Sviluppo di un Parser in Prolog secondo la DCG per l'analisi di frasi in lingua italiana

Alessio Capriotti, Davide De Grazia, Enrico Piergallini

ABSTRACT

Il presente lavoro propone lo sviluppo di un parser in Prolog per effettuare il parsing di frasi in lingua italiana. Attraverso l'utilizzo della notazione delle regole grammaticali in Prolog (DCG), il software analizza e verifica la correttezza sintattica delle frasi fornite in input. In particolare, l'implementazione comprende la costruzione del parse tree, offrendo una rappresentazione strutturale delle frasi analizzate.

INTRODUZIONE

Il campo della *Natural Language Processing* (NLP) rappresenta ad oggi un ambito di ricerca sempre più cruciale, mirato a sviluppare soluzioni avanzate per comprendere e interagire con il linguaggio umano attraverso sistemi informatici. In questo contesto, il parsing svolge un ruolo fondamentale per il riconoscimento e la comprensione della struttura sintattica delle frasi. Questo lavoro propone un approccio al parsing delle frasi in lingua italiana mediante l'utilizzo del linguaggio di programmazione Prolog, noto per la sua natura dichiarativa e la flessibilità. Basandoci sulla notazione delle regole grammaticali in Prolog, anche conosciute come *Definite Clause Grammars* (DCGs), ci proponiamo di affrontare il problema del parsing in modo esaustivo e comprensibile, affrontando prima alcuni concetti teorici propedeutici.

IL PROBLEMA GENERALE

La *Natural Language Processing* è un ramo dell'Intelligenza Artificiale che si occupa di sviluppare modelli computazionali allo scopo di conferire ai computer la capacità di comprendere il testo e le parole pronunciate in modo simile a quanto possono fare gli esseri umani. L'NLP si occupa di sfide complesse, tra cui la comprensione del significato, la traduzione automatica, la *sentiment analysis* e molto altro. Per semplificare, si potrebbe pensare la questione come se l'obbiettivo della NLP fosse quello di creare un collegamento tra i linguaggi naturali, utilizzati dagli esseri umani, e i linguaggi di programmazione, compresi dai computer. Nel paragrafo successivo verrà analizzata la differenza tra questi due mondi e il motivo per cui questa differenza genera una grande complessità ai fini della NLP.

Confronto tra linguaggi naturali e linguaggi di programmazione

I linguaggi naturali, a differenza dei linguaggi di programmazione, non sono progettati per essere tradotti in un insieme finito di operazioni matematiche. Questi linguaggi, come l'italiano, l'inglese o il francese, rappresentano infatti ciò che gli esseri umani utilizzano per comunicare tra loro, spesso esprimendo concetti astratti, vaghi o sfumati. Al contrario, un programma scritto con un linguaggio di programmazione guida la macchina in modo preciso, indicandole esattamente cosa fare, con una sequenza finita di operazioni elementari. Oltre a ciò, un linguaggio naturale è costituito da un insieme di frasi ben formate che è, a priori, indeterminato. Non è possibile cioè caratterizzare tutte le frasi di una lingua in modo finito e deterministico, a causa di fattori come la varietà di vocaboli, l'evoluzione continua della lingua e fenomeni come la produttività (ovvero la creazione di nuove parole) e l'obsolescenza (la caduta in disuso di altre parole). Ma supponiamo, per assurdo, di poter conoscere l'intero vocabolario di una lingua, considerando tutte le parole, i termini e i dialetti possibili, e di monitorare la sua continua evoluzione: a questo punto rimarrebbe forse il problema più significativo, quello legato alle ambiguità.

Le ambiguità

Un'ambiguità in un linguaggio naturale si verifica quando una parola o una frase può essere interpretata in modi diversi a causa di mancanza di chiarezza semantica, strutturale o contestuale. In altre parole, non sempre c'è una corrispondenza univoca tra significato e significante, e ciò può rendere difficile determinare il senso esatto di un'espressione senza il contesto adeguato. Un'ambiguità può essere classificata principalmente come:

- **lessicale**, quando l'ambiguità riguarda il contenuto concettuale di una singola parola. Ad esempio, la parola "*riso*" può indicare sia il tipo di cereale che il participio passato del verbo *ridere* e così via.
- oppure **sintattica**, quando una frase può essere interpretata in modi diversi a seconda della struttura sintattica che gli attribuiamo. Ad esempio, la frase "*Maria non sta andando a Parigi in treno*" è ambigua, dato che può essere intesa in diversi modi tra i quali: "Maria sta andando a Parigi ma con un mezzo di trasporto diverso dal treno", oppure "Maria ha preso il treno ma non per andare a Parigi", oppure "non è Maria che sta andando".

Definire e gestire tutte queste ambiguità, che spesso per gli esseri umani possono sembrare di interpretazione ovvia, è invece una sfida molto critica nella progettazione di un sistema di NLP, poiché richiede la considerazione del contesto, delle regole grammaticali e delle conoscenze semantiche per ottenere un risultato accurato. Più avanti verrà ripreso il tema delle ambiguità per affrontare (anche se marginalmente) la questione.

SISTEMI DI REGOLE FORMALI

Le frasi in una lingua come l'italiano o l'inglese sono molto più di semplici sequenze arbitrarie di parole. Non possiamo concatenare qualsiasi insieme di parole e ottenere una frase ragionevole. Quantomeno, il risultato deve conformarsi a ciò che consideriamo grammaticale.

Una **grammatica** per una lingua è un insieme di regole che specifica quali sequenze di parole sono accettabili come frasi di quella lingua, ovvero stabilisce come le parole devono raggrupparsi e in quali ordine all'interno della frase. Per cui, data una grammatica per una lingua, possiamo esaminare qualsiasi sequenza di parole e verificare se soddisfa i criteri per essere una frase accettabile. Se la sequenza è effettivamente accettabile, il processo di verifica avrà stabilito quali sono i gruppi di parole e come sono assemblati nella frase. In altre parole, avrà stabilito la struttura della frase. Definire questa struttura non è sempre semplice, ci sono infatti diversi approcci per farlo che hanno richiesto diversi decenni di studi nell'ambito della linguistica. Per i nostri scopi ci limiteremo all'approccio della cosiddetta **grammatica formale**.

Grammatiche formali

La grammatica formale è un sistema di regole che delineano matematicamente (e perciò in modo preciso) la struttura di un linguaggio naturale. Ci sono quattro tipi di grammatiche formali, secondo la gerarchia teorizzata da Chomsky (1956):

- **Grammatiche regolari** (**Regular Grammars** o **Type 3**): questo è il livello meno potente della gerarchia. Le grammatiche regolari sono utilizzate per descrivere linguaggi regolari, come quelli riconosciuti dalle espressioni regolari. Sono adatte per modellare la sintassi di molti linguaggi di programmazione per le operazioni di ricerca e sostituzione.
- **Grammatiche libere dal contesto** (**Context-Free Grammars** o **Type 2**): questo tipo di grammatiche sono "libere dal contesto" nel senso che la produzione di una regola di grammatica può essere applicata indipendentemente dal contesto in cui appare. In altre parole, la struttura di una frase o di un'espressione può essere definita senza dover considerare il contesto circostante, e quindi l'aspetto semantico non viene preso in considerazione.
- **Grammatiche sensibili al contesto** (**Context-sensitive grammar** o **Tipo 1**): queste grammatiche sono chiamate "sensibili al contesto" perché le regole di produzione possono considerare il contesto circostante. Questa caratteristica conferisce loro una maggiore flessibilità, consentendo di descrivere una gamma più ampia di linguaggi rispetto alle grammatiche libere dal contesto. Di conseguenza, le grammatiche sensibili al contesto sono considerate più potenti in quanto possono rappresentare strutture linguistiche più complesse.

- **Grammatiche non limitate** (**Recursively enumerable** o **Tipo 0**): queste grammatiche rappresentano il livello più elevato di potenza espressiva tra tutti i tipi di grammatiche, non hanno restrizioni specifiche sulla forma delle regole di produzione e infatti generano esattamente tutti i linguaggi che possono essere accettati da una Macchina di Turing.

D'ora in avanti, concentreremo la nostra attenzione sulle grammatiche di Tipo 2 per ragioni di semplicità e di espressività. Le grammatiche sensibili al contesto sono infatti potenzialmente complesse da implementare a causa della loro complessità non lineare, e perciò vengono principalmente esaminate in ambito teorico. Allo stesso modo, escluderemo anche le grammatiche non limitate che a maggior ragione risultano impraticabili. D'altra parte, escluderemo le grammatiche regolari poiché il loro utilizzo è circoscritto a specifici contesti e non forniscono un'espressività sufficiente per i nostri scopi.

Grammatica libera dal contesto: definizione formale

Una grammatica libera dal contesto G può essere definita come una quadrupla:

$$G = \langle C, T, R, \Sigma \rangle$$

dove:

- **C** è un insieme finito di simboli **non terminali**, usati per rappresentare sintagmi o costituenti sintattici, ovvero gruppi di parole che si combinano in una struttura grammaticale all'interno di una frase.
- **T** è un insieme finito di simboli **terminali**, ovvero l'insieme delle parole effettive del linguaggio (nel nostro caso) .
- **R** è un insieme finito di **regole di produzione** (o **derivazione**). Ogni regola è nella forma $\alpha \to \beta$, dove α è un simbolo non terminale appartenente all'insieme C e β è una lista ordinata di elementi appartenenti a C e/o T (ovvero $\alpha \in C$ e $\beta \in C \cup T$). Le regole consentono di definire strutture in base alle sottostrutture di cui sono composte.
- **Σ** è un particolare elemento chiamato **simbolo di partenza**, è un elemento appartenente a C ed è il simbolo corrispondente al sintagma di livello più alto.

Una grammatica può essere utilizzata per generare frasi considerando ciascuna delle sue regole come una regola di "riscrittura", ovvero α viene riscritto come sequenza dei simboli β. Ciascun simbolo non terminale in questa sequenza β viene riscritto secondo un'ulteriore regola appropriata e così via fino a quando tutti i simboli nella sequenza diventano simboli terminali, cioè parole o caratteri del linguaggio.

Questo approccio viene utilizzato per la generazioni di frasi. Inoltre, una grammatica può essere utilizzata per lo scopo opposto, ossia per riconoscere se una sequenza di parole rappresenta una frase sintatticamente "ben formata". Questo punto di vista è utilizzato invece nell'analisi di una frase, in modo da fornire una corrispondente analisi strutturale.

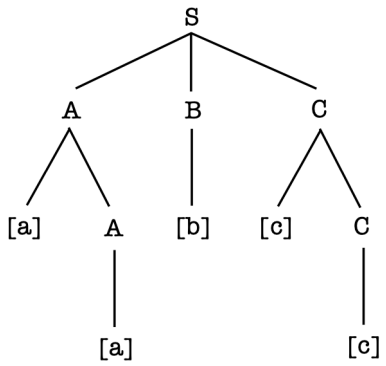
Qui di seguito viene proposto un esempio di grammatica *context-free*, dove `S` è il simbolo di partenza, `A`, `B` e `C` sono simboli non terminali e `[a]`, `[b]` e `[c]` sono simboli terminali:

```
S → A, B, C.
  A → [a].
A → [a], A.
  B → [b].
  C → [c].
C → [c], C.
```

Inoltre, come si può notare, alcune delle regole contengono richiami a se stesse, nello specifico:

```
A → [a], A.
C → [c], C.
```

Queste sono chiamate *regole ricorsive* e consentono la generazione di stringhe di qualsiasi lunghezza fino all'infinito: perciò possono descrivere stringhe come ad esempio "*abccc*" e "*aaaaaabcc*", ma non "*abbbcc*". Tali regole possono essere per di più rappresentate sotto forma di una struttura ad albero (detto **albero di derivazione**) nel seguente modo.



In questo albero, `S` è la *radice*, `A`, `B` e `C` sono i *nodi intermedi*, mentre i due `[a]`, `[b]` e i due `[c]` sono le *foglie* dell'albero.

CATEGORIE SINTATTICHE

In questa sezione verrà esaminato un modello che utilizza le regole grammaticali per descrivere la struttura delle frasi.

Categorie grammaticali e strutture costituenti

Le *strutture costituenti*, o più semplicemente *costituenti*, sono formate a partire da categorie grammaticali di base. Di seguito sono elencate alcune delle categorie più comuni assieme a qualche esempio:

- sostantivi (N): *tavolo, computer, penna.*
- determinanti (DET): *il, la, alcuni.*
- verbi (V): *mangiare, mangiando, dorme, avere.*
- aggettivi (ADJ): *grande, veloce, suo.*
- ausiliari (AUX): *ha, è, deve, potrebbe.*
- avverbi (ADV): *ieri, raramente, piuttosto.*
- congiunzioni (CONJ): *e, o.*
- preposizioni (PREP): *di, con, su, per.*
- pronomi (PRON): *lui, noi, che.*

Questi categorie possono quindi essere combinate in diversi modi per formare le sottostrutture della frase, ovvero i costituenti. Alcuni di questi vengono proposti qui di seguito:

- frase o *sentence* (S): ovvero l'intera espressione, ad es.: *quella ragazza con il vestito rosso è una cantante*, etc.
- sintagma nominale o *noun phrase* (NP): un nome o un pronome insieme agli elementi che lo modificano o lo specificano, ad es.: *il computer*, *l'uomo con gli occhiali*, *alcuni degli uomini più ricchi*, etc.
- sintagma verbale o *verb phrase* (VP): un verbo o un gruppo verbale e i suoi complementi immediati (diretti, indiretti e preposizionali), ad es.: *è addormentato*, *ha dato a Maria un libro*, *deve essere alla ricerca di qualcosa*, etc.
- sintagma preposizionale o *prepositional phrase* (PP): una preposizione seguita da un sintagma nominale, ad es.: *con un telescopio*, *in fondo al giardino*, etc.

- sintagma avverbiale o *adverbial phrase* (ADVP): un avverbio insieme ai suoi modificatori, ad es.: *ieri sera*, *meno spesso*, *il più velocemente possibile*, etc.

(L'elenco descritto è puramente indicativo, per diverse applicazioni potrebbe infatti essere più vantaggioso riconoscere diversi costituenti intermedi tra la frase e la parola, a seconda dei casi.)

Descrizione delle strutture costituenti

Una struttura costituente è delineata attraverso regole di produzione, come abbiamo visto in precedenza. Per esempio, si potrebbe affermare che una frase `S` è composta da un sintagma nominale `NP` e un sintagma verbale `VP`. Questo ci porta alla seguente regola:

$$\texttt{S} \rightarrow \texttt{NP, VP}$$

Allo stesso modo, un sintagma verbale `VP` potrebbe essere composto da un verbo `V` seguito da un sintagma nominale `NP` e da un sintagma avverbiale `ADVP`:

$$\texttt{VP} \rightarrow \texttt{V, NP, ADVP}$$

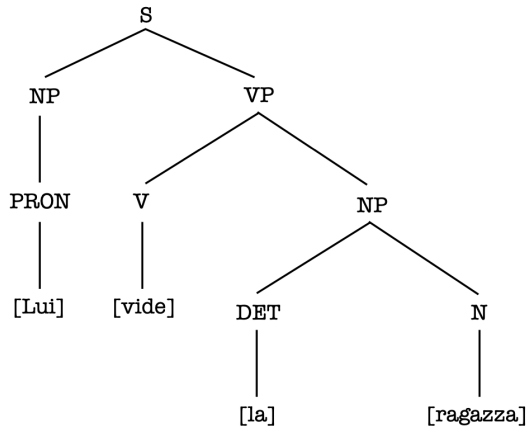
e così via.

Naturalmente, le possibili combinazioni sono numerose e dipendono principalmente dalla lingua che si decide di analizzare. Ad esempio, l'italiano, l'inglese, lo spagnolo e tutte le altre lingue latine seguono uno schema di costruzione della frase detto SVO (Soggetto-Verbo-Oggetto). Il giapponese o il turco sono lingue SOV (Soggetto-Oggetto-Verbo), mentre l'arabo è una lingua VSO (Verbo-Soggetto-Oggetto). Tuttavia, anche una volta scelta la lingua, la complessità persiste a causa delle numerose eccezioni che devono essere considerate. Pertanto, il processo di descrizione completa dei costituenti è tutt'altro che semplice.

Frasi ben formate e alberi di parsing

Le frasi nel linguaggio naturale possono essere analizzate mediante un determinato insieme di regole, come mostrato sopra. Una frase è considerata **ben formata** se esiste almeno un insieme o sequenza di regole che consentono di definire una descrizione strutturale completa della frase, in modo da determinare la sua correttezza secondo la grammatica definita. Tale descrizione può essere rappresentata sotto forma di una struttura ad albero, detto **albero di parsing** (o **parse tree**), riflettendo esattamente quali regole sono state utilizzate per costruirla e in quale ordine. A titolo illustrativo,

ecco l'albero di parsing per la frase "*Lui vide la ragazza*", insieme alle regole grammaticali impiegate nella sua costruzione.




```
S → NP, VP.
NP → PRON.
NP → DET, N.
VP → V, NP.
PRON → [Lui].
V → [vide].
DET → [la].
N → [ragazza].
```

IL PROBLEMA DI PARSING IN PROLOG

Data una grammatica formale, il problema di definire se una frase è ben formata è comunemente noto come **problema di parsing**. Un programma che si occupa di fare il parsing di un testo dato in input è chiamato **parser**.

Il linguaggio di programmazione Prolog si rivela particolarmente adatto a questo scopo. La sua natura dichiarativa lo rende infatti estremamente adatto per la descrizione di regole grammaticali di un linguaggio. Inoltre, l'introduzione del formalismo delle regole grammaticali in Prolog, noto anche come *grammatiche a clausole definite* (DCG), semplifica la scrittura di parser in Prolog. Benché il formalismo in DCG non sia parte integrante del Prolog standard, molte implementazioni di Prolog lo forniscono automaticamente come supporto built-in.

Premesse di base

La struttura principale che stiamo esaminando nel contesto del problema di parsing è una frase, intesa come una sequenza di parole di cui bisogna determinarne la struttura. Dato che il modo standard per rappresentare una sequenza è per mezzo di una lista, l'input del parser verrà quindi rappresentato da una lista di atomi in Prolog. Tuttavia, un primo problema che potrebbe sorgere è quello della normalizzazione del testo, ovvero come renderlo uniforme e coerente attraverso un processo di standardizzazione. Nello specifico, data una frase, come si può suddividere il testo in una sequenza di parole o di caratteri. Questo viene definito come **problema di tokenizzazione** (o **segmentazione**), dove un token può essere una parola, una frase, un simbolo o una

qualsiasi altra unità significativa del testo. Si pensi ad esempio ad una frase del tipo "*L'aereo FR-756, che ha preso Mario, arriverà a New York per le 13:40*": in questo caso la suddivisione in token è una questione tutt'altro che banale, dato che bisogna gestire parole apostrofate, segni di punteggiatura e caratteri speciali. Per i nostri scopi, ci porremo nel caso più semplice possibile, in cui non ci sono le tre eccezioni viste nell'esempio appena descritto, e quindi i token corrisponderanno esattamente alle parole della frase separate da spazi. Inoltre, non verrà analizzata la procedura per convertire una frase "grezza" in una lista Prolog, per cui l'input verrà fornito al parser direttamente in forma di lista.

Definizione delle regole

Fatte le dovute premesse, implementeremo ora un programma per verificare se una data sequenza di parole costituisce una frase ben formata. In seguito, considereremo anche come implementare un programma che ricavi la struttura della frase e la visualizzi a schermo, ma per ora sarà più semplice ignorare questa complessità aggiuntiva.

Sviluppiamo la nostra analisi prendendo ad esempio la seguente frase rappresentata in Prolog come una lista di atomi nel seguente modo:

```
[il, cane, mangia, un, biscotto]
```

Per prima cosa dobbiamo definire la grammatica, e la più semplice può essere descritta dalla seguente affermazione: una frase `sentence` è formata da un sintagma nominale `NP` e da un sintagma verbale `VP`; un sintagma nominale `NP` è formato da un articolo `DET` e da un sostantivo `N`; un sintagma verbale `VP` può invece essere formato da un verbo `V` o da un verbo `V` e un sintagma verbale `VP`. Si potrebbe facilmente esprimere tale affermazione sotto forma di enunciati logici, oppure utilizzando la notazione formale della grammatica libera dal contesto:

$$\begin{aligned} \texttt{S} &= \texttt{NP} \wedge \texttt{VP} \\ \texttt{NP} &= \texttt{DET} \wedge \texttt{N} \\ \texttt{VP} &= \texttt{V} \vee (\texttt{V} \wedge \texttt{NP}) \end{aligned}$$

$$\begin{aligned} \texttt{S} &\rightarrow \texttt{NP, VP.} \\ \texttt{NP} &\rightarrow \texttt{DET, N.} \\ \texttt{VP} &\rightarrow \texttt{V.} \\ \texttt{NP} &\rightarrow \texttt{V, NP.} \end{aligned}$$

Approccio con uso dell'`append`

Secondo la prima regola della grammatica, il compito si scompone nel trovare nella frase `sentence` un sintagma nominale `noun_phrase` all'inizio della sequenza e poi un sintagma verbale `verb_phrase` in ciò che rimane. Alla fine dovremmo aver esaurito esattamente le parole della sequenza, senza che nessuna avanzi. Dato che rappresentiamo le sequenze come liste, abbiamo già a disposizione il predicato `append` per scomporre una lista in altre due. Perciò possiamo scrivere:

```
sentence(X) :- append(Y, Z, X), noun_phrase(Y), verb_phrase(Z).
```

Allo stesso modo, possiamo definire `noun_phrase(X)` e `verb_phrase(X)`:

```
noun_phrase(X) :- append(Y, Z, X), determiner(Y), noun(Z).
verb_phrase(X) :- verb(X).
verb_phrase(X) :- append(Y, Z, X), verb(Y), noun_phrase(Z).
```

A questo punto definiamo le parole, specificando la categoria sintattica alla quale appartengono, nel seguente modo:

```
determiner([il]).
determiner([un]).
noun([cane]).
noun([biscotto]).
verb([mangia]).
```

Il programma è ora completo: ci dirà infatti con successo quali sequenze di parole sono frasi ben formate secondo la grammatica. Tuttavia, è opportuno fare un'osservazione importante, considerando ad esempio la prima clausola e lanciando il seguente goal:

```
?- sentence([il, cane, mangia, un, biscotto]).
```

La variabile `X` verrà istanziata a `[il, cane, mangia, un, biscotto]`, ma inizialmente `Y` e `Z` non saranno istanziate. Infatti il goal, con la tecnica del backtracking, genererà una alla volta tutte le possibili combinazioni di `Y` e `Z` tali che, quando `Z` viene aggiunto a `Y`, il risultato è `X`. Tutte queste possibili combinazioni sono elencate qui di seguito:

```
Y = [], Z = [il, cane, mangia, un, biscotto]
Y = [il], Z = [cane, mangia, un, biscotto]
Y = [il, cane], Z = [mangia, un, biscotto]
Y = [il, cane, mangia], Z = [un, biscotto]
Y = [il, cane, mangia, un], Z = [biscotto]
Y = [il, cane, mangia, un, biscotto], Z = []
```

A questo punto il programma prenderà dall'elenco una combinazione alla volta per verificare se `Y` è una `noun_phrase` con un procedimento analogo (cioè, provando a soddisfare `noun_phrase(Y)`). Se viene verificato ciò, si dovrà appurare che `Z` sia una `verb_phrase` allo stesso modo. E' perciò evidente che questo approccio non è molto efficiente, visto che il predicato `append(Y,Z,X)` genera un gran numero di soluzioni, la maggior parte delle quali sono però inutili. Per cui conviene trovare un modo più diretto e meno dispendioso per arrivare alla soluzione.

Approccio con liste di differenza

Per risolvere il problema descritto sopra si potrebbe utilizzare il seguente formalismo, evitando così l'uso dell'`append`: ogni simbolo non terminale verrà rappresentato come un predicato con due argomenti dove il primo rappresenta l'input iniziale, ovvero l'intera sequenza di parole da analizzare, e il secondo rappresenta la sequenza rimanente dopo l'applicazione del predicato. Questo tipo di rappresentazione prende il nome di **liste di differenza** (**difference lists**).

Utilizzando l'approccio con liste di differenza, possiamo quindi ridefinire il predicato `sentence` come segue:

```
sentence(Input, Rest) :- noun_phrase(Input, Part), verb_phrase(Part, Rest).
```

Da notare che la sequenza di parole rimanenti dopo l'applicazione di `noun_phrase` coincide con l'input che viene passato al predicato `verb_phrase`, al contrario `Rest`, che è la sequenza di parole rimanenti dopo l'applicazione di `verb_phrase`, non fa da input a nessun altro predicato. Da questo osservazione, si può constatare che, se `Rest` è una lista vuota (`[ ]`), si ha che `sentence` può essere decomposta per trovare una `noun_phrase` all'inizio della sequenza e poi per trovare una `verb_phrase` in ciò che resta. Alla fine di questo processo, dovremmo aver utilizzato esattamente le parole della sequenza. Allo stesso modo, definiamo i predicati `noun_phrase` e `verb_phrase`:

```
noun_phrase(Input, Rest) :- determiner(Input, Part), noun(Part, Rest).
verb_phrase(Input, Rest) :- verb(Input, Rest).
verb_phrase(Input, Rest) :- verb(Input, Part), noun_phrase(Part, Rest).
```

Infine, bisogna adottare una simile convenzione anche per specificare le singole parole della frase in modo da indicare cosa rappresentano:

```
determiner([il|Rest], Rest).
determiner([un|Rest], Rest).
noun([cane|Rest], Rest).
noun([biscotto|Rest], Rest).
verb([mangia|Rest], Rest).
```

A questo punto il programma è completo. Per cui si potrebbero lanciare i seguenti goal:

```
?- sentence([il, cane, mangia, un, biscotto], []).
   true.
?- sentence([il, cane, mangia], []).
   true.
?- sentence([il, cane, mangia, un], []).
```

```
   false.
?- noun_phrase([il, cane], []).
   true.
?- verb_phrase([mangia, il, biscotto], []).
   true.
```

In modo opposto si potrebbe pensare di lanciare gli stessi goal, passando tuttavia una variabile `X` come primo argomento del predicato. In tal modo si possono ottenere tutte le combinazioni possibili che soddisfano il predicato in questione.

```
?- sentence(X, []).
   X = [il, cane, mangia] ;
   X = [il, cane, mangia, il, cane] ;
   X = [il, cane, mangia, il, biscotto] ;
   X = [il, cane, mangia, un, cane] ;
   X = [il, cane, mangia, un, biscotto] ;
   X = [il, biscotto, mangia] ;
   X = [il, biscotto, mangia, il, cane] ;
   X = [il, biscotto, mangia, il, biscotto] ;
   etc ...
```

A questo punto, è importante sottolineare un altro problema che emerge: nonostante la semplicità del programma, il codice appare già abbastanza disordinato. La situazione diventa notevolmente più complessa quando vengono aggiunte nuove clausole e ulteriori argomenti ai predicati. Per superare questo problema, si potrebbe considerare l'idea di abbandonare la sintassi standard del Prolog e adottare una nuova notazione.

Approccio con DCG

Le "Grammatiche a Clausole Definite", o "Definite Clause Grammar" (DCG), si riferiscono a una notazione utilizzata in Prolog per esprimere regole grammaticali in modo più conciso, migliorando la leggibilità del codice rispetto al Prolog standard attraverso l'eliminazione di dettagli non cruciali. Questa notazione riduce anche il rischio di errori di battitura da parte del programmatore, semplificando notevolmente la scrittura di parser e analizzatori sintattici. Le regole grammaticali definite in DCG vengono poi trasformate in clausole Prolog ordinarie direttamente dal sistema. Per illustrare ciò, possiamo semplificare l'esempio precedentemente analizzato nel seguente modo, con un risultato totalmente equivalente:

```
sentence --> noun_phrase, verb_phrase.
noun_phrase --> determiner, noun.
verb_phrase --> verb.
verb_phrase --> verb, noun_phrase.
determiner --> [il].
```

```
determiner --> [un].
noun --> [cane].
noun --> [biscotto].
verb --> [mangia].
```

L'unica differenza, rispetto all'approccio con liste di differenza, è che in questo caso, per esprimere il goal, si dovrà sfruttare il predicato built-in `phrase(Grammar, InputList)` che restituisce il valore `true` se la frase `InputList` che si desidera analizzare rispetta la grammatica `Grammar`.

```
?- phrase(sentence, [il, cane, mangia, un, biscotto]).
   true.
?- phrase(sentence, [il, cane, mangia, un]).
   false.
?- phrase(sentence, X).
   X = [il, cane, mangia] ;
   X = [il, cane, mangia, il, cane] ;
   X = [il, cane, mangia, il, biscotto] ;
   X = [il, cane, mangia, un, cane] ;
   etc ...
```

Il problema della concordanza

In questa sezione, viene analizzata una soluzione per gestire la complessità della concordanza linguistica. Per concordanza si intendono le regole che gestiscono la coerenza tra le parti variabili del discorso quando sono collegate sintatticamente tra di loro. Ogni lingua ha regole specifiche per la concordanza; prendiamo ad esempio l'inglese, che affronta solo la concordanza numerica senza declinazioni per genere o persona nei verbi. In italiano, invece, la concordanza può coinvolgere il genere (maschile o femminile), il numero (singolare o plurale) e la persona (1a, 2a o 3a). Dopodichè ci possono altri tipi di concordanza come quella verbale (ad es. *consecutio temporum*).

Per chiarezza nell'esposizione, ci concentreremo specificamente sulla concordanza numerica, dando per scontata quella di genere e persona, oltre che verbale. Tuttavia, è possibile considerare anche quest'ultime introducendo ulteriori parametri in modo analogo. Per garantire ciò, sarà sufficiente aggiungere un argomento `Number` ai predicati, che specifica se il termine è al singolare o al plurale. Applicando questa modifica, si otterrà il seguente risultato:

```
sentence(Number) --> noun_phrase(Number), verb_phrase(Number).
noun_phrase(Number) --> determiner(Number), noun(Number).
verb_phrase(Number) --> verb(Number).
verb_phrase(Number) --> verb(Number), noun_phrase(_).
```

```
determiner(singular) --> [il].
determiner(plural) --> [i].
noun(singular) --> [cane].
noun(plural) --> [cani].
noun(singular) --> [biscotto].
noun(plural) --> [biscotti].
verb(singular) --> [mangia].
verb(plural) --> [mangiano].
```

Si noti che nella seconda regola per la `verb_phrase`, si sottolinea il concetto che la pluralità o singolarità di un sintagma verbale dipende esclusivamente da quella del verbo e non da quella dell'oggetto, nel caso in cui quest'ultimo sia presente. Di conseguenza, "*il cane mangia il biscotto*" oppure "*il cane mangia i biscotti*" sono entrambe costruzioni corrette.

Generazione dell'albero di parsing

Gli alberi di parsing risultano utili in quanto forniscono una rappresentazione strutturale di una frase rispetto a una grammatica specifica, come si è visto in precedenza. Pertanto, si potrebbe impiegare un argomento aggiuntivo per generare un albero di parsing come risultato dell'analisi.

```
sentence(Number,  sentence(NP, VP)) --> noun_phrase(Number, NP),
    verb_phrase(Number, VP).
noun_phrase(Number, noun_phrase(DET, N)) --> determiner(Number, DET),
    noun(Number, N).
verb_phrase(Number, verb_phrase(V)) --> verb(Number, V).
verb_phrase(Number, verb_phrase(V, NP)) --> verb(Number, V),
    noun_phrase(_, NP).

determiner(singular, determiner(il)) --> [il].
determiner(plural, determiner(il)) --> [i].
noun(singular, noun(cane)) --> [cane].
noun(plural, noun(cani)) --> [cani].
noun(singular, noun(biscotto)) --> [biscotto].
noun(plural, noun(biscotti)) --> [biscotti].
verb(singular, verb(mangia)) --> [mangia].
verb(plural, verb(mangiano)) --> [mangiano].
```

Il nuovo programma genererà un parse tree con il seguente goal:

```
?- sentence(_, Tree, [il, cane, mangia, i, biscotti], []).
   T = sentence(noun_phrase(determiner(il), noun(cane)),
   verb_phrase(verb(mangia), noun_phrase(determiner(il),
   noun(biscotti))))
```

```
?- sentence(_, Tree, [il, cane, mangia], []).
   T = sentence(noun_phrase(determiner(il), noun(cane)),
   verb_phrase(verb(mangia)))
```

Implementazione di un parser in un caso realistico

Fino ad ora sono stati affrontati esempi elementari che sono utili per comprendere i concetti fondamentali. Tuttavia, si potrebbe pensare di estendere i concetti visti per introdurre nuove tipologie di categorie, sintagmi e vincoli sulla concordanza, in modo da analizzare frasi più complesse e quindi più simili ad una situazione realistica. Inoltre, data la complessità delle frasi da analizzare, potrebbe essere a sua volta complessa la lettura del parse tree. Per tale motivo è stato pensato di implementare una soluzione che converta l'albero in un formato grafico, grazie all'uso di Graphviz, un software open source per disegnare grafi descritti nel linguaggio DOT.

Il codice completo, assieme ad alcuni test, viene riportato di seguito (tutto il codice che è stato riportato in questa relazione è disponibile anche su GitHub al seguente link: `https://github.com/davidedg11/prologparsing/`).

```
% Importazione di alcune librerie utili
:- use_module(library(apply)).
:- use_module(library(gv)).
:- use_module(library(yall)).
:- use_module(library(term_ext)).


sentence(Number, Gender, sentence(NP, VP)) -->
   noun_phrase(Number, Gender, NP),
   verb_phrase(Number, Gender, VP).
sentence(Number, Gender, sentence(NP, PP, VP)) -->
   noun_phrase(Number, Gender, NP),
   prepositional_phrase(_, _, PP),
   verb_phrase(Number, Gender, VP).
sentence(Number, Gender, sentence(NP, VP, PP)) -->
   noun_phrase(Number, Gender, NP),
   verb_phrase(Number, Gender, VP),
   prepositional_phrase(_, _, PP) .

proper_noun_phrase(Number, Gender, noun_phrase(PN)) -->
   proper_noun(Number, Gender, PN).
noun_phrase(Number, Gender, noun_phrase(DET, N)) -->
   determiner(Number, Gender, DET),
   noun(Number, Gender, N).
noun_phrase(Number, Gender, noun_phrase(DET, N, REL)) -->
   determiner(Number, Gender, DET),
   noun(Number, Gender, N), relative_clause(Number, _, REL).
```

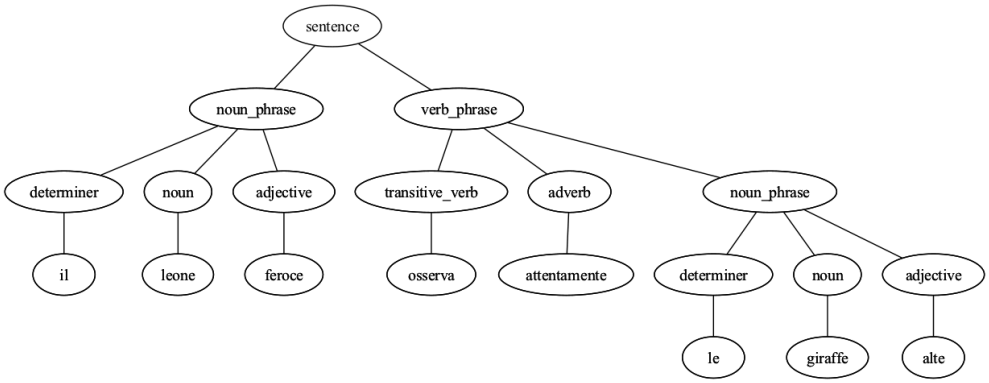
```
noun_phrase(Number, Gender, noun_phrase(DET, N, ADJ)) -->
   determiner(Number, Gender, DET),
   noun(Number, Gender, N),
   adjective(Number, Gender,ADJ).
noun_phrase(Number, Gender, noun_phrase(DET, N, ADJ, REL)) -->
   determiner(Number, Gender, DET),
   noun(Number, Gender, N),
   adjective(Number, Gender, ADJ),
   relative_clause(Number, _, REL).

relative_clause(Number, _, relative_clause(RP, VP)) -->
   relative_pronoun(_, _, RP),
   verb_phrase(Number, _, VP).
prepositional_phrase(Number, Gender, prepositional_phrase(PREP, NP)) -->
   preposition(_, _, PREP),
   noun_phrase(Number, Gender, NP).

verb_phrase(Number, _, verb_phrase(V)) -->
   transitive_verb(Number, V).
verb_phrase(Number, _, verb_phrase(V, NP)) -->
   transitive_verb(Number, V), noun_phrase(_, _, NP).
verb_phrase(Number, _, verb_phrase(V)) -->
   intransitive_verb(Number, V).

verb_phrase(Number, _, verb_phrase(V, ADV, NP)) -->
   transitive_verb(Number, V),
   adverb(_, _, ADV),
   noun_phrase(_, _, NP).
verb_phrase(Number, _, verb_phrase(V, ADV)) -->
   transitive_verb(Number, V),
   adverb(_, _, ADV).
verb_phrase(Number, _, verb_phrase(V, ADV)) -->
   intransitive_verb(Number, V),
   adverb(_, _, ADV).

determiner(singular, ma, determiner(il)) --> [il].
determiner(singular, fem, determiner(la)) --> [la].
determiner(plural, ma, determiner(i)) --> [i].
determiner(plural, fem, determiner(le)) --> [le].

noun(singular, ma, noun(ragazzo)) --> [ragazzo].
noun(singular, fem, noun(ragazza)) --> [ragazza].
noun(plural, ma, noun(ragazzi)) --> [ragazzi].
noun(plural, fem, noun(ragazze)) --> [ragazze].

noun(singular, fem, noun(giraffa)) --> [giraffa].
```

```
noun(plural, fem, noun(giraffe)) --> [giraffe].

noun(singular, ma, noun(leone)) --> [leone].
noun(plural, ma, noun(leoni)) --> [leoni].

noun(singular, ma, noun(binocolo)) --> [binocolo].

proper_noun(singular, _, proper_noun('Jenny')) --> ['Jenny'].

adjective(singular, ma, adjective(biondo)) --> [biondo].
adjective(singular, fem, adjective(bionda)) --> [bionda].
adjective(plural, ma, adjective(biondi)) --> [biondi].
adjective(plural, fem, adjective(bionde)) --> [bionde].

adjective(singular, ma, adjective(alto)) --> [alto].
adjective(singular, fem, adjective(alta)) --> [alta].
adjective(plural, ma, adjective(alti)) --> [alti].
adjective(plural, fem, adjective(alte)) --> [alte].

adjective(singular, _, adjective(feroce)) --> [feroce].
adjective(plural, _, adjective(feroci)) --> [feroci].

adverb(_, _, adverb(attentamente)) --> [attentamente].

relative_pronoun(_, _, relative_pronoun(che)) --> [che].

preposition(_, _, preposition(con)) --> [con].

transitive_verb(singular, transitive_verb(osserva)) --> [osserva].
transitive_verb(plural, transitive_verb(osservano)) --> [osservano].
intransitive_verb(singular, intransitive_verb(dorme)) --> [dorme].
intransitive_verb(plural, intransitive_verb(dormono)) --> [dormono].

test_sentence(X) :- phrase(sentence(_, _, _), X).

generate_parse_tree(Sentence, Tree) :-
    sentence(_, _, Tree, Sentence, []).

% Per esportare un parse tree in formato DOT,
% che può poi essere convertito in un file SVG
% utilizzando la libreria Graphviz.
export_tree_(Out, Tree, Id) :-
  Tree =.. [Op|Trees],
  ascii_id(Id),
  dot_node_id(Out, Id, options{label: Op}),
  maplist(export_tree_(Out), Trees, Ids),
  maplist(dot_edge_id(Out, Id), Ids).
```

```
% Viene visualizzato a schermo il parse tree
% view_parse_tree(Sentence, Tree) :-
  sentence(_, _, Tree, Sentence, []),
  gv_view({Tree}/[Out0]>>export_tree_(Out0, Tree, _)).

% Viene esportato parse_tree.svg
% export_parse_tree(Sentence, Tree) :-
  sentence(_, _, Tree, Sentence, []),
  gv_export('parse_tree.svg', {Tree}/[Out0]>>export_tree_(Out0, Tree, _)).
```

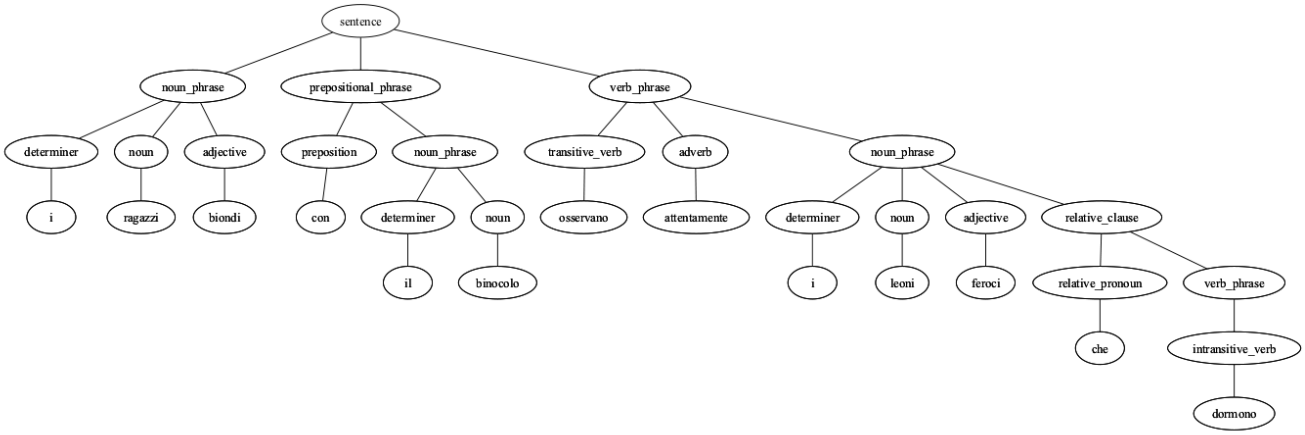
Vediamo ora alcuni goal che possono essere sfruttati per comprendere meglio il funzionamento del programma:

```
?- test_sentence([il, leone, feroce, osserva, attentamente, le, giraffe, alte]).
   true.

?- generate_parse_tree([il, leone, feroce, osserva, attentamente, le, giraffe,
   alte], Tree).
   Tree = sentence(noun_phrase(determiner(il), noun(leone), adjective(feroce)),
   verb_phrase(transitive_verb(osserva), adverb(attentamente),
   noun_phrase(determiner(le), noun(giraffe), adjective(alte)))).

?- view_parse_tree([il, leone, feroce, osserva, attentamente, le, giraffe,
   alte], Tree).
```



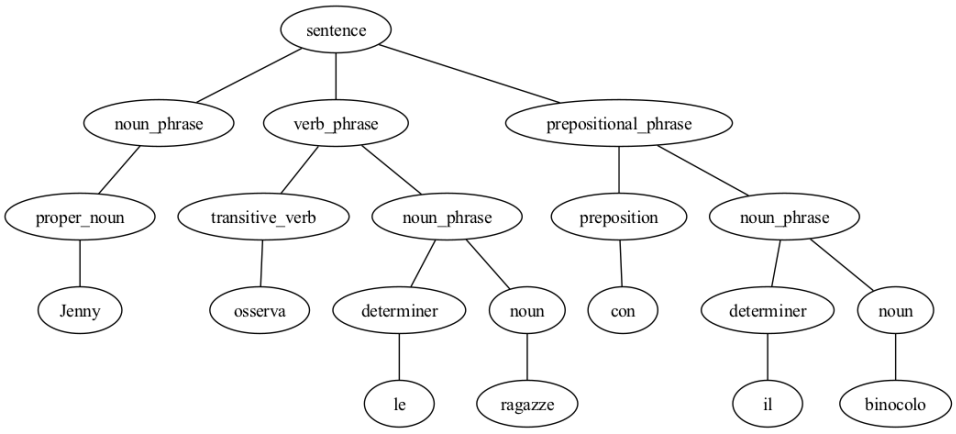

```
?- test_sentence([i, ragazzi, biondi, con, il, binocolo, osservano,
   attentamente, i, leoni, feroci, che,   dormono]).
   true.
```

```
?- view_parse_tree([i, ragazzi, biondi, con, il, binocolo, osservano, attentamente,
   i, leoni, feroci,  che, dormono], Tree).
```



A questo punto, è importante sottolineare un ultimo aspetto fondamentale che appare chiaramente nel prossimo esempio, facendo il parsing della frase "*Jenny osserva le ragazze con il binocolo*":

```
?- generate_parse_tree([Jenny, osserva, le, ragazze, con, il, binocolo], T).
   T = sentence(noun_phrase(proper_noun(Jenny)),
   verb_phrase(transitive_verb(osserva), noun_phrase(determiner(le),
   noun(ragazze))), prepositional_phrase(preposition(con),
   noun_phrase(determiner(il), noun(binocolo)))) .

?- view_parse_tree([Jenny, osserva, le, ragazze, con, il, binocolo], T).
```



Come si può notare, riprendendo un concetto espresso all'inizio della relazione, si verifica un'ambiguità sintattica a causa del sintagma preposizionale "*con il binocolo*",

il quale viene presunto come il mezzo attraverso il quale Jenny vede le ragazze. Questa presunzione è fatta da noi esseri umani in modo quasi automatico, tuttavia non c'è nulla che escluda la possibilità che siano le ragazze ad essere in possesso di un binocolo. Nella realizzazione del nostro programma, abbiamo scelto di adottare una soluzione conservativa, mantenendoci neutrali. In altre parole, il programma non si sbilancia nell'affermare se "*con il binocolo*" sia un complemento di mezzo o di specificazione. Pertanto constatiamo solo il fatto che si tratta di un sintagma preposizionale senza connetterlo ad altri sintagmi della frase.

Il programma sviluppato e descritto, resta in ogni caso una semplice applicazione, i margini di miglioramento sono ampi, per cui si invita il lettore a spingersi oltre per esplorare i confini del parsing e del mondo della NLP in generale.

BIBLIOGRAFIA

Patrick Saint-Dizier Harold Somers Annie Gal, Guy Lapalme. *Prolog for Natural Language Processing*. John Wiley Sons, 1989.

Ivan Bratko. *Prolog Programming for Artificial Intelligence*. John Wiley Sons, 2012.

James H. Martin Dan Jurafsky. *Speech and Language Processing*. Stanford University, 2023.

Hannes Max Hapke Hobson Lane, Cole Howard. *Natural Language Processing in Action*. Manning, 2019.

Christopher S. Mellish William F. Clocksin. *Programming in Prolog: Using the ISO Standard*. Springer, 2003.

Bratko (2012) Hobson Lane (2019) Annie Gal (1989) William F. Clocksin (2003) Dan Jurafsky (2023)