

# Lezione21

## Table of contents

- [PROLOG e Linguaggio Naturale](#)
  1. [`nlp1.pl`](#)
  2. [`nlp2.pl`](#)
  3. [`nlp3.pl`](#)
  4. [`nlp4.pl`](#)
  5. [`chatbot.pl`](#)

## PROLOG e Linguaggio Naturale



Alain Colmerauer (1941-2017)

La programmazione logica non nasce nel tentativo di realizzare agenti che ragionino e decidano cosa fare sulla base di deduzioni razionali, ma nasce in Francia nel tentativo di realizzare sistemi di **Natural Language Processing** (NLP). Questi sistemi, realizzati e sperimentati per la prima volta dall'informatico francese Alain Colmerauer, sono in grado di estrarre informazione dal linguaggio naturale e utilizzare queste informazioni per produrre risposte o estrarre dati.

PROLOG assomiglia molto a un linguaggio per descrivere grammatiche, come le forme di Backus-Naur (BNF): tuttavia, le grammatiche non contestuali che conosciamo, non sono abbastanza per il linguaggio naturale siccome questo è ambiguo. Il lavoro cominciato in Francia, ha portato a una versione delle NLP che oggi chiamiamo **classiche** o **simboliche**: lo strumento che rende operativo il riconoscimento del linguaggio è la grammatica.

Chi si occupa di NLP, al giorno d'oggi usa versioni più moderne, dette NLP **sub-simboliche**, utilizzando delle reti neurali o simili, in grado di associare un significato/intento a una frase.



Avram Noam Chomsky (1928)

Seguendo quello detto dal "padre della linguistica moderna" Chomsky, il problema delle NLP è sicuramente quello di spezzare una frase in parti costitutive che permettano di verificare la correttezza e che permettano di

attribuire un significato: identificazione delle *parti del discorso* (POS). Se riusciamo ad attribuire un significato a ogni POS, allora riusciremmo ad attribuire significato alla frase, ragionando una frase alla volta.

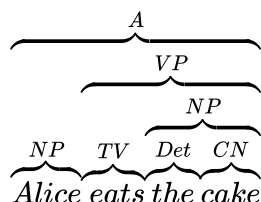
Ogni sequenza contigua di parole della frase, prende il nome di *sintagma* (o phrase).

Non è detto che una sezione abbia senso, per questo vengono classificate in categorie diverse, guardando la 1<sup>a</sup> parola rilevante che identifica l'inizio del sintagma, la testa, andando avanti in modo di costruire un *albero sintattico*. I sintagmi a loro volta, possono essere classificati:

- in *sintagmi nominali* (noun phrase, NP) in cui la testa è un nome comune o proprio, o un pronome;
- in *sintagmi verbali* (verb phrase, VP) in cui la testa è un verbo;
- in *sintagmi preposizionali* (prepositional phrase, PP) in cui la testa è una preposizione.

Dal punto di vista linguistico, si sostiene che i linguaggi naturali si sono sviluppati utilizzando questi sintagmi specifici, per come è fatto il nostro cervello, che ha struttura fisica tale per cui questa tipologia di linguaggi naturali con questi sintagmi, vengono privilegiati. Nel nostro caso, vedremo soltanto l'applicazione all'inglese, comunque traducibile in italiano molto semplicemente.

La frase "*Alice eats the cake*" può essere ricondotta a un albero sintattico, che preso in sintagmi singoli può risultare ambigua. Questo non è da preoccuparci però, siccome la grammatica stessa, senza contesto, rende una frase ambigua di natura.



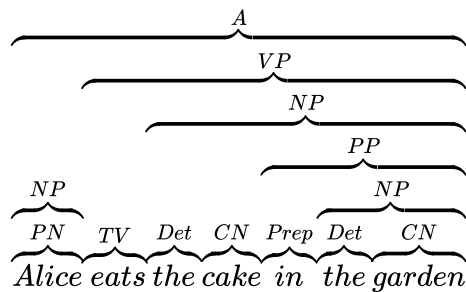
- *A* indica che la frase è un'*asserzione*;
- *TV* (transitive verb) indica un *verbo transitivo*, siccome *eats* è una sua voce;
- *Det* (determiner) indica un *determinante*, come "*a*", "*this*", "*that*" e altri;
- *CN* (common name) indica un nome comune.

Potremmo vedere degli esempi ulteriori, costruendo una grande esemplificazione della grammatica inglese, in formato BNF

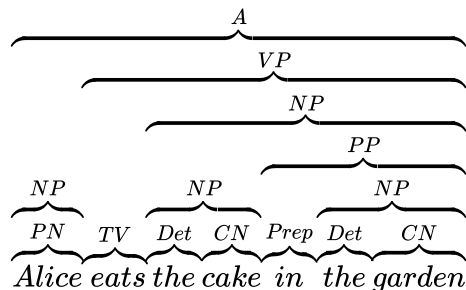
```
A  → NP VP
NP  → PN | Det CN PP*
VP  → IV PP* | TV NP PP*
PP  → Prep NP
Det  → {the}
Prep → {in, near}
PN   → {Alice, bob, London}
CN   → {book, cake, garden, house, kid, lake}
TV   → {eats, reads}
IV   → {lives, runs}
```

che tuttavia non elimina ambiguità.

La frase "*Alice eats the cake in the garden*" può significare che "Alice mangia la torta che si trova in giardino"



oppure che "Alice mangia in giardino la torta"



Avendo in mano la grammatica, un programma PROLOG può essere composto, allo scopo di analizzare queste frasi: non saremo tuttavia in grado di eliminare l'ambiguità e potremo accettare frasi senza senso.

 ChatGPT



Sviluppato da OpenAI e rilasciato nel Novembre del 2022, ChatGPT è un bot di comunicazione, intelligente, in grado di rispondere a quesiti o richieste, o comunque, capace di rispondere in linguaggio naturale, usando un trasformatore generativo pre-addestrato (GPT) come fondamento del Large Language Model (LLM).

A differenza di quello che si potrebbe pensare, ChatGPT e le sue ulteriori versioni (originariamente GPT-3.5, dal Marzo 2023 GPT-4), non capisce per davvero quello che gli viene detto, ma risponde soltanto nel modo che gli sembra più opportuno, basandosi sui dati con cui è stato addestrato, usando statistica e calcoli per determinare quale risposta fornire.

Dataset	# <a href="#">tokens</a>	Proporzione
<a href="#">Common Crawl</a>	410 miliardi	60%
WebText2	19 miliardi	22%
Books1	12 miliardi	8%
Books2	55 miliardi	8%
Wikipedia	3 miliardi	3%

nlp1.pl

 [Vedere il programma](#) nlp1.pl

Il primo esempio che vediamo è quello di un analizzatore sintattico, chiamato anche *parser*. Abbiamo una clausola per ogni produzione della grammatica e per ciascuna, ci sarà clausola con 2 argomenti:

1. argomento sequenza di token d'analizzare;

2. la sequenza di token la cui analisi non è andata a buon fine.

L'analizzatore avrà lo scopo di dirci se la frase chiesta è corretta grammaticalmente e se è accettabile (accettore di linguaggi). Per ogni possibilità non deterministica, l'analisi cambierà di precisione se necessario.

Questo approccio alle grammatiche, prende il nome di *difference list*, perché appunto si parte da una lista più lunga e si arriva a un certo punto con una parte finale: s'identificano 2 sezioni per cui la 1<sup>a</sup> viene consumata dalla clausola mentre la 2<sup>a</sup> è ciò che rimane.

```
a(I, R) :-  
    np(I, R1),  
    vp(R1, R).  
% ...
```

La prima clausola fa riferimento al non terminante `a` (affermazione), ha 2 argomenti: la lista di token che chiamiamo `I` e il resto, tutti i token che rimangono dopo aver trovato un'asserzione, `R`. Osservando la grammatica, sappiamo che un'asserzione deve iniziare con un *NP* `np` (prende tutta la frase), lasciando da parte un `R1`. A seguire c'è un *VP* `vp`, che prenderà tutto quello rimasto dopo aver tolto *NP* `np`, andando avanti finché deve, lasciando indietro un resto `R`.

```
% ...  
np([PN | R], R) :-  
    pn(PN).  
np([Det, CN | R1], R) :-  
    det(Det),  
    cn(CN),  
    optpps(R1, R).  
% ...
```

Un `np` è un nome proprio (1<sup>a</sup> clausola) che comincia con un `pn` e va avanti con un `R`, il resto in uscita. I `pn` vengono elencati sotto. Esempi sono "*Alice*", "*Bob*" e "*London*": se trovo uno di questi, quello che viene dopo è un resto.

Nel secondo caso, abbiamo un determinatore `Det`, un nome comune `CN` (elencati sotto con `cn()`), una sequenza opzionale di proposizioni `optpps`.

```
% ...  
optpps(R, R).  
optpps(I, R) :-  
    pp(I, R1),  
    optpps(R1, R).  
% ...
```

`optpps` è formata da 0 proposizioni quando la lista che arriva è anche resto `optpps(R, R)` (non togliamo niente), oppure c'è un sintagma proposizionale `pp` seguito da 0 o più sintagmi proposizionali `optpps`.

```
% ...  
vp([IV | R1], R) :-  
    iv(IV),  
    optpps(R1, R).  
vp([TV | R1], R) :-  
    tv(TV),  
    np(R1, R2),
```

```
optpps(R2, R).  
% ...
```

Nel nostro caso un verbo intransitivo è fatto da una parola sola, "*lives*" o "*runs*".



Il verbo intransitivo **IV** è seguito da **R1**, quest'ultimo possibilmente una sequenza di più sintagmi proposizionali. A seconda di quanto prendiamo lunga la frase, avremo un resto **R**.

Il sintagma con invece il verbo transitivo **TV** ha un nome subito dopo, che preso avrà poi una sequenza di sintagmi proposizionali, interna ad **R2**, generante resto **R**.

Il goal

```
a([Alice, eats, the, cake], R).
```

chiede se quello che stiamo scrivendo "*Alice eats the cake*", è corretto, mostrando quella parte di frase che non è inseribile nell'asserzione.

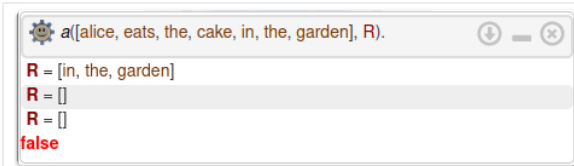
 Il risultato della prima esecuzione ("Run!")	 No scelte non deterministiche ("Next")
---	--

Il goal

```
a([Alice, eats, the, cake, in, the, garden], R).
```

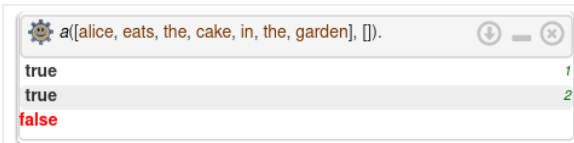
è ambiguo e può avere più interpretazioni.

1. "*Alice eats the cake*" è la prima interpretazione alla quale per forza arrivavamo. Il resto **R** è uguale a "*in the garden*".
2. La sequenza di proposizioni può essere vuota. Se andiamo avanti, avremo un resto **R** che è stringa vuota con "*in the garden*" ora parte dell'asserzione.
3. La frase viene presa per intero, ma esiste un'eventuale interpretazione della frase, compatibile con l'albero sintattico. Infatti, abbiamo visto più in alto, che per i 2 alberi sintattici ci sono corrispondenze non deterministiche.

 Le 3 possibili interpretazioni
--

La cosa che più ci da fastidio è il fatto che tutta la frase all'inizio non l'ha presa, quando poteva essere fatto. Per risolvere possiamo specificare all'esecutore di prendere tutte le interpretazioni con frase intera; quel modo di leggere le frasi dove nulla rimane.

```
a([Alice, eats, the, cake, in, the, garden], []).
```

 Delle 3 sequenze, ne prendiamo solo 2
---

Dettagli linguistici quali l'aggiunta delle versioni plurali e delle congiunzioni dei verbi, porta all'aumento rapido della complessità. Per evitare il problema, si usa una sintassi alternativa del linguaggio che viene detta *grammatica a clusole definite* (DCG).

1. Introduzione dei nomi comuni plurali.
2. Aggiunta della persona (prima, seconda, terza) e il numero (singolare o plurale).
3. Garanzia di concordanza tra persona e numero di soggetto e verbo.

PROLOG consente di utilizzare un artificio sintattico: siccome tutti i predicati hanno sempre le 2 liste in fondo, si può utilizzare una libreria che permette di non scrivere le parti, siccome sarà poi il programma ad aggiungerle.

Solitamente, la parte del lessico viene data in carico a un altro file, che verrà incluso nel file da eseguire; noi questo non lo facciamo.

I verbi sono gli ultimi in fondo: hanno una forma prima singolare e terza singolare come "*live*", "*lives*" o "*eat*", "*eats*"; 2 atomi diversi. La stessa cosa succede per i nomi comuni come "*book/books*" o "*garden/gardens*".

```
% ...
np(tps, [PN | R], R) :-
    pn(PN).
np(tps, [Det, CN | R1], R) :-
    det(Det),
    cn(CN, _),
    optpps(R1, R).
np(ntps, [Det, CN | R1], R) :-
    det(Det),
    cn(_, CN),
    optpps(R1, R).
% ...
```

Se abbiamo un nome proprio, sappiamo già che questo vorrà un verbo in terza persona singolare. Se il nome comune è il primo, allora questo è singolare; se il nome comune è il secondo, allora è plurale.

```
a(I, R) :-
    np(PN, I, R1),
    vp(PN, R1, R).
% ...
```

Estratta la persona e il suo tipo, che sia singolare o plurale, questa sarà la stessa da andare a usare nel verbo. La prima clausola del sintagma nominale ci dirà se stiamo parlando di `tps` o `ntps`, dove il 1° è quello dove la persona che viene presa è con la "s" finale; il 2° con la forma "infinitive".

Il goal

```
a([Alice, eat, the, cake, in, the, garden], []).
```

restituirà `false` siccome il programma sa che sulla sinistra deve esserci un verbo alla terza persona singolare.

Riusciamo anche a costruire qualcosa che permette non solo di accettare affermazioni, ma anche di accettare domande. Per farlo, sfruttiamo una caratteristica delle domande poste in inglese.

1. Alla domanda esempio "*who eats the cake*", la parola "*who*" viene usata al posto del soggetto che sta prima del resto della frase.
2. Domande a risposta "*yes*" o "*no*" hanno "*does*" nella frase.
3. Domande per il luogo "*where*" sono un set delle domande a risposta "sì" o "no".

```
s(I, R) :-  
    a(I, R).  
s(I, R) :-  
    q(I, R).  
% ...
```

La frase `s` (sentence) può essere un'affermazione `a` oppure una domanda `q`. L'affermazione `a` segue la solita grammatica mentre la domanda `q` può essere una domanda "*yes, no*", una domanda "*who*" o una domanda "*where*" a seconda di come comincia.

Le domande sono implementate dai predicati `aux` (auxiliary, *do, does*), `qwho`, `qyn` e `qwhere`.

```
% ...  
qyn(I, R) :-  
    aux(I, R).  
% ...
```

```
% ...  
aux([does | R1], R) :-  
    np(tps, R1, R2),  
    vp(ntps, R2, R).  
aux([do | R1], R) :-  
    np(ntps, R1, R2),  
    vp(ntps, R2, R).  
% ...
```

La domanda `qyn` comincia con un ausiliare "*do/does*":

- se "*does*" è presente, dopo ci vuole una noun phrase soggetto `np` in terza persona singolare (come "*Alice eat*"), e un qualcosa che comunque non è in terza persona singolare (come "*the cake*");
- se "*do*" è presente, tutto non è in terza persona singolare.

Siccome `np` e `vp` li avevamo già, ci basta semplicemente chiederlo se c'è un ausiliare o l'altro, permettendoci di fare il parsing di queste domande.

```
% ...  
qwhere([where | R1], R) :-  
    aux(R1, R).  
% ...
```

"*where*" è seguito da un'ausiliare, la cui struttura già abbiamo.

```
% ...
qwho([who | R1], R) :-
    vp(_, R1, R).
% ...
```

La domanda "*who*" è una seguita da "*do/does*", a seconda di chi sta facendo la domanda, aspettandosi una risposta al singolare o plurale. Siccome in concordanza entreremo a prescindere, ci basta dire che un `vp` qualsiasi può bastare.

Il goal

```
s([who, eat, the, cake, in, the, garden], []).
```

chiede se la frase ha senso: ci viene restituito 2 volte `true`. Anche il goal che va a sostituire alla terza persona singolare "*eat*" con "*eats*", avrà senso e restituirà allo stesso modo 2 volte `true`.

Il goal

```
s([where, does, Alice, eat, the, cake], []).
```

avrà un senso e una sola interpretazione, `true`.

	
Risposte al 1° goal	Risposta al 2° goal

## nlp4.pl

 [Vedere il programma](#) nlp4.pl

Possiamo anche richiedere la stampa della sintassi.

Siccome l'albero sintattico può essere esplorato, chiedere all'esecutore di stampare il soggetto di una frase o il luogo in cui una certa azione avviene, può essere fatto.

```
s(SS, I, R) :-
    a(SS, I, R).
s(SS, I, R) :-
    q(SS, I, R).
% ...
```

Una frase `s` è tale se riusciamo a ricondurla ad affermazione `a` o domanda `q`. `I` è la sequenza di token da elaborare, `R` il resto rimasto, `SS` la semantica o significato

```
% ...
a(action(VPN, [subject(SNP) | SPPs]), I, R) :-
    SVP = action(VPN, SPPs),
    np(SNP, PN, I, R1),
    vp(SVP, PN, R1, R).
% ...
```



Un'affermazione `a` è sempre formata da:

- una noun phrase `np` con i soliti 3 argomenti, con produzione aggiuntiva di semantica (se stiamo parlando di "*Alice*", allora `SNP` vogliamo sia quella);
- un verbo `vp` con i soliti 3 argomenti, con semantica aggiunta, l'azione `SVP` (che cosa fa il soggetto? "*eats*", "*writes*", "*read*").

Il risultato che viene costruito è un termine azione `action`, che ha come 1° argomento il tipo di azione compiuta e come 2° argomento una lista che inizia sempre con un termine soggetto dell'azione `subject(SNP)`, seguito dai modificatori `SPPs` che in questo caso stiamo attribuendo all'azione.

```
% ...
np(id(PN), tps, [PN | R], R) :-
    pn(PN).
np(the(CN), tps, [Det, CN | R], R) :-
    det(Det),
    cn(CN, _).
np(the(CN, SPPs), tps, [Det, CN | R1], R) :-
    det(Det),
    cn(CN, _),
    SPPs = [_ | _],
    optpps(SPPs, R1, R).
np(all(N), ntps, [Det, CN | R], R) :-
    det(Det),
    cn(N, CN).
np(some(N, SPPs), ntps, [Det, CN | R1], R) :-
    det(Det),
    cn(N, CN),
    optpps(SPPs, R1, R).
% ...
```

Supponiamo di avere un nome proprio `PN`: se così è, allora prima abbiamo un `tps` (verbo terza persona singolare) che comincia con un nome. Il 1° argomento `id(PN)` diventa il risultato.

Se dico "*the chairs*", mi riferisco a "tutte le sedie" `the(CN)` `the(CN, SPPs)`; se dico "*some chairs*" mi riferisco ad alcune sedie `some(N, SPPs)`.

```
% ...
vp(action(N, SPPs), tps, [IV | R1], R) :-
    iv(N, IV),
    optpps(SPPs, R1, R).
vp(action(IV, SPPs), ntps, [IV | R1], R) :-
    iv(IV, _),
    optpps(SPPs, R1, R).
vp(action(N, [object(SNP) | SPPs]), tps, [TV | R1], R) :-
    tv(N, TV),
    np(SNP, _, R1, R2),
    optpps(SPPs, R2, R).
vp(action(TV, [object(SNP) | SPPs]), ntps, [TV | R1], R) :-
    tv(TV, _),
    np(SNP, _, R1, R2),
    optpps(SPPs, R2, R).
% ...
```

I verbi transitivi hanno sempre un oggetto.

Prendendo il verbo "*eats*", l'oggetto "*the cake*", costruendo l'azione che ha come verbo il primo e come

oggetto il secondo, aggiungendo le altre feature `SPPs`, otteniamo la semantica.

Dal goal

```
s(SS, [Alice, eats, the, cake], []).
```

ci aspettiamo un'azione di tipo *"eat"*, un soggetto univocamente identificato dall'identificatore *"Alice"*, un oggetto di categoria *"cake"*.

Dal goal

```
s(SS, [Alice, eats, the, cake, in, the, garden], []).
```

ci aspettiamo 2 cose possibili: o un'azione *"eat"* che si svolge in un luogo con modificatore *"in the garden"*, con *"Alice"* soggetto identificativo e *"cake"* l'oggetto; oppure un'azione *"eats"* che ha come soggetto l'identificativo *"Alice"* e come oggetto non una *"cake"* qualsiasi, ma quella che ha come modificatore *"in the garden"*.

	
Semantica del primo goal	Casi del secondo goal

## chatbot.pl

 [Vedere il programma](#) chatbot.pl

Il programma che stiamo per vedere, funziona come un bot di messaggistica: sulla base di una conoscenza, questo bot risponde a delle domande poste in linguaggio naturale. Con `swipl` installato, possiamo vedere il suo comportamento nel terminale.

Navighiamo alla cartella con dentro i file `.pl`, ed eseguiamo il comando `swipl chatbot.pl`.

```
maruko@markarch ~/Documents/uni/3_anno/ai/Prolog swipl chatbot.pl
A very simple chatbot

> 
```

Il messaggio di benvenuto del bot

Se la base di conoscenza è vuota, il bot restituirà una frase che lo accenna: `I do not know what to say`.

```
maruko@markarch ~/Documents/uni/3_anno/ai/Prolog swipl chatbot.pl
A very simple chatbot

> does alice eat the cake?
I do not know what to say

> 
```

does alice eat the cake?

Se informiamo il bot con la frase `alice eats the cake.`, questo a domande future fornirà risposta.

```
✖ maruko@markarch ~/Documents/uni/3_anno/ai/Prolog swipl chatbot.pl
A very simple chatbot

> alice eats the cake.
OK
> 
```

OK

```
✖ maruko@markarch ~/Documents/uni/3_anno/ai/Prolog swipl chatbot.pl
A very simple chatbot

> alice eats the cake.
OK
> does alice eat the cake?
alice eats the cake
> who eats the cake?
alice eats the cake
> 
```

does alice eat the cake?, who eats the cake?

Da notare che l'affermazione non viene immagazzinata da nessuna parte.

Le risposte alle domande, si basano sull'affermazione sì, ma sono ottenute dalla domanda, che simmetricamente produce un risultato.

---

16/05/2023