UNIVERSITÀ DEGLI STUDI DI TORINO
MASTER'S DEGREE IN COMPUTER SCIENCE

Esercizio 1: Traduttore transfer sintattico IT-EN

Diego Ercoli

January 26, 2021

Docente ..... Prof. Alessandro Mazzei
Corso ..... Tecnologie del linguaggio naturale

1 Obiettivi

Questa esercitazione mira ad implementare un traduttore automatico dall'italiano all'inglese basato su *Trasfer Sintattico*[1].

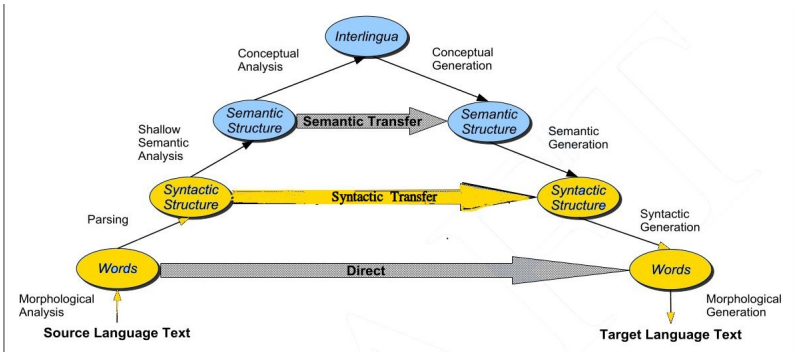


Figure 1: Il triangolo di Vauquois (1968)

Come evidenziato in Figura 1, il processo sotteso al suddetto paradigma consta nell'analizzare morfologicamente la frase in input, costruire il relativo albero di parsing, applicare ad esso opportune regole di trasformazione volte alla costruzione di una struttura sintattica per la lingua target, ponendo quindi le basi per la generazione della frase tradotta.
Ciascuna fase del processo verrà analizzata nel dettaglio nelle successive sezioni, prestando particolare attenzione alle strutture dati impiegate ed ai dettagli implementativi rilevanti. L'intera pipeline di elaborazione è stata implementata nel linguaggio Java.

[1] Approccio simbolico ben noto in letteratura definito da Vaquois nel suo triangolo[JM09]

1.1 Requisiti

La Traduzione automatica è un task intrinsecamente difficile, ancora oggi i moderni traduttori non sono in grado si svolgere pienamente quella *letteraria* [JM09].
L'esercitazione si pone l'obiettivo di costruire un applicativo in grado di elaborare un numero ben ristretto di frasi della lingua italiana, a tal proposito vengono riportate le specifiche dei vincoli che devono essere soddisfatti dall'input:

* Il periodo in ingresso deve essere mono-proposizionale[2]
* La frase deve essere affermativa
* Il verbo deve essere coniugato al modo Indicativo
* L'avverbio deve essere un modificatore del verbo
* La frase non deve contenere congiunzioni
* I **lemmi** dei termini della frase devono appartenere al seguente lessico:

```
1 {essere, la, spada-laser, di, tuo, padre, avere, fare, una, mossa, leale, il, ultimo, avanzo, del,
      vecchio, repubblica, spazzare, via}
```

Listing 1: Lessico accettato per l'Italiano

La violazione di uno o più dei suddetti vincoli comporterà il lancio di un'eccezione custom da parte dell'applicativo.
Il programma deve essere dunque in grado di elaborare le seguenti frasi:

(i) È la spada-laser di tuo padre

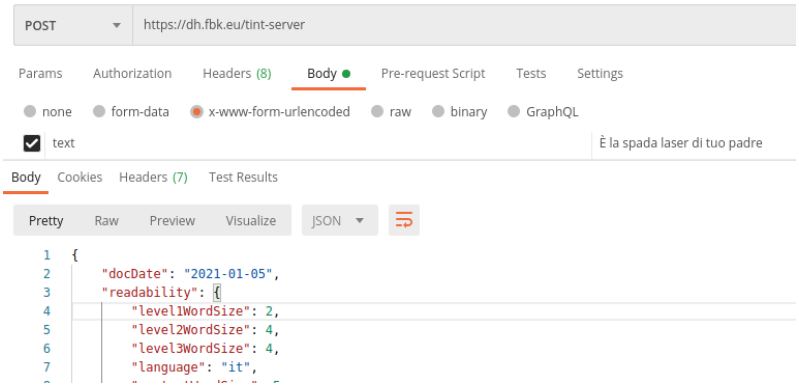
(ii) Ho fatto una mossa leale

(iii) Gli ultimi avanzi della vecchia Repubblica sono stati spazzati via

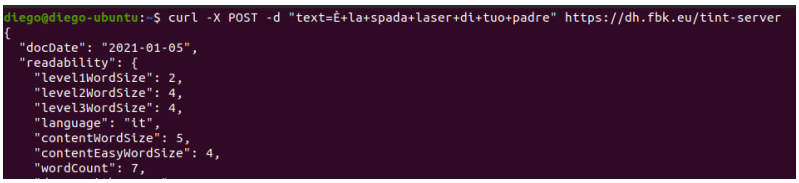
2 Analisi

La prima fase del processo di traduzione consta nell'eseguire l'analisi morfologica e sintattica della frase inserita dall'utente (Figura 1).
A tal fine, si è fatto uso delle API offerte dal progetto open-source Tint [PM18]. Si sono sperimentate due modalità di interazione:

1. Importazione libreria Java ed uso delle relative API[3]
2. Chiamata API REST al WebService di Tint, specificando nel body della richiesta HTTP la frase da processare. In Figura 2 viene mostrato come costruire la richiesta utilizzando l'applicativo Postman [4] ed il command line tool CURL[5].
   In questo modo si può dunque sviluppare applicativi di NLP senza essere legati all'utilizzo del linguaggio di programmazione Java, tuttavia i tempi di elaborazione possono essere influenzati dalla connessione di rete.



(a) Client Postman



(b) CURL

Figure 2: Chiamata REST al server di Tint e successiva HTTP Response contenente dati in formato JSON

Indipendentemente dal tipo di interazione scelta, in entrambi i casi Tint restituisce in formato JSON il risultato della sua pipeline di elaborazione che comprende: segmentazione del testo in tokens, analisi morfologica ed individuazione dei morfemi, POS tagging, lemmatizzazione, dependency parsing.
Analizzando il contenuto del JSON in questione, tra i vari dati è possibile notare la presenza di due JSONArray:

- array "*tokens*": è il risultato della segmentazione del testo in **N tokens**[6]. Ciascun elemento ha come attributi il lemma, il POS, l'insieme di morfermi, ... etc. Sono inoltre esplicitate in un sotto-array delle features linguistiche in relazione al valore del POS: genere e numero nel caso di un nome, tempo verbale nel caso di un verbo, ... etc.

- array "*basic-dependencies*": è il risultato dell'analisi sintattica, più nello specifico del parsing a dipendenze. L'array contiene le **N-1 dipendenze**[7], per ciascuna vengono specificati:

    * indice $i \in [1, N]$ che punta alla parola della frase che funge da **testa**

[2] Il periodo deve essere composto da una sola proposizione, non sono pertanto ammesse proposizioni subordinate
[3] Per maggiori dettagli si rimanda il lettore alla sezione *TINT Java API* della pagina: https://dh.fbk.eu/research/tint/download-and-usage/
[4] Applicativo client utilizzabile per testare API REST (https://www.postman.com/)
[5] Strumento con interfaccia a riga di comando utilizzabile per trasferire dati attraverso la rete
[6] Nel seguito con le espressioni "parole della frase" o "elementi lessicali della frase" faremo riferimento ai suddetti tokens
[7] Relazioni grammaticali binarie dirette tra gli elementi lessicali della frase.

* indice $j \in [1, N]$ ($j \neq i$) che punta alla parola che funge da **dipendente**
* etichetta della relazione

Nell'array è presente inoltre una relazione fittizia con etichetta "ROOT" che identifica la parola che costituisce la testa della frase (radice dell'albero).

Come verrà discusso nella Sezione 3, nella fase di *Transfer Sintattico* sarà necessario attraversare l'albero a dipendenze dalla radice verso le foglie, passando per i vari nodi intermedi. Dato un nodo, dobbiamo disporre di opportuni puntatori che ci consentano di ottenere la lista dei nodi figli in tempo $\mathcal{O}(1)$ e di poter effettuare quindi la visita in ampiezza in tempo $\mathcal{O}(N)$; cosa che risulterebbe impraticabile se utilizzassimo come struttura dati direttamente l'array *basic-dependencies* di Tint (per ogni nodo visitato, l'insieme dei suoi figli sarebbero prelevati ogni volta in tempo $\mathcal{O}(N)$, un'ipotetica visita dell'albero in ampiezza richiederebbe quindi un tempo $\mathcal{O}(N^2)$).
Di conseguenza, a partire dall'array di relazioni di dipendenze sopra citato, viene costruita una struttura dati ad Albero con un numero arbitrario di figli (Figura 3).

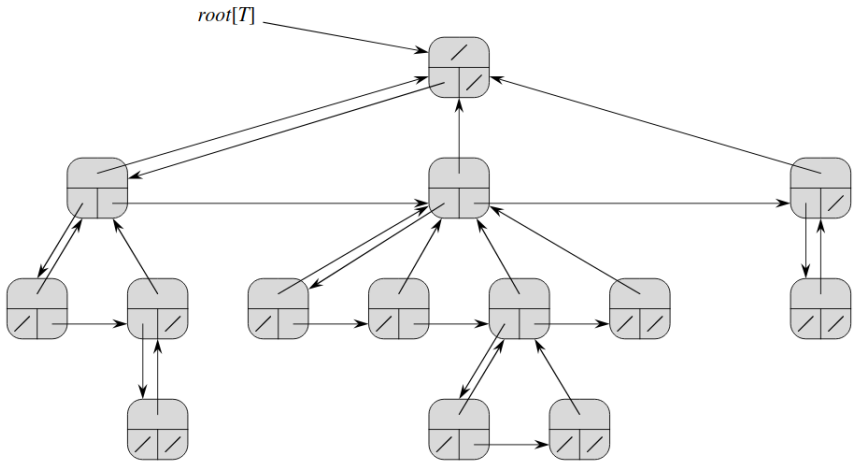


Figure 3: Rappresentazione intuitiva struttura dati ad albero.

```java
public class TreeNode<E> implements Iterable<E> {
    private List<TreeNode<E>> children;
    private TreeNode<E> father;
    private E el;
    //COSTRUTTORE e metodi getter e setter non mostrati per brevità.
    /**
     * Aggiunge un figlio (sotto-albero) all'albero corrente
     * @param child
     */
    public void addChild(TreeNode<E> child) {
        subTreeNodes.add(child);
        child.father = this;
    }
    /**
     * Restituisce true se l'albero sotteso al nodo corrente è una foglia
     * @return
     */
```

```
18      public boolean isLeaf() {
19          return children.isEmpty();
20      }
21
22     /**
23       * Restituisce i nodi figli che costituiscono sotto-alberi.
24       * @return the subTreeNodes
25       */
26      public List<TreeNode<E>> getSubTreeNodes() {
27          return subTreeNodes;
28      }
29      @Override
30      public Iterator<E> iterator() {
31          //Visita in ampiezza
32          return new WithSearchIterator<E>(this);
33      }
34
35  }
```

Listing 2: Implentazione in Java struttura dati albero con numero arbitrario di figli.

Nel Listing 3 vengono riportati i dettagli relativi al processo di parsing e di generazione della struttura dati ad albero. Ogni nodo dell'albero, corrispondente in maniera biunivoca ad una parola *i* della frase, viene dotato di un puntatore "aggiuntivo" ad un oggetto di tipo *MorphSyntaxInfo* che incapsula le informazioni morfologiche della parola in questione, provenienti dagli attributi dell'elemento *i* dell'array "tokens" di Tint precedentemente descritto. Per ogni relazione di dipendenza, il nodo corrispondente al dipendente viene aggiunto alla lista dei figli del nodo corrispondente alla testa della relazione.
Al termine del processo restituiamo l'albero appena creato, che incorpora tutte le informazioni morfo-sintattiche della frase che potranno essere acquisite e processate visitando l'albero a partire dalla radice (testa della frase).

```
1  public TreeNodeNode<MorphSyntaxInfo> buildDepParsingTreeNode(String sentence){
2          //Ottengo i dati da Tint in formato JSON
3          String json = getJSONViaHTTP(sentence);
4          //Deserializzo la stringa JSON in un oggetto JAVA
5          TintParserOutput obj = getParsedObject(json);
6          //Ottengo la lista di dipendenze di TINT
7          TintParserOutput.SentenceDependencyTint[] dependencies = obj.getSentences()[0].
               getBasicDependencies();
8          //Ottengo l'elenco di tokens di TINT (parole) che inglobano le informazioni
               morfologiche.
9          TintParserOutput.SentenceTokenTint[] morph_words = obj.getSentences()[0].getTokens();
10         //Dichiaro un array di nodi corrispondenti alle parole della frase
11         TreeNode<MorphSyntaxInfo>[] word_nodes = new TreeNode[morph_words.length];
12         for (int i=0;i<morph_words.length;i++) {
13             //Dichiaro un contenitore di informazioni morfologiche per il termine corrente
14             MorphSyntaxInfo info = null;
15             //Estraggo le informazioni morfologiche da morph_words[i] e le assegno ad info.
16             //...Nota: Codice omesso per brevità.
17             //Creo il nodo per la parola ed assegno il puntatore all'info morfologica
18             word_nodes[i] = new TreeNode<>(info);
19         }
20         int root=-1; //indice del nodo che costituirà la radice dell'albero
21         //Ciclo sulla lista di dipendenze
22         for(TintParserOutput.SentenceDependencyTint relation: dependencies){
23             //in Tint gli indici partondo da 1, mentre in Java partono da 0
24             int head = relation.getGovernor()-1; //indice parola head
25             int dep = relation.getDependent()-1; //indice parola dipendenza
```

```
26              //Estraggo il tipo di relazione (label)
27              String label_edge = relation.getDep();
28              //L'etichetta root indica qual'è la testa della frase
29              if(label_edge.toLowerCase().equals("root"))
30                  root = dep;
31              else
32                  /*Il nodo associato al dipendente diventa figlio del nodo
33                  associato alla testa*/
34                  word_nodes[head].addChild(word_nodes[dep]);
35              //Annoto l'etichetta della relazione sul nodo figlio
36              word_nodes[dep].getEl().setDepRelation(label_edge);
37          }
38          //Restituisco la radice dell'albero
39          return word_nodes[root];
40      }
41  }
```

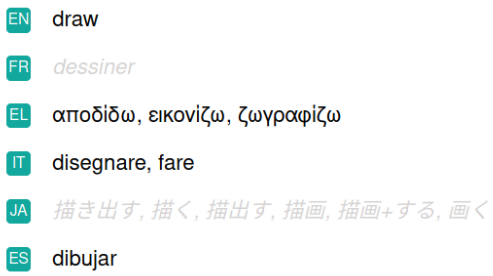
Listing 3: Creazione **albero a dipendenze** a partire dalla frase in ingresso.

3 Transfer

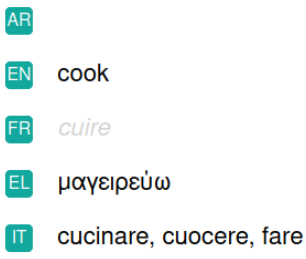
Una volta ottenuto l'albero di parsing per la frase nella lingua sorgente, dobbiamo disporre di opportune regole/fonti di conoscenza che ci consentano di attuare il Transfer lessicale e sintattico.

3.1 Transfer lessicale

Denotiamo con I il lessico 1 dichiarato nella Sottosezione 1.1.Possiamo notare come non tutti i lemmi siano monosemici. Il caso più evidente è dato dal verbo polisemico "fare", il quale in relazione al contesto può assumere un significato (*senso*) differente. In Figura vengono riportate due possibili accezioni ottenute consultando la risorsa Babelnet, che evidenziano come la traduzione debba avvenire sulla base del particolare senso del lemma.



(a) Gloss: Drawing an artifact



(b) Gloss: Transform and make suitable for consumption by heating

Figure 4: Due Babel-synset in cui compare il lemma "fare". In ogni synset, BabelNet fornisce l'elenco di sinonimi nelle varie lingue ed una definizione (Gloss) del significato sotteso

Per evitare di dover svolgere Word Sense Disambiguation e quindi di dover analizzare il contesto in cui la parola occorre, sono state fatte le seguenti ipotesi semplificative:

- I termini polisemici i cui lemmi appartengono ad I assumono un significato definito a priori e non dipendente dal contesto

- La **traduzione** dei lemmi è una relazione funzionale ed ovunque definita R tra gli elementi di I ed i lemmi della lingua inglese appartenenti all'insieme denotato con E. Ovvero vale la seguente condizione:

$$\forall x \in I \quad \exists! y \in E \quad t.c. (x, y) \in R$$

In questo modo per la traduzione possiamo ricorrere ad un dizionario isomorfo italiano-inglese, implementabile in Java con un HashMap.

```
1  private Dizionario() {
2          this.items = new HashMap<String, String>() {{
3              put("essere", "be");
4              put("la", "the");
5              put("spada-laser", "light-saber");
6              put("di","of");
7              put("tuo", "your");
8              put("padre", "father");
9
10             put("avere", "have");
11             put("fare", "make");
12             put("una","a");
13             put("mossa","move");
14             put("leale","fair");
15
16             put("il","the");
17             put("ultimo","last");
18             put("avanzo","remnant");
19             put("del","of");
20             put("vecchio","old");
21             put("repubblica","republic");
22             put("spazzare","sweep");
23             put("via","out");
24         }};
25     }
```

Listing 4: Dizionario isomorfo IT->EN

Nel Transfer lessicale, visitiamo ciascun nodo dell'albero tramite ricerca in ampiezza, ed estraiamo il lemma della parola associata al nodo.
In generale, viene tradotto direttamente l'intero lemma utilizzando il dizionario sopra definito. Tuttavia, nel caso in cui la parola fosse una preposizione, anziché tradurre direttamente l'intero lemma, traduciamo i morfemi con l'uso del suddetto dizionario e poniamo uno spazio tra i morfemi tradotti per generare il risultato. Ad esempio, data la preposizione "della", individuiamo i morfemi "del-la" derivanti dall'analisi morfologica. Traduciamo quindi i singoli morfermi (del->of, la->the) in modo da ottenere la traduzione "of the".
Il risultato della traduzione viene salvato in un attributo del nodo corrente visitato.

3.2 Transfer sintattico

Il processo di generazione, che rappresenta la fase finale della nostra pipeline di elaborazione (Fig. 1), è affidato alla libreria Java SimpleNLG [GR09], un **realizzatore linguistico** che utilizzando regole grammaticali dell'inglese (morfologiche e sintattiche) converte la rappresentazione astratta della frase (**sentence plan ibrido**) data in input in testo effettivo.

Pertanto, nel nostro caso il ruolo del Transfer Sintattico è quello di fungere da ponte tra l'albero a dipendenze finora discusso ed il sentence plan ibrido richiesto in input da SimpleNLG.

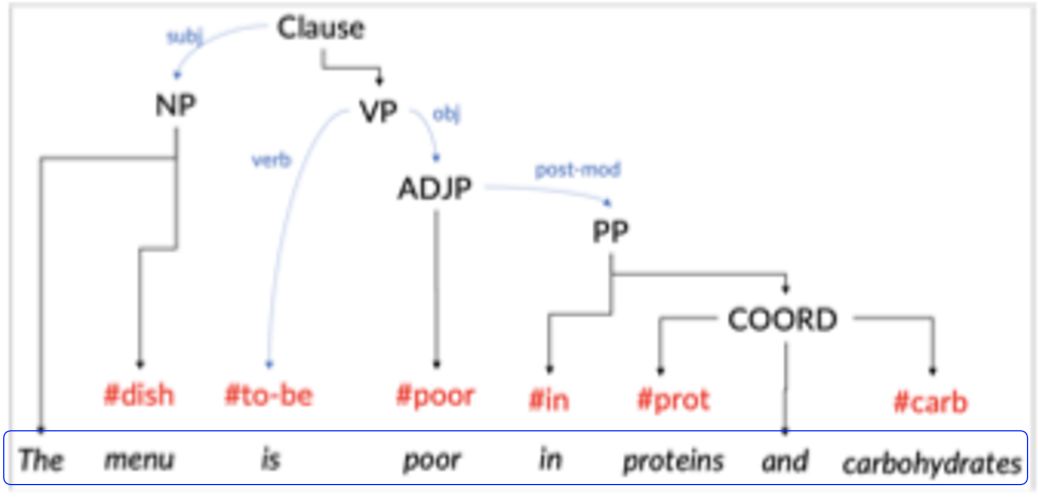


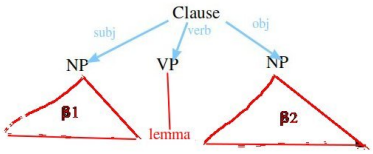
Figure 5: Nella parte alta viene raffigurato un sentence plan ibrido, nell'ultimo livello viene presentato il risultato della realizzazione linguistica (testo)

Ci limitiamo a fare alcune considerazioni sulle peculiarità di quest'ultima struttura dati citata, considerando l'esempio in Figura 5:

- Vi sono relazioni binarie asimmetriche grammaticali tra la frase ed i costituenti.
  Nell'esempio la relazione *subj* lega la frase ad un NP (sintagma nominale)
- Si può specificare relazioni binarie asimmetriche grammaticali anche tra i costituenti.
  Nell'esempio la relazione *post-mod* lega il sintagma aggettivale (dominante) ad un PP (modificatore).
- Le foglie dell'albero ospitano i lemmi in lingua inglese. Sarà compito di SimpleNLG effettuare la generazione morfologica sulla base delle informazioni linguistiche fornite (ad esempio, se il soggetto è in terza persona singolare ed il tempo verbale è al presente, verrà aggiunto il suffisso "-s" al verbo)
- In questa rappresentazione astratta della frase, l'ordine delle parole non c'è in quanto sarà stabilito da SimpleNLG al momento della generazione, in maniera conforme alle regole morfo-sintattiche dell'inglese

Descriviamo ora il processo di trasformazione dell'albero a dipendenze in sentence plan ibrido, supponendo che la frase in ingresso sia in forma attiva e che il verbo sia transitivo[8]..

[8]NOTA BENE: per rendere l'esposizione più chiara, la descrizione si è focalizzata al caso in cui la frase fosse attiva ed il verbo transitivo. Tuttavia, il programma dispone di opportune regole per gestire casi relativi a frasi in forma passiva, predicati nominali, ... etc.



(a) Sentence plan ibrido

```
//*---SOGGETTO---*
if(info.getDepRelation() == RELATION.NSUBJ) {
    //Nodo NP
    NPPhraseSpec NP_subj = factory.createNounPhrase();
    //Assegno relazione subj
    sentence.setSubject(NP_subj);
    //Genero in maniera ricorsiva il sottoalbero Beta1
    NP_subj = (NPPhraseSpec) generatePhrase(node).
        getPhrase();
}
//*---VERBO---*
if(info.getPos() == POS.VERB) {
    //NODO VP
    VPPhraseSpec VP = factory.createVerbPhrase();
    //Assegno relazione verb
    sentence.setVerb(VP);
    //Aggiungo ai figli di VP un nodo foglia (lemma)
    VP.setVerb(info.getEnglishLemma());
}
//*---COMPLEMENTO OGGETTO---*
If(info.getDepRelation() == RELATION.DOBJ){
    //Nodo NP
    NPPhraseSpec NP_obj = factory.createNounPhrase();
    //Assegno relazione obj
    sentence.addComplement(NP_obj);.
    //Genero in maniera ricorsiva il sottoalbero Beta2
    NP_obj = (NPPhraseSpec) generatePhrase(node).
        getPhrase();
}
```

(b) Snippet di codice costruzione sentence-plan ibrido a partire dall'albero a dipendenze

Figure 6: Da una parte porzione del codice utilizzato per trasformare l'albero a dipendenze in un sentence plan. Dall'altra rappresentazione grafica della struttura dati generata.

In primo luogo, si esaminano i nodi dell'albero a dipendenze posti ad una profondità massima di 1 per determinare le relazioni grammaticali della frase (*subj*, *verb*, *obj*) che sono di interesse per il sentence plan ibrido (Figura 6 (a)). Ognuna di queste relazioni coinvolge un opportuno costituente: si costruisce il sotto-albero sotteso (nell'esempio β_1 per il soggetto, β_2 per il complemento oggetto) richiamando la funzione *generatePhrase* (Listing 6 (b)) passandogli il nodo dell'albero a dipendenze appena visitato. La funzione tratta tale nodo come una radice di un sotto-albero, che verrà visitato nell'ottica di ottenere le informazioni utili per generare il sotto-albero del sentence plan ibrido.
Inoltra, una peculiarità della funzione *generatePhrase* è quella di riuscire a gestire la seguente ricorsione:

$$NP \longrightarrow Det\ Noun\ PP \mid Det\ Noun$$

$$PP \longrightarrow Prep\ NP$$

Una volta costruito il sentence plan, SimpleNLG si occuperà di generare la frase per l'inglese, come già discusso all'inizio di questa sezione.

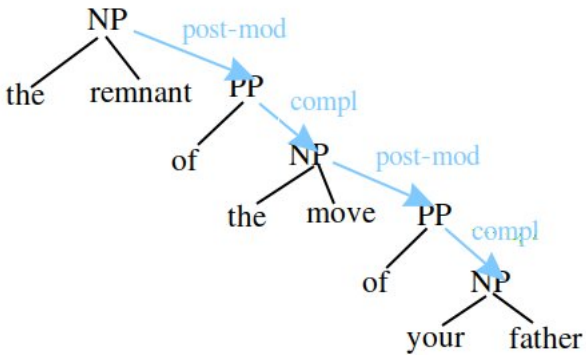


Figure 7: Struttura ricorsiva nel sentence plan ibrido

4 Conclusioni

Sono stati svolti gli esperimenti di traduzione su quattro frasi, come mostrato in Figura 8.

```
'
1-È la spada-laser di tuo padre
2-Ho fatto una mossa leale
3-Gli ultimi avanzi della vecchia Repubblica sono stati spazzati via
4-Feci gli ultimi avanzi della vecchia mossa della leale repubblica di tuo padre
0-exit

Seleziona una frase: 1
 ->  It is the light-saber of your father.

Seleziona una frase: 2
 ->  I have made a fair move.

Seleziona una frase: 3
 ->  The last remnants of the old republic have been swept out.

Seleziona una frase: 4
 ->  I made the last remnants of the old move of the fair republic of your father.
```

Figure 8: Console Java durante esecuzione del programma

Possiamo quindi concludere dicendo che l'approccio simbolico basato su TransferSintattico ha generato buoni risultati.
La Neural Machine Translation oggigiorno è il modello attualmente più in voga date le sue performances, tuttavia necessita di molti dati a disposizione, sotto forma ad esempio di pagine multilingua da cui poter apprendere. Gli approcci simbolici hanno il vantaggio di basarsi esclusivamente su regole, quindi si può sulla carta tradurre anche lingue poco diffuse sul Web. Inoltre, nelle reti neurali le decisioni prese della macchina non sono trasparenti e spesso incomprensibili anche agli occhi degli esperti, mentre negli approcci simbolici i processi di che hanno portato ad un determinato output sono interpretabili e direttamente correggibili in caso di problemi.

References

[GR09] A. Gatt and E. Reiter. Simplenlg: A realisation engine for practical applications. *Proceedings of the 12th European Workshop on Natural Language Generation*, pages 90–93, 2009.

[JM09] Dan Jurafsky and James H. Martin. *Speech and language processing : an introduction to natural language processing, computational linguistics, and speech recognition.* Pearson Prentice Hall, Upper Saddle River, N.J., 2009.

[PM18] A. Palmero Aprosio and G. Moretti. Tint 2.0: an all-inclusive suite for nlp in italian. *Proceedings of the Fifth Italian Conference on Computational Linguistics CLiC-it*, 10:1–12, 2018.