Sistemi Cognitivi

Progetto di traduzione automatica interlingua

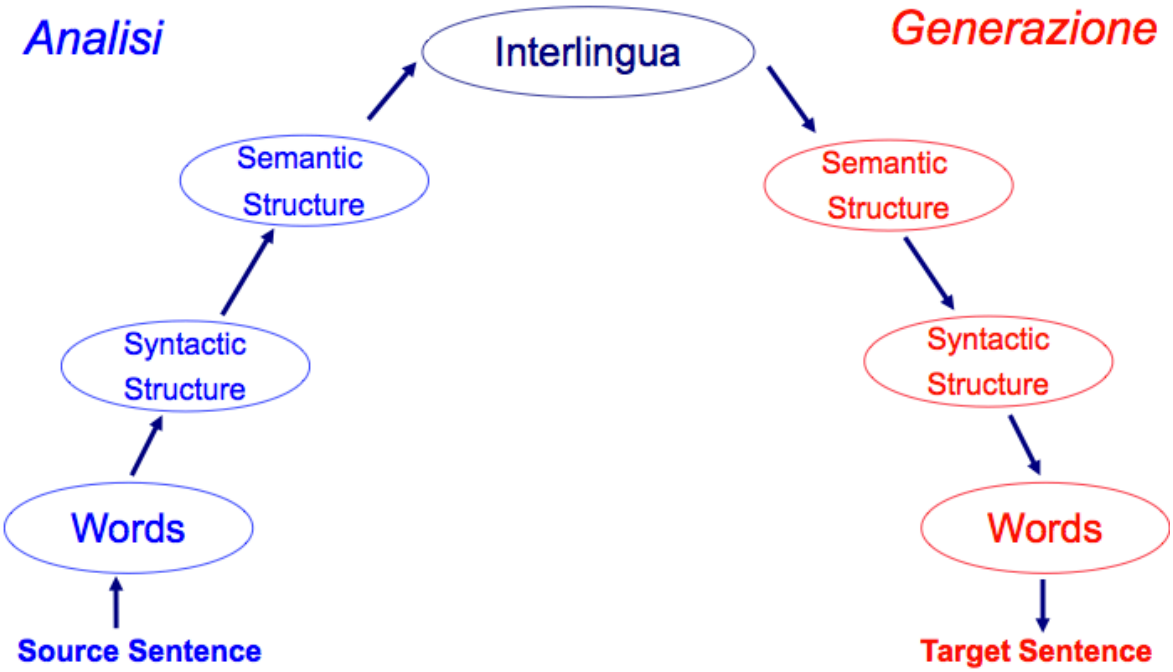
Andrea Pacino

Il Progetto

Il progetto consiste nella realizzazione di un traduttore automatico interlingua.

La traduzione interlingua è il livello più profondo della traduzione automatica nel Triangolo di Vauquois. Ciò permette di effettuare un'analisi più approfondita delle frasi da tradurre.

Triangolo di Vauquois:



L'analisi viene effettuata a livelli. Abbiamo il livello delle parole, il livello sintattico ed il livello semantico. Tali livelli vengono ripercorsi al contrario per la generazione della frase tradotta.

Nel nostro caso la traduzione viene effettuata su un insieme di frasi in italiano da tradurre in inglese.

Le fasi

Il progetto consiste in 5 fasi differenti che portano alla traduzione delle frasi:

1. Scrivere una Context-Free Grammar (G1) per l'italiano che parsifichi le frasi contenute nel file Sentences1.txt
2. Implementare in Java l'algoritmo CKY usando la grammatica G1 per parsificare le frasi contenute nel file.
3. Dotare di semantica (G2) la grammatica G1 e provarla in NLTK
4. Costruire un sentence planner che per ogni formula prodotta dalla grammatica G2 produca un sentence plan (abstract-syntax tree) valido.
5. Unsando la libreria SimpleNLG implementare un semplice realizer che trasformi i sentence plans in frasi inglesi.

Descrizione

Nella prima fase bisogna costruire la **Context-Free Grammar** per le frasi contenute nel file Sentence1.txt verrà salvata nel file di testo G1.txt.
La Context-Free Grammar è in *Forma Normale di Chomsky* ovvero:

- $A \rightarrow BC \; con \; B, C \; \in simoboli \; non \; terminali \; della \; grammatica$
- $A \rightarrow a \; con \; a \; \in alfabeto \; dei \; simboli \; terminali \; della \; grammatica$

I simboli non terminali si distinguono in:

- S *indica il simbolo di start;*
- NP *indica un sintagma nominale;*
- VP *indica un sintagma verbale;*
- Con *indica una congiunzione;*
- Num *indica una quantificazione;*
- Art *indica un articolo;*

- Prep *indica una preposizione;*
- Dim *indica un aggettivo dimostrativo.*

L'alfabeto dei simboli terminali invece rappresenta tutte le singole parole contenute nelle frasi.

Nella seconda fase bisogna costruire l'algoritmo di **CKY**.
L'algoritmo CKY utilizza le grammatiche Context-Free in forma normale di Chomsky, l'algoritmo è di tipo bottom-up con programmazione dinamica ed utilizza un oracolo basato su regole.
Calcola tutti i possibili parse in tempo $O(n^3)$. Tale algoritmo crea una tabella in cui sulla diagonale inseriamo il simbolo non terminale relativo ad ogni parola della frase e mano a mano cerchiamo ogni relazione tra i vari simboli non terminali. Se al termine dei 3 cicli *for* (da qui ne deriviamo la complessità) nella posizione $[0][n]$ della tabella non vi è inserito il simbolo non terminale, ed iniziale, "S". Allora tale frase non può essere parsificata dalla grammatica G1.
La classe *CKY* di *Java* consiste nel metodo *main* e nei metodi *readGrammar*, *readSentences* ed il metodo *printCKY*.
Il metodo *main:*

- Inizializza una *HashMap<String, String>* in cui inseriremo tutte le regole della grammatica;
- Popola l'*HashMap* con il metodo *readGrammar;*
- Popola l'*ArrayList<String>* che conterrà le singole frasi del file Sentence1.txt con il metodo *readSentences;*
- Stampa la frase analizzata e la relativa tabella con il metodo *printCKY.*

Il metodo *readGrammar* legge dal file G1.txt ogni regola della grammatica inserendole nell'*HashMap<String, String>* che conterrà tutte le regole grammaticali. Nella chiave sarà contenuto il simbolo terminale o i 2

simboli non terminali a destra di ogni regola ed il valore conterrà il simbolo non terminale a sinistra di ogni regola (es. <"VP NP","S">).
Il metodo *readSentences* legge dal file Sentences1.txt ogni frase e le inserisce in un *ArrayList<String>* che verrà restituito alla classe chiamante.
Il metodo *printCKY* legge le frasi da parsificare e le parsifica tramite il metodo *cky* che implementa l'algoritmo CKY, spiegato in precedenza, che restituisce la tabella che verrà stampata con la frase parsificata.

Nella terza fase dotiamo di **semantica** la grammatica G1. Tale semantica verrà salvata in un file "*G2.fcfg"* ed è scritta in λ-First Order Logic (FoL) per rappresentare al meglio il significato semantico di una frase. La FoL si basa su predicati e costanti e fa un mapping fra oggetti formali e mondo reale. La lambda abstraction serve per specificare l'ordine di applicazione delle regole semantiche.
Dopo la specifica della semantica in FoL utilizziamo NLTK per trasformare le frasi contenute nel file in regole logiche che verranno salvate in un file "*G2.txt".*

Nella quarta fase costruiamo un **sentence planner** in *Java* che per ogni formula prodotta dalla grammatica G2 ci restituirà un sentence plan valido.
In questa fase si costruisce un *Abstract-Syntax tree* a dipendenze per ricavarne il sentence plan.
La classe *Planner* tramite il metodo *createTree* costruisce un l'AS-tree con il solo nodo *Root* e crea un nodo figlio tramite il metodo *createNode* che prende come parametri la semantica da analizzare ed il nodo padre a cui far riferimento.

Il metodo *createNode* analizza la semantica mediante espressioni regolari create ad hoc per i predicati verbali o nominali, le congiunzioni, i quantificatori "*per ogni"* ed "exists". Prima si controlla se siano presenti quantificatori, successivamente le congiunzioni ed infine i predicati.
Se c'è match tra la semantica ed una espressione regolare si procede alla memorizzazione di tutte le occorrenze in un *ArrayList<String>* ed alla rimozione di esse dalla semantica da analizzare. Ad ogni elemento nell'*ArrayList* si effettua una chiamata ricorsiva al *createNode* per la generazione dei nodi figli di tale relazione. L'analisi prosegue finché non sarà più possibile estrarre informazioni dalla semantica.
Nell'analisi di un predicato (*verb(subj,obj,pp)*) possiamo avere 3 forme diverse:

- Un solo argomento: abbiamo solo il soggetto;
- Due argomenti: abbiamo soggetto ed oggetto;
- Tre argomenti: abbiamo soggetto, oggetto e predicato;

Nella quinta fase bisogna implementare un **reasoner**, mediante la libreria SimpleNLG, che trasformi i sentence plan in frasi in inglese. La classe *TheReasoner* mediante il metodo *getSentence* che prende come attributo un AS-tree visitato in *preOrderTraversal* ed analizza l'albero salvando in una *HashMap<String, NLGelement>* gli elementi terminali ed in un'altra le parti della frase da comporre che poi saranno inserite nel *SPhraseSpec* per specificare ogni parte della frase che sarà "*realizzata*" e restituita.