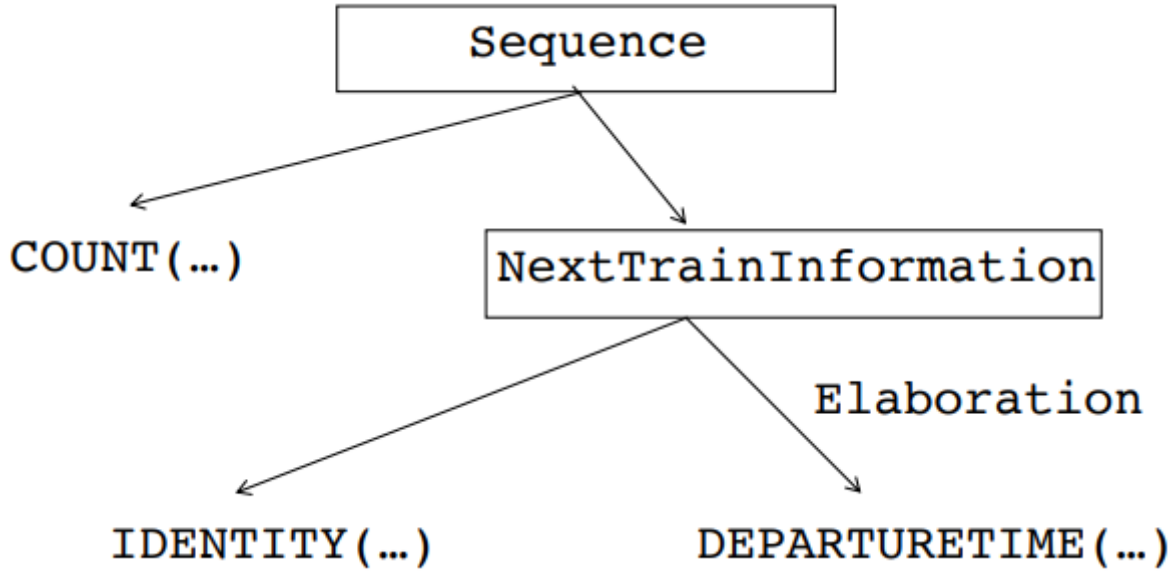
## Task NLG

1. **Content Determination**: decidere cosa voglio scrivere. Dati dei dati, generare del contenuto.
    - Messaggi provenienti da una struttura dati
    - Messaggi basati su entità
    - Esempio: $IDENTITY(NEXTTRAIN, CALEDONIANEXPRESS) \rightarrow$ *The next train is the Caledonian express*
2. **Discourse Planning**: che relazione c'è tra i pezzi di discussione che voglio comunicare? Abbiamo due tipi di relazioni:
    - Conceptual grouping
    - Rhetorical Relationships
3. **Sentence Aggregation**: in quante frasi devo comunicare il mio contenuto?

- Un mapping 1-a-1 dai messaggi alle frasi produrrebbe un testo poco naturale, non fluente
- Messaggi devono essere combinati per produrre frasi complesse e naturali
- Risultato: **sentence specification** o **sentence plan**
- **Esempio senza aggregazione**: *The next train is the Caledonian Express. It leaves Aberdeen at 10 am.*
- **Esempio con aggregazione**: *The next train, which leaves at 10 am, is the Caledonian Express*

4. **Lexicalisation**: scelgo quale struttura dare alla mia frase (sia come struttura generale che come struttura dati utilizzata)
    - La lessicalizzazione determina **quali parole e quali relazioni sintattiche** usare per esprimere i concetti del dominio.
    - Esempio: la frase deve essere attiva oppure passiva? Meglio "il cane morde l'uomo" oppure "l'uomo è morso dal cane"? Vogliamo una certa **nuance** di significato.
5. **Referring expression generator**: bilanciare fluenza ed ambiguità $\Longrightarrow$ quali parole usare per esprimere il concetto?
6. **Syntactic and morphological realisation**: Dato come input l'albero sintattico a dipendenze genero la corretta sequenza morfologica rispettando la grammatica della lingua naturale selezionata (come se fosse un parser al contrario, opposto al fare la lemmatizzazione)
    - Esempio sulle regole morfologiche: dato che devo dare il passato alla frase, metto *-ed* dopo *walk* per fare *walked*
    - Esempio sulle regole sintattiche: Il soggetto va messo prima del verbo $\rightarrow$ *John walked* meglio di *walked John*.
7. **Orthographic Realization**: scelgo punti, spazi... ortografia insomma.
    - Seguo le regole ortografiche: le frasi cominciano con la lettera grade, le dichiarative finiscono con un punto, per le domande ci va il punto interrogativo...

**Discourse Planning Schema Example**:



# Lezione 11: Sistemi di dialogo e Chatbots

- Dialogo $\neq$ Machine Translation. Solitamente sistemi come Siri hanno sia chatbot functionality che sistemi di dialogo e machine translation insieme.

## Dialogo

- **Attività collaborativa**
- Può essere uomo-uomo: ricezione, ragionamento e risposta a battute (pragmatica, semantica, sintassi e morfologia è data dalla persona).
- Uomo-macchina: può usare dei "cheat" tipo → Quando trovo la parola "ama", print "ah, l'amore!".

### Key features del dialogo

- Ha dei **turni**
- I turni si organizzano in unità basiche di comunicazione chiamate **atti di dialogo**
- L'atto del dialogo è soggetto al **contesto conversazionale**
- I partecipanti al dialogo danno sempre dei **segnali di grounding** durante la conversazione

### Turn-taking

Capire quando tocca a te a parlare, ci sono dei segnali:

- Silenzio
- Segni di esitazione
- Sintassi
- Intonazione
- Intensità
- Body language

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

### Dialogue Acts

Hanno una natura totalmente diversa a seconda del contesto!

- Assertiva
- Imperativa
- Commissiva (fare un commitment ad un'azione futura)
- Espressiva
- Dichiarativa

### Implicazione

Quando implichi qualcosa dal contesto, senza dirlo, mentre parli di qualcosa.
Questo viene a costruire il cosiddetto **common ground**.

# Architetture per Chatbots

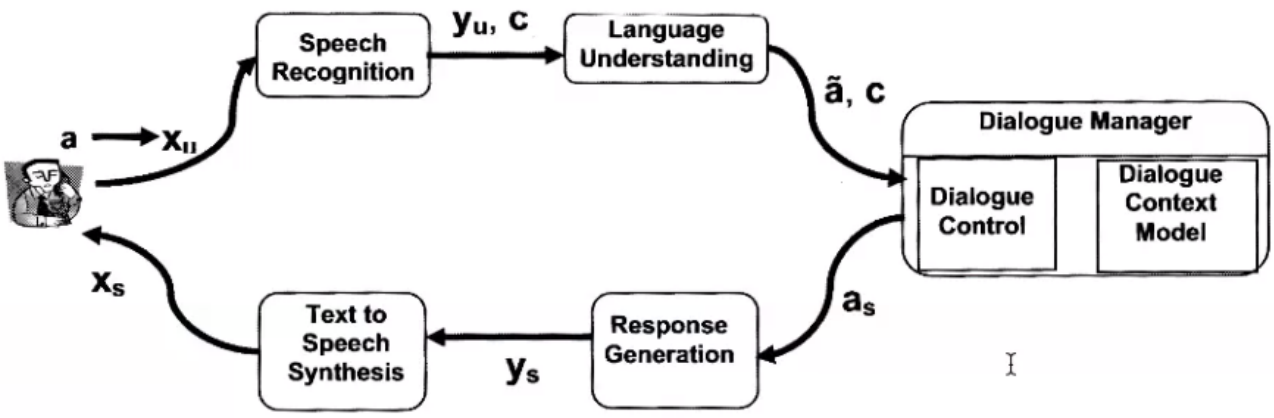
- Rule-based:
  - Pattern-action rules (**Eliza**)
  - Eliza+ a mental model
  - ALICE
- Corpus-based (da un chat corpus enorme)
  - Information Retrival
  - Neural network encoder-decoder

### Eliza algorithm

**function** ELIZA GENERATOR(user *sentence*) **returns** *response*

Find the word *w* in *sentence* that has the highest keyword rank
  **if** *w* exists
    Choose the highest ranked rule *r* for *w* that matches *sentence*
    *response* ← Apply the transform in *r* to *sentence*
    **if** *w* = 'my'
      *future* ← Apply a transformation from the 'memory' rule list to *sentence*
      Push *future* onto memory stack
  **else** (no keyword applies)
    **either**
      *response* ← Apply the transform for the NONE keyword to *sentence*
    **or**
      *response* ← Pop the top response from the memory stack
  **return**(*response*)

# Architettura dei sistemi a dialogo




- $x_u$ : user acoustic signal
- $y_u$ : speech recognition hypothesis (words)
- **a** : user dialogue act (intended)
- **ã** : user dialogue act (interpreted)
- $a_s$ : system dialogue act
- $y_s$ : system word string
- $x_s$ : system acoustic signal
- **c** : confidence SR

# NLU: natural language understanding

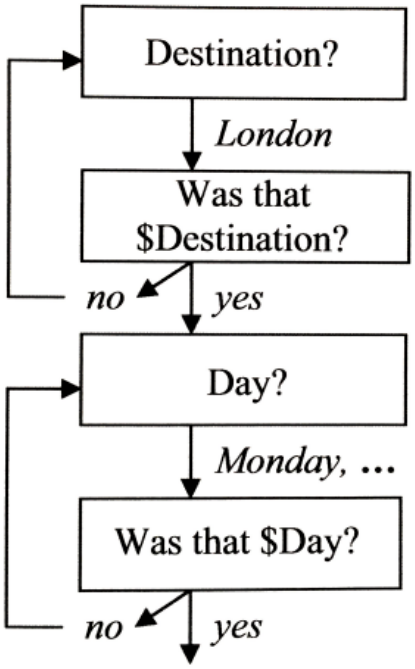
- **Classica**: meaning = sintassi + semantica + pragmatica (tutto quello che abbiamo fatto fino ad ora)
- **Moderna**: semantica semplificata (GUS Semantics: attraverso una struttura chiave valore tu hai dei dati tipati di tipo (Slot,Type), ad es. (ORIGIN CITY, CITY)) + Ontologia di dominio OPPURE utilizzo di sistemi statistici + ML su dialoghi già annotati (more is better, stack more layers).

# Dialogue Manager

# Graph-based Dialogue Control

- Suitable only for simple interaction
  - One can adapt ASR
- Nodes and transitions grow fast
  - 5 items with 3 values -> $3^5$
- Center for Spoken Language

Understanding (CSLU) toolkit




# Frame Based Control

| | | |
|---|---|---|
| ❑Destination: unk<br>❑Date: unk<br>❑Time: unk | S: *Where are you flying to?*<br>U: London → | ❑Destination: London<br>❑Date: unk<br>❑Time: unk |

- User's utterance **can include various permutation**
- Over-answering: *London at 9 a.m.*
- Complex grammar:
  - Destination
  - Destination + Date
  - Destination + Time
  - Destination + Date + Time

# Problems of Frames

- Not easily applicable to complex task
  - May not be a single frame
  - Dynamic construction of

-> **Travel Plans**

  - Goal: get from Paris to Berlin
  - Options: fly, train, drive
  - Flight: airline, airport, price, date
  - Train: station, class, discount?, reservation?
  - Drive: directions, fastest or cheapest?
- Complex Frame
  - Frame hierarchy
  - Agenda

# Information State Approach: static

1. **Informational components** are a functional specification of the aspects of context (e.g. structures, principles) that are part of the dialogue model.
2. **Formal representations** are data structures, including accessibility relations for each of the informational components. For example, a history of previous utterances might be represented as a simple list, while a set of topics might be represented as a stack. (***RECURSIVE STRUCTURE!***)

# Information State Approach: dynamic

3. **Dialogue Moves** are abstractions of the kinds of utterance information that is seen as relevant for updates. This could be the same sort of speech acts or dialogue acts as used by many approaches.
4. **Update rules** specify how the informational components change as a result of observation and processing of dialogue moves. For example, once a question has been answered, the question is no longer a motivating factor requiring an answer
5. **Update Strategy** is a method for deciding which rules to apply when.

# Domande probabili

- I 5 Algoritmi di parsing delle grammatiche a dipendenze.
- A cosa serve il lambda calcolo? Come rappresentarvi articoli e sostantivi?
- CKY: spiegazione, a cosa serve, simulazione su carta.
- Ambiguità sintattica (PP attachment e Coordination, con esempi)
- Differenze tra i task di *Referencing Expression* e *Lessicalizzazione* in NLG.
- Problema dell'espressività delle lingue naturali, Chomsky, gerarchia e delle grammatiche Mildly Context Sensitive
- Differenza fra HMM e MEMM: spiegazione dei pro e dei contro, differenze a livello di probabilità utilizzate.
- Cos'è una grammatica CCG?
- Nell'architettura finale dell'NLG, perché abbiamo visto he è divisa in 3 fasi? Me ne parli.
- Anatomia di un parser
- Semantica formale e composizionale con esercizio "Paolo ama Francesca" (Montaque) e tipi ambiguità con esempi
- Semantica di Montague (fol + lambda calcolo) e la sintassi a dipendenze
- Parseval: valutazione per le PCFG (precision e recall)