Relazione es. 2 – Valentino Di Cianni

Per questo progetto ho scelto di implementare l'esercizio numero 2 (traduzione IT -> IT-YO) utilizzando come linguaggio JavaScript e Node.js come server environment.

La grammatica utilizzata come test per questo esercizio è stata creata a partire dalle tre frase di esempio fornite, prendendo ispirazione dalle grammatiche già presenti nelle slide del corso. Ho cercato il più possibile di mantenere la sintassi dei file .fcfg così da poterla utilizzare anche con altre librerie come nltk in python. La grammatica risultante è presentata già ridotta in forma normale di Chomsky per semplicità di utilizzo.
Dal momento che non esisteva una libreria JS per gestire il parsing di file del tipo .fcfg , ho implementato una funzione che, preso un file di questo genere, mi ritornasse un oggetto *grammatica* nella forma:

```
1.  {
2.    [corpo_della_produzione_1 : 'testa_della_produzione_1'],
3.    [corpo_della_produzione_2 : 'testa_della_produzione_2'],
4.    .
5.    .
6.    .
7.    [corpo_della_produzione_N : 'testa_della_produzione_N']
8.
9.  }
```

Questa scelta di utilizzare il corpo della produzione come chiave è funzionale all'implementazione dell'algoritmo CKY.

IMPLEMENTAZIONE ALGORITMO CKY

Una prima fase preparatoria prevede la divisione in token della frase in input: questa divisione è fatta molto semplicemente eliminando l'eventuale punteggiatura presente e dividendo i vari token prendendo come riferimento gli spazi bianchi.

Una volta ottenuti i token della frase da analizzare, questi, insieme alla grammatica, vengono passati alla funzione `parse_CKY(words, grammar)`la quale esegue l'algoritmo CKY e ritorna l' albero sintattico di derivazione della stringa in input. In caso di frase non sintatticamente corretta viene restituito un messaggio di errore.

Andando più nel dettaglio riguardo l'implementazione dell'algoritmo, come prima cosa viene richiamata la funzione `createMatrix(dim)` che ritorna una matrice quadrata riempita in ogni posizione con un oggetto nella forma:

```
1.   m[i][j] = {
2.      heads : [],
3.      body: [],
4.      leftTree : [],
5.      rightTree : [],
6.      word : undefined
7.  };
```

Fino a questo passaggio tutti i campi degli oggetti nella matrice sono ancora vuoti.
Nel passo successivo, cominciano ad essere inizializzati gli oggetti della matrice sulla diagonale (in posizione [j-1][j]) con l'aggiunta del campo word, ovvero le parole prese dalle lexical rules della grammatica, e il campo head, ovvero la testa della produzione relativa alla word preso sempre dalle lexical rules.

```
function parse_CKY(words, grammar){
    let numTokens = numWords + 1;
    let table = createMatrix(numTokens);

    for(let j=1; j < numTokens; j++){
        table[j-1][j].heads = grammar[getKey([words[j-1]])];
        table[j-1][j].word = words[j-1];
```

Dopo aver completato le prime due caselle sulla diagonale della matrice, nei successivi due cicli innestati vengono considerate le 'heads' presenti nelle caselle [j-2][i+1] : per ogni coppia di elementi presente in questi vettori si va a guardare nella grammatica se esiste una regola che abbia come corpo la coppia in questione. In caso affermativo la casella considerata viene aggiornata con le informazioni relative alla testa della regola, al corpo, e vengono aggiunti i riferimenti alle caselle che hanno prodotto la regola (leftTree e rightTree) per ricostruire l'albero di derivazione una volta arrivati alla fine dell'algoritmo.

```
for(let i = j-2; i >= 0; i--){
    for(let k = i+1; k < j; k++){

        let b = table[i][k].heads;
        let c = table[k][j].heads;

        // per ogni coppia di elementi presenti in b e c
        for(let it_B = 0; it_B < b.length; it_B++){
            for(let it_C = 0; it_C < c.length; it_C++){

                let roots = grammar[getKey([b[it_B],c[it_C]])];

                if(roots !== undefined){
                    // per ogni regola che ha come corpo getKey([b[iterB],c[iterC]])
                    roots.forEach{(r) => {
                        table[i][j].heads.push(r);
                        table[i][j].body.push(getKey([b[it_B],c[it_C]]));
                        table[i][j].leftTree.push(table[i][k]);
                        table[i][j].rightTree.push(table[k][j]);
                    });
                }
            }
        }
    }
}
```

Infine, se la casella [0][numWords] contiene tra le *'heads'* il simbolo S (radice della grammatica), allora la frase in input è sintatticamente corretta, e viene ritornato l'albero ricostruito tramite backtracking andando ricorsivamente a considerare le caselle leftTree e rightTree, riferimenti delle produzioni che hanno prodotto il risultato.

```
 if(table[0][numWords].heads.includes("S")){
        return new Tree(getTreeFromTable(table[0][numWords], "S"));
    }
    return "ATTENZIONE: Questa frase non è sintatticamente corretta";
}
```

La funzione ricorsiva `getTreeFromTable(node, production)` ricerca all'interno della lista delle 'heads' l'indice che identifica la produzione corretta, crea un nodo con quella produzione e, se la lista *'body'* non è vuota, ricorsivamente va a ricostruire l'albero sul ramo di sinistra a su quello di destra.
Se invece la lista *'body'* è vuota, allora siamo di fronte ad una *lexical rule* e viene assegnato al campo *'word'* la parola di riferimento.

```
1.  function getTreeFromTable(node, production) {
2.      let index = node.heads.findIndex(i => i === production);
3.      let root = new Node(node.heads[index]);
4.
5.      if(node.body.length !== 0) {
6.          let leftRight = JSON.parse(node.body[index]);
7.
8.          if (leftRight[0]) {
9.              root.left = getTreeFromTable(node.leftTree[index], leftRight[0]);
10.         }
11.         if (leftRight[1]) {
12.             root.right = getTreeFromTable(node.rightTree[index], leftRight[1]);
13.         }
14.     }
15.     if(node.word){
16.         root.leaf = node.word;
17.     }
18.     return root;
19. }
```

DA ITALIANO A ITALIANO-YODA

L'ultima parte dell'esercizio trasforma l'albero di derivazione ottenuto al passo precedente in un albero Italiano-Yoda. Per fare ciò, vengono applicate due regole:

1. SVX (forma standard dell'Italiano) -> SXV
2. SXV -> XSV (forma standard dell'Italiano-Yoda)

```
3.  function svx_to_sxv(t) {
4.      let vp = t.getRootNode().right;
5.      let tmp = vp.left;
6.      vp.left = vp.right;
7.      vp.right = tmp;
8.
9.      return t;
10. }
11.
12. function sxv_to_xsv(t) {
13.     let vp = t.getRootNode().right;
14.     let tmp = vp.left;
15.
16.     vp.left = t.getRootNode().left;
17.     t.getRootNode().left = tmp;
18.     return t;
19. }
```

La prima funzione va a prendere il nodo VP, sempre a destra del nodo radice, e scambia i suoi due figli. La seconda funzione invece scambia il figlio sinistro di VP con in nodo NP (sempre a sinistra

del nodo radice). Il risultato è una frase della forma XSV, ovvero Resto_della_frase – Soggetto – Verbo.