Grammatica Klingon con Parsing CKY e Semantica

Indice

Componenti del gruppo

- **Gianluca Barmina** 884084
- **Marco Amato** 882348

Descrizione del Progetto

Questo progetto è composto di 4 parti:

- Implementazione dell'algoritmo di parsing sintattico CKY.
- Testare l'algoritmo CKY su una grammatica a costituenti per la lingua inglese (L1 di Jurafsky)
- Implementazione di una grammatica a costituenti per la lingua Klingon e parsificazione di alcune frasi sfruttando CKY.
- Implementazione di una grammatica con semantica per la lingua Klingon e calcolo della semantica di alcune frasi sfruttando un parser semantico di NLTK.

Contenuti

Il progetto è strutturato nel seguente modo:

1. **CKYParser.py:** Contiene il codice Python che implementa l'algoritmo CKY per il parsing sintattico.
2. **resources.py:** Contiene il codice Python che implementa le strutture utilizzate per:
    - Grammatica L1 di Jurafsky
    - Grammatica per la lingua Klingon, scritte in Chomsky Normal Form.
    - Grammatica per la lingua Klingon con semantica
    - Frasi in inglese e Klingon utilizzate per i test
3. **main.py:** Contiene il codice Python che esegue i vari test.

Utilizzo

```
python main.py [0-1]
```

Che avvia i seguenti test:

- Parsing sintattico di due frasi in inglese tramite CKY, utilizzando la grammatica L1 di Jurafsky.
- Parsing sintattico delle 4 frasi Klingon come da requisito di consegna, utilizzando la grammatica Klingon che abbiamo ideato e l'implementazione di CKY.
- Parsing semantico delle stesse frasi Klingon, utilizzando la grammatica con semantica per il Klingon, da noi sviluppata.

Il parametro opzionale `0` o `1` permette di scegliere se stampare ogni step dell'esecuzione di CKY o meno.

Note aggiuntive

- La grammatica Klingon può non essere completa e potrebbe richiedere ulteriori regole per coprire una vasta gamma di frasi.

Task 1.A: Parser CKY

La versione di CKY implementata è quella non-probabilistica e rule-based. L'input consiste in

- Una frase su cui effettuare il parsing
- Una grammatica context free, a costituenti in Chomsky Normal Form

Modellazione della frase

La frase è stata modellata come una lista di stringhe, in cui ogni parola è una elemento della lista

Modellazione della grammatica

La grammatica è stata modellata come una lista di tuple. Ogni tupla è composta da:

- Una stringa rappresentante la parte sinistra della regola. Questa stringa rappresenta un simbolo non terminale.
- Una lista di stringhe rappresentante la parte destra della regola. Ognuna delle stringhe rappresenta un simbolo terminale o non terminale.

La grammatica fornita è in formato CNF, quindi:

- Ogni tupla ha una lunghezza di 1 o 2
- Se la lunghezza è 2, allora la tupla è nella forma `(A, [B, C])`, dove `A`, `B` e `C` sono simboli non terminali
- Se la lunghezza è 1, allora la tupla è nella forma `(A, [b])`, dove `A` è un simbolo non terminale e `b` è un simbolo terminale
- E' ammessa solo la ricorsione a sinistra

Algoritmo CKY

L'obiettivo dell'algoritmo è quello di creare una struttura sintattica per la frase fornita in input. Il successo dell'algoritmo ci dice due informazioni:

- Data la grammatica e la frase esiste un albero sintattico che rappresenta quella frase

- La frase è sintatticamente corretta rispetto alla grammatica data

L'algoritmo sfrutta la programmazione dinamica per riempire una matrice quadrata di dimensione `n x n`, dove `n` è la lunghezza della frase. Ogni colonna della matrice corrisponde ad una parola della frase.

L'idea fondamentale dell'algoritmo è che data una regola `A -> BC`, possiamo considerare la produzione di destra delimitata da due estremi: `i` e `j`. A questo punto, possiamo dividere la produzione in due parti, costituite dai due sintagmi: `B` e `C`. Considerando ciò, esiste una posizione `k`, tale che `i < k < j`, che divide la produzione in due parti:

- da `i` a `k` ci sarà `B`
- da `k` a `j` ci sarà `C`

Sfruttando questa divisione la tabella verrà costruita in modo da avere:

- Nella posizione `[i][j]` il simbolo non terminale `A`
- Nella posizione `[i][k]` il simbolo non terminale `B`
- Nella posizione `[k][j]` il simbolo non terminale `C`

Questo per rappresentare che il sintagma `A` (lungo da `i` a `j`) produce `B` (lungo da `i` a `k`) e `C` (lungo da `k` a `j`)

Mantenendo questa idea si può intuire che le parti sinistre delle regole che producono i terminali (ossia le parole della frase) verranno inserite nella diagonale principale della matrice.
Considerando una regola `A -> b` stiamo infatti dicendo che il sintagma `A` (lungo da `i` a `j`) produce il terminale `b` (lungo da `i` a `j`). Tuttavia in questo caso sarà prodotta esattamente una parola, che avrà quindi lunghezza unitaria, ossia `i = j`, e sicuramente non potrà essere ulteriormente divisa.
Dato che ogni posizione della diagonale avrà `i = j`, ogni posizione `[i][j]` conterrà il simbolo non terminale che produce la parola in posizione `j` della frase.

Il riempimento della matrice avviene sfruttando tre indici: `i`, `j` e `k`.

- `j` rappresenta la colonna della matrice.
- `i` rappresenta la riga della matrice. Fissata la colonna `j`, `i` viene decrementato scorrendo le posizioni della colonna `j` in ordine decrescente per riempirle.
- `k` rappresenta la posizione in cui viene divisa la frase. Scorre fra `i` e `j` per trovare la posizione in cui dividere la produzione di destra della regola.

L'algoritmo è strutturato in tre cicli annidati

Primo ciclo: j

Il ciclo più esterno scorre le colonne della matrice in modo crescente e ha il compito di riempire la diagonale principale. In particolare per ogni colonna (ossia per ogni parola della frase) viene controllato se esiste una regola `A -> b` tale che `b` è la parola in posizione `j` della frase. Se questa esiste allora viene inserito il simbolo non terminale `A` nella posizione `[j][j]` della matrice. Ossia nell'elemento della diagonale principale che corrisponde alla colonna (parola) `j-esima`.

Essendo i successivi cicli annidati, il riempimento non viene fatto in un colpo solo.
Nel momento in cui ci sono posizioni valide per l'avvio dei cicli successivi, il riempimento della diagonale si ferma e si passa al riempimento delle altre posizioni della matrice. Il riempimento della diagonale proseguirà poi quando i cicli successivi verranno terminati.

Secondo ciclo: i

Il ciclo interno successivo, una volta fissata la colonna `j`, scorre le righe della matrice, partendo dalla diagonale principale e arrivando alla riga `0`.
Il compito del ciclo è di permettere il riempimento delle colonne della matrice. Ogni colonna viene riempita in modo decrescente partendo dalla fondo (diagonale principale) e arrivando alla riga `0`.

Questo ciclo sfrutta il fatto che la posizione della diagonale principale per la colonna `j`, considerata ogni volta, è `j` stesso, quindi inizialmente `i = j`.

Un'altra funzione fondamentale di questo ciclo è fissare un estremo sinistro della frase dopo che il precedente ha fissato l'estremo destro. Fissati entrambi, il ciclo successivo potrà scorrere le posizioni intermedie della frase per individuare la presenza di regole che generano la frase definita dagli estremi `i` e `j`.

Terzo ciclo: k

Una volta fissati gli estremi `i` e `j`, il ciclo più interno scorre le posizioni intermedie fra essi per trovare tutte le possibili posizioni `k` in cui dividere la frase. Ossia individuare i sintagmi della frase (separati in posizione `k`) tali per cui esiste una regola che li genera.

A questo punto stiamo considerando sicuramente sintagmi composti non da singole parole. Questo perché se fossero singole parole, allora sarebbero già state inserite nella diagonale principale. Di conseguenza andiamo a considerare solo le regole della forma `A -> BC`, dove `B` e `C` sono simboli non terminali.

Dati gli estremi `i` e `j`, l'obiettivo è individuare tutti valori di `k` tali per cui `i < k < j` ed esista una regola `A -> BC` tale che `B` è nella posizione `[i][k]` e `C` è nella posizione `[k][j]`. Più in particolare ciò che ci interessa è il non terminale `A`, che verrà inserito nella posizione `[i][j]` della matrice.

Addentrandoci nel codice, in questo ciclo, avendo fissato gli estremi `i` e `j`, e una certa posizione intermedia `k`, si considera ogni regola della grammatica della forma `A -> BC`. Per ogni **parte destra della regola** in particolare chiamiamo `left` la parte sinistra di essa (primo sintagma generato) e `right` la parte destra (secondo sintagma generato). Verifichiamo se `left` è presente nella posizione `[i][k]` e se `right` è presente nella posizione `[k][j]`. Se entrambe le condizioni sono verificate, ciò significa che:

- Abbiamo trovato una regola che genera la frase definita dagli estremi `i` e `j`
- Abbiamo trovato la posizione `k` in cui dividere la frase
- Sappiamo che il primo sintagma della parte destra della regola (ossia `B`) è nella posizione `[i][k]`
- Sappiamo che il secondo sintagma della parte destra della regola (ossia `C`) è nella posizione `[k][j]`

Considerando ciò, per rappresentare che questa regola produce la frase, inseriamo il simbolo non terminale `A` nella posizione `[i][j]` della matrice.

Ogni volta che per una certa posizione intermedia `k` vengono analizzate tutte le regole, ed eventualmente vengono aggiunte alla matrice, il ciclo termina e si passa alla posizione intermedia successiva.
Ogni volta che per un certo estremo sinistro `i` vengono considerate tutte le posizioni intermedie, il ciclo termina e si passa all'estremo sinistro successivo, scorrendo verso `j`.

- Ciò si traduce nello scorrimento e riempimento della colonna `j-esima` verso l'alto, ossia verso `0`

Ogni volta che per un certo estremo destro `j` vengono considerate tutte le possibilità di estremi sinistri e posizioni intermedie, il ciclo termina e si passa all'estremo destro successivo, scorrendo verso `n-1`.

- Ciò si traduce nello scorrimento da sinistra a destra delle colonne. Ogni colonna sarà poi considerata dal ciclo successivo.

L'effetto di tutto ciò è che la frase va ad aumentare di grandezza, considerando sempre più parole, tramite lo spostamento dell'estremo destro. Ugualmente avviene, fissato l'estremo destro, con quello sinistro.

Risultato dell'algoritmo

Data una frase ed una grammatica, l'algoritmo effettua con successo il parsing della frase se nella cella `[0][n-1]` della matrice è presente il simbolo non terminale `S` che rappresenta la radice dell'albero sintattico.
La presenza di questo simbolo come anticipato prima indica che la frase è sintatticamente corretta rispetto alla grammatica data e che sia stato trovato un albero sintattico che la rappresenta. L'albero sintattico può essere ricostruito a partire dalla matrice, partendo dalla cella `[0][n-1]`.

NOTA: Nello pseudo-codice visto a lezione e presente nel libro di testo, il primo indice delle righe è `0` mentre il primo indice delle colonne è `1`.
Nella nostra implementazione abbiamo scelto per comodità e naturalezza di utilizzare indici che partono da `0` sia per le righe che per le colonne. Le conseguenti modifiche apportate rispetto allo pseudo-codice sono:

- Il primo ciclo fa scorrere l'indice `j` da `0` (non da `1`) a `n - 1`, dove `n` è la lunghezza della frase
- Il secondo ciclo fa scorrere l'indice `i` da `j-1` (non da `j-2`) a `0`
- Il terzo ciclo fa scorrere l'indice `k` da `i` (non da `i+1`) a `j-1`
- Quando nella tabella si valuta la presenza del secondo sintagma della parte destra della regola (`C`), si utilizza l'indice `k+1` (non `k`)

Task 1.B: CKY su grammatica L1 di Jurafsky

Il primo task su cui è stato eseguita l'implementazione del parser CKY sono state due frasi in lingua inglese.

- *book the flight through Houston*
- *does she prefer a morning flight*

La grammatica utilizzata è la grammatica L1 di Jurafsky, ridotta alle componenti necessarie per il parsing delle due frasi.
In particolare questa grammatica rispetta un requisito fondamentale per poter funzionare con questa tipologia di Parser, ossia l'essere in Chomsky Normal Form. Come descritto nella sezione precedente quindi avrà solo regole che generano due non terminali oppure un singolo terminale, e avrà ricorsione solo a sinistra.

Per ogni frase, l'output prodotto dal parser consiste in una tabella.
Il punto della tabella su cui porre attenzione è la posizione in alto a destra (`[0][n-1]` dove `n` è la lunghezza della frase). Per ogni frase la tabella in output presenta il simbolo `S` nella posizione indicata. Ciò sta a significare che ognuna è quindi sintatticamente corretta rispetto alla grammatica utilizzata.

Intepretazione della tabella

In questa sezione verrà fatta l'interpretazione di alcune celle della tabella rispetto ad un esempio di applicazione. Consideriamo la frase *book the flight through Houston*

Il risultato prodotto è il seguente:

```
    ------------- 0 ------------------  -1-     -------- 2 -------      -3-     --
---------------- 4 ------------------
0|  ['S', 'VP', 'V', 'Nominal', 'Noun'] []      ['S', 'VP', 'X2']   []      ['S',
'VP', 'X2', 'S', 'S', 'VP', 'VP']
1|  []                                  ['Det'] ['NP']              []      ['NP']
2|  []                                  []      ['Nominal', 'Noun'] []
['Nominal']
3|  []                                  []      []                  ['P']   ['PP']
4|  []                                  []      []                  []      ['NP',
'PN']
```

Considerando la seguenti celle:

- `[3][4]` contenente il simbolo `PP`
- `[3][3]` contenente il simbolo `P`
- `[4][4]` contenente il simbolo `NP`

Il `PP` è il sintagma che "produce" la frase dalla posizione `3` alla `4`, di conseguenza si trova nella posizione `[3][4]`. Esso è composto dal sintagma che "produce" la frase nella posizione `3` della frase (`P`), e quindi nella posizione `[3][3]` della tabella, e dal sintagma che "produce" la frase nella posizione `4` della frase (`NP`), e quindi nella posizione `[4][4]` della tabella.

Ovviamente la conoscenza del fatto che `PP` è composto da `P` e `NP` è data dalla grammatica utilizzata.
Ciò significa che nella grammatica sarà presente una regola: `PP -> P NP`. Quindi un altra interpretazione che può essere fatta è che nella pozione `[3][3]` è contenuto il primo sintagma generato e nella posizione `[4][4]` è contenuto il secondo. Mentre nella posizione `[3][4]` è contenuta la parte sinistra della regola `S -> VP PP` che genera i due sintagmi.

Considerando ora le seguenti celle:

- `[2][4]` contenente il simbolo `Nominal`
- `[2][2]` contenente il simbolo `Nominal`
- `[3][4]` contenente il simbolo `PP`

Il `Nominal` è il sintagma che "produce" la frase dalla posizione `2` alla `4`, di conseguenza si trova nella posizione `[2][4]`. Esso è composto dal sintagma che "produce" la frase nella posizione `2` (`Nominal`), e quindi nella posizione `[2][2]`, e dal sintagma che "produce" la frase dalla posizione `3` alla `4` (`PP`), e quindi nella posizione `[3][4]`.

Anche in questo caso la conoscenza del fatto che `Nominal` genera `Nominal` e `PP` è data da una regola presente nella grammatica: `Nominal -> Nominal PP`. Quindi in `[2][2]` è contenuto il primo sintagma generato, in `[3][4]` il secondo, e unendoli considerando anche le posizioni della frase generate, in `[2][4]` è contenuto il sintagma (parte sinistra della regola) che li genera.

Per questa frase, nella posizione `[0][4]` il simbolo `S` è presente tre volte. Ciò significa che per la grammatica data sono possibili tre alberi sintattici distinti che generano la frase (quindi si hanno tre radici possibili). Una possibilità per distinguere i tre alberi è l'applicazione del backtracking per tenere traccia delle diverse catene di produzioni che portano alla generazione della frase.

Il fatto che poi, oltre ad `S`, in posizione `[0][4]` siano presenti anche altri simboli, come ad esempio `VP`, indica che l'intera frase è generata anche da altri sintagmi, e quindi, per esempio, che la frase stessa sia un `VP`.

Task 1.C: Grammatica Klingon

Le regole della grammatica Klingon che abbiamo scritto sono un sottoinsieme di tutte quelle che dovrebbero essere implementate in una grammatica per frasi Klingon arbitrarie.

In particolare ci siamo attenuti alle regole sufficienti a parsificare le frasi nella consegna del laboratorio. Qui indichiamo le scelte più significative.

Gestione nomi e suffissi

I nomi in Klingon possono avere suffissi che possono avere funzioni diverse. Nel caso nostro l'unico suffisso presente era `Daq` a indicare il fatto che il nome sia un luogo e una certa azione si svolga in prossimità di quel luogo.

Abbiamo quindi rappresentato la particella principale del nome con `Noun` come parte sinistra di ogni regola e i relativi terminali come parte destra. I suffissi sono stati rappresentati invece con `NounSuffix` come parte sinistra di ogni regola e i relativi terminali come parte destra.

La regola che poi ci permette di legare i due elementi, componendoli, è `NounCompound -> Noun NounSuffix`.

Il `NounCompund` è poi utilizzato in altre regole, come quelle relative ai verbi o al simbolo di start (`Sentence`) per permettere di strutturare frasi più complesse.

Gestione verbi prefissi e suffissi

I verbi in Klingon possono avere sia suffissi che prefissi.

- I prefissi indicano la persona e la pluralità di soggetto e complemento oggetto (E.g `Da` prima di un verbo indica che il soggetto è in seconda persona singolare, e che il complemento oggetto è in terza persona singolare)
- I suffissi indicano dei qualificatori per le azioni del verbo o per la frase intera (E.g. Quando un verbo viene seguito da `'a'` la frase è in forma interrogativa

Quindi abbiamo previsto un sintagma che racchiuda il verbo + eventuali prefissi o suffissi, e lo abbiamo chiamato `VerbCompound`.
La parte principale del verbo è chiamata `Verb`, i prefissi sono chiamati `VerbPrefix` e i suffissi `VerbSuffix`

La composizione dei verbi con i prefissi e suffissi è data da regole che hanno `VerbCompound` come sintagma a sinistra.

È possibile avere: verbi senza suffisso e senza prefisso, verbi con solo prefisso, verbi con solo suffisso o verbi con entrambi. Per questo sono state realizzate delle regole ad hoc per ogni caso.

Gestione delle radici

La regole relative alle radici possibili hanno parte sinistra `Sentence` e sono state strutturate in modo da rispettare l'ordine delle parole tipico della lingua Klingon: `oggetto-verbo-soggetto`.

Tuttavia, avendo separato la gestione dei verbi e nomi dai relativi prefissi e suffissi, le possibilità di combinazione sono molteplici. Ad esempio si prevede che la frase sia composta da:

- `NounCompound` (nome + suffisso) e verbo
- `NounCompound` (nome + suffisso) e `VerbCompound` (verbo + prefisso e/o suffisso)
- ...

Sono state gestite tutte le possibili combinazioni di nomi, verbi e conseguenti prefissi e suffissi.

Ambiguità tra verbo essere e pronomi personali

In Klingon, fissata una certa numerosità (singolare o plurale) e una certa persona (prima, seconda o terza) i pronomi personali corrispondono al verbo essere coniugato al presente per la stessa persona e numerosità. E.g. `mah` indica sia il pronome personale `noi` che la prima persona plurale del verbo essere al presente `noi siamo`.

Di conseguenza, per frasi che contengono queste casistiche, gli alberi generati considereranno entrambe le opzioni.

Risultato dell'esecuzione di CKY

L'esecuzione di CKY sulle frasi fornite ha mostrato che sono tutte sintatticamente corrette. Questo è evidenziato dal fatto che nella posizione `[0][n-1]` (dove `n`è la lunghezza della frase) della matrice è presente il simbolo `Sentence` che rappresenta la radice dell'albero sintattico.

I risultati ci mostrano anche che, nonostante l'ambiguità tra verbo essere e pronomi personali, non ci sia ambiguità sintattica nel parsing intero di nessuna frase. Questo è mostrato dal fatto che nella posizione `[0][n-1]` della matrice è presente il simbolo `Sentence` solo una volta, ossia è stato generato un singolo albero sintattico.

Task 1.D (extra): Grammatica Klingon con semantica

Qui indichiamo le scelte più significative in merito alla grammatica semantica scritta per la lingua Klingon.

Processo di derivazione della semantica

Il processo di derivazione della semantica che abbiamo seguito si basa sul reverse engineering.

In particolare abbiamo analizzato una frase per volta e abbiamo identificato una formula di logica del prim'ordine che potesse rappresentare la semantica della frase intera.

Una volta effettuato ciò, abbiamo individuato all'interno della formula, le singole sotto-formule che rappresentassero la semantica dei vari sintagmi per poi estrarle e assegnarle ad ogni sintagma. All'occorrenza,

sono stati inseriti operatori lambda per fare il bind di variabili ancora non legate nelle sotto-formule ma che sarebbero state legate in seguito tramite Beta reduction.

Abbiamo poi applicato manualmente le operazioni di Beta reduction utilizzando le parti estratte e individuato quali delle parti fossero le funzioni e quali gli argomenti.

Una volta certi della correttezza logica della formula prodotta tramite Beta reduction, abbiamo riportato le regole di λ-FoL per ogni sintagma all'interno del codice.

Dato che ci siamo fortemente basati sulle frasi fornite, la grammatica con semantica che abbiamo scritto è molto specifica, è quindi corretta per le frasi fornite ma non è detto che lo sia per frasi più complesse.

Determiners

In alcune frasi fornite dalla consegna appaiono nomi comuni. Queste frasi, tradotte in inglese o in italiano, portano il nome accompagnato dall'articolo determinativo (E.g. pa' -> la stanza).

Abbiamo quindi tradotto i terminali di questo tipo in modo da rappresentare il significato di un nome preceduto da un articolo determinativo. Ovvero, se in una frase parlo del bambino (e non di un bambino qualsiasi) e successivamente mi riferisco nuovamente al bambino, a livello semantico mi riferisco alla stessa entità.

Abbiamo chiamato questo sintagma particolare composti da articolo e nome `DetNoun`.

E.g semantica di "The Child"

```
DetNoun[SEM=<λQ. ∃c.((child(c) ^ Q(c)) ^ ∀y.(child(y) -> (c = y)))>] -> "puq"
```

Reificazione degli eventi

In Klingon, suffissi di verbi e nomi possono essere utilizzati per esprimere informazioni aggiuntive che poi si rispecchiano nella semantica della frase. Per gestire il legame tra verbo/nome e suffissi abbiamo usato il meccanismo di reificazione degli eventi, in particolare la versione di Davidson, che non prevede una separazione fra *Agent* e *Patient* dell'azione.

Abbiamo quindi creato una variabile `e` che rappresenta un evento, essa poi si riferisce sia al verbo, sia al predicato rappresentante l'informazione aggiuntiva data dal suffisso. I casi in cui abbiamo utilizzato la reificazione sono:

- Suffisso (di verbo) `'a'`: per esprimere la forma interrogativa. In questo caso la semantica relativa alla creazione dell'evento è stata posta nella parte principale del verbo (`Dajatlh`, ossia "*to speak*")

```
IV[SEM=<λQ λX.∃e.(X(λx.speak(e,you,x)) ^ Q(e))>] -> "Dajatlh"
```

- Suffisso (di nome) `Daq`: per esprimere che il nome è un luogo e l'azione si svolge in prossimità di esso. In questo caso la semantica relativa alla creazione dell'evento è stata posta nella parte principale del

nome (`pa'`). I predicati relativi al verbo e al fatto che l'azione si svolga in prossimità del luogo sono poi inseriti nella composizione tramite beta reduction.

```
DetNoun[SEM=<λQλR.exists r.((room(r) ^ ∃e.(Q(e)(r) ^ R(e)) ^ ∀y.(room(y) -> (r =
y)))))>] -> "pa'"
```