Python Implementation of a Polynomial-Runtime Algorithm for CCG

Mildly Context Sensitive Languages:

By 1985, it became a consensus among linguists, that the context-free class of languages, as described by Chomsky (1963) is too restrictive to be able to describe natural languages.

An example for this, discussed by Bresnan et al. (1982), showed the Dutch cross-serial clause construction exceeds the capabilities of context-freeness, however it was argued by some to still maintain the capabilities of weak context freeness.

A formal proof that natural languages are not of the weak context free class came from Shieber (1985) using data collected from native Swiss-German speakers. In his proof, Shieber presented an argument referring to a particular construction of Swiss-German, the cross-serial subordinate clause, and provided evidence that these clauses do indeed cross the context-free barrier.

An example for a language with cross serial dependencies is the copy language –

$L = \{\omega\omega | \omega \in \{0,1\}\}$.

The term "Mildly Context Sensitive" was coined by Joshi (1985) in an attempt to pinpoint the exact formal power required to adequately describe the syntax of natural languages.

The term is meant to account for languages, such as natural languages, that exceed the restrictions on context free languages. However, the context sensitive languages, which are the next step in the Chomsky hierarchy, appeared to be too broad for them.

The main difference between mildly context sensitive languages and context free languages is that the mildly context sensitive grammar can support cross-serial dependencies, as well as all the other properties of context-free grammar.

Over the years, several grammar formalisms that have been introduced in the literature satisfy the properties of the mildly context sensitive grammar, and are able to parse sentences in natural languages.

The first formalism – Tree adjoining grammar (TAG) was suggested by Joshi (1985). Though later, Joshi discovered with his students Vijay-Shanker and David Weir, that TAGs are equivalent to Head Grammars (HG), which was a formalism previously introduced by Pollard (1984). Subseuently, two more formalisms were suggested: Linear Indexed Grammars (LIG) by Weir and Joshi (1988) and Combinatory Categorial Grammar (CCG) by Steedman (1985, 2000).

In the following chapters, I will introduce my own Python implementation for a polynomial-runtime algorithm, developed by Kuhlmann and Satta (2014), for CCG.

CCG – Introduction

Combinatory Categorial Grammar (CCG) is a grammar formalism, first introduced by Steedman. It provides an interface between the syntax and underlying semantics, in which the application of syntactic rules is entirely based on the syntactic category.

The syntactic categories are either primitive categories such as *N, S*, or complex categories such as *S\N, N/N*.

Primitive categories may represent nouns, phrases, or sentences. While complex categories, such as verbs or adjectives, are comprised of categories that represent the type of their result. Some examples being, *VP* or *AP* and categories that represent their arguments or complements. These categories are represented as functions *X/Y* and *X\Y* with an argument *Y* and a result *X*. The slash distinguishes between arguments to the left of the functor (indicated by the forward slash /) and arguments to the right of the functor (indicated by the backward slash \).

An example for the function representing of the verb "proved" from Steedman and Baldridge (2011) – proved := *(S\NP)/NP*

This syntactic category identifies the transitive verb "proved" as a function that takes an *NP* argument from the right – proved What? This makes a *VP*. It also takes an *NP* argument from the left – Who proved what. This results in a grammatical sentence.

CCG defines several combinatory rules:

(1) a. Forward application $X/Y \;\; Y \rightarrow_{>} X$
    b. Backward application $X\backslash Y \;\; Y \rightarrow_{<} X$
(2) a. Forward composition $X/Y \;\; Y/Z \rightarrow_{>B} X/Z$
    b. Forward crossing composition $X/Y \;\; Y\backslash Z \rightarrow_{>B} X\backslash Z$
    c. Backward composition $Y\backslash Z \;\; X\backslash Y \rightarrow_{<B} X\backslash Z$
    d. Backward crossing composition $Y/Z \;\; X\backslash Y \rightarrow_{<B} X/Z$
(3) a. Forward type-raising $X \rightarrow_{>T} T/(T\backslash X)$
    b. Backward type-raising $X \rightarrow_{<T} T\backslash(T/X)$
(4) conjunction $\alpha \;\; \alpha\backslash\alpha/\alpha \;\; \alpha \rightarrow_{<\Phi>} \alpha$
    * where T\X and T/X are parametrically licensed categories by the lexicon

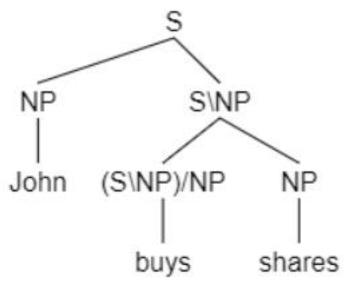
English examples for CCG from Hockenmaier (2003):

A string α is considered grammatical if each word in the string can be assigned a category defined by the lexicon, so that the lexical categories in α can be combined by grammar combinators to form a constituent. This is called a derivation, and it is represented in the following manner:

```
John      buys        shares
______   _________   _____
 NP      (S\NP)/NP     NP
         ___________________>
```

```
                    S\NP
_____________________________<
                  S
```

This constituent structure corresponds to a tree:



Other example:

```
John       bought      and     Mary      ate       potato
_______   _________   _____   _______   _______   _______
  N         S\N/N      conj      N       S\N/N       N
_______>T                     _______>T
S/(S\N)                       S/(S\N)
___________________>B         __________________>B
       S/N                           S/N
_______________________________________________<Φ>
                        S/N
__________________________________________________________
                             S
```

However, for sentences such as "John will offer and may sell Books to Bob", we require a higher degree of composition combinator:

$X/Y \;\; Y/Z/W \;\rightarrow_{>B^2}\; X/Z/W$

To create the derivation:

```
John     will          offer      and     may           sell       books    to Bob
______   __________    ________   ______  __________    ________   ______   ______
  N      (S\N)/(S\N)   S\N/P/N    conj    (S\N)/(S\N)   S\N/P/N     N        P
         _____________________>B²         _____________________>B²
               (S\N)/P/N                        (S\N)/P/N
         ______________________________________________________<Φ>
                             (S\N)/P/N
         ________________________________________________________________>
                                  S\N/P
         __________________________________________________________________________>
                                     S\N
___________________________________________________________________________________<
                                      S
```

Therefore we will use the *"$ convention*" defined by Steedman (2000) to create a general composition rule with degree n: B^n to combine larger complex categories.

The $ convention:

For a category α, $\alpha\$$ denotes the set containing α and all functions (leftward and rightward) into a category in $\alpha\$$

(5) a. Generalized forward composition $X/Y \ (Y/Z)/\$ \rightarrow_{>B^n} (X/Z)/\$$
b. Generalized forward crossing composition $X/Y \ (Y\backslash Z)\$ \rightarrow_{>B^n} (X\backslash Z)\$$
c. Generalized backward composition $(Y\backslash Z)\backslash\$ \ X\backslash Y \rightarrow_{<B^n} (X\backslash Z)\backslash\$$
d. Generalized backward crossing composition $(Y/Z)\$ \ X\backslash Y \rightarrow_{<B^n} (X/Z)\$$

The combinatory rules detailed above are capable of recognizing a sentence in a mildly context sensitive grammar. Following is an example for parsing a string with cross-serial dependencies in CCG from Vijay-Shanker and Weir (1993):

Example:

The following CCG generates $\{\omega c \omega \mid \omega \in \{a, b\}^+\}$.

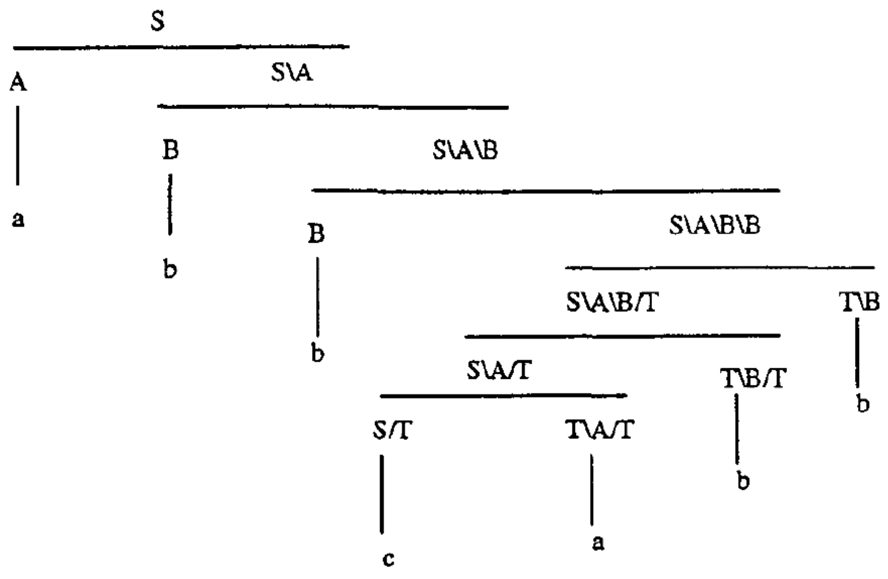
Let G :=

$a := A,\ T\backslash A/T,\ T\backslash A$

$b := B,\ T\backslash B/T,\ T\backslash B$

$c := C,\ S/T$

The string α to be parsed is: $abbcabb$ and the image below from the paper, shows that it is sussecfuly parsed with CCG to a sentence (with the category S):



Exponential Time Algorithm

There have been several suggestions in the literature for exponential running time algoritm for CCG, including that of Shieber et al. (1995).

Shieber at al. have suggested a CKY-style algorithm that represented grammar as a deduction system:

CCG consists of two parts: A lexicon that maps words to sets of basic categories and rules for combining categories into other categories.

Categories are defined as follows:

- Basic categories (or lexicon entries) are categories.
- If c1 and c2 are categories, then (c1/c2) and (c1\c2) are categories.

Categories can be combined by two classes of rules: Application rules and composition rules (discussed in the previous section). Each class of rules has two directions, determined by the slash feature in the category: Forward: / and backward: \.

Deductive Parsing:

We are given a CCG and a string $w = w_0 \cdots w_{n-1}$ to be parsed, where each w_i is a lexical token. The algorithm uses a logic with items of the form *[X, i, j]* where *X* is a category and *i,j* are index positions in *w*, such that the substring from w_i to w_j can be reduced to the category *X*. The goal of the algorithm is the construction of the item *[S, 0, n]* which asserts the existence of a derivation tree for the input string. The construction starts with axioms of the form *[X, i, i+1]* where $w_i \coloneqq X$ is a lexicon entry. We can combine both application and composition rules into a single forward inference rule:

$$\frac{[X/Y, i, j] \;\; [Y\beta, j, k]}{[X\beta, i, k]} \quad X/Y\;Y\beta \rightarrow X\beta$$

And a symmetric backward inference rule:

$$\frac{[Y\beta, i, j] \;\; [X\backslash Y, j, k]}{[X\beta, i, k]} \quad Y\beta\;X\backslash Y \rightarrow X\beta$$

Where X, Y are categories and β is a (possibly empty) stack of slashes and categories.

These rules are used recursively beginning with the start symbol S, until exhasting the entire string.

The algorithm uses an amount of time and space exponential with respect to the length of the input string *n*. This is because the rules may be used to grow the arity of primary input categories up to some linear function of *n*, thus producing exponentially many categories.

Polynomial Time Algorithm:

Vijay-Shanker and Weir (1993) made a breakthrough by developing the first polynomial-time algorithm for recognition in CCG. Their algorithm, which also builds on the notion of CKY, is based on a decomposition of CCG derivations into smaller pieces.

They explained that despite the need for mildly context sensitive grammars to contain complete encodings of unbounded stacks to control the derivations, which should result in an exponential time algorithm as described above; In CCG, the use of stacks is limited since different branches of derivation cannot share stacks. This enables the use of the context-free property used in the CKY algorithm.

Vijay-Shanker and Weir proposed a method to extend the CKY algorithm to handle the limited use of stacks in CCG.

Their proposition is based on the use of nine inference rules, each with a forward and a backward application, that account for all different branches of the derivation of a string. The basic notion is that for every step only a single rule will apply, and rules are implemented recursively so that the final step would propegate back to the first rule, determining if the entire string was recognized by the grammar.
In their paper, Vijay-Shanker and Weir provided a proof of correctness for the algorithm, as well as a proof of runtime complexity of $O(n^7)$ with regard to the length *n* of the input string.

Another polynomial algorithm for CCG was proposed by Kulmann and Satta (2014). They built upon the idea of the Vijay-Shanker and Weir algorithm with several minor changes resulting in a simpler version of the algorithm that includes only three inference rules that are able to account for all the different steps in the derivation.

Their algorithm is easier to prove correct, and is proved to be of a runtime complexity of $O(n^6)$, a slight improvement over the previous algorithm.

Both the Vijay-Shanker and Weir, and The Kulmann and Sata algorithms described in the papers, account only for the application and composition combinators in CCG. However, additional rules can easily be added to support type-raising and conjunction.

The algorithm that Kulmann and Sata presented in their paper, is comprised of similar concepts to that of the exponential time algorithm mentioned above:

We are given a CCG lexicon and a string $w = w_0 \cdots w_{n-1}$ to be parsed, where items are of the form *[X,i,j]*. The goal is the same: Construct the item *[S, 0, n]* to determine if the string *w* is a grammatical sentence in the CCG.

The difference is that to avoid exponential runtime, we limit the items to be of categories whose arities are bounded by some constant c. This constant is determined by the lexicon and an explanation for its calculation is detailed in the paper.

If at a certain step in the parsing, we derive a category that is too long, i.e. its arity exceeds the bound c, it cannot be parsed any longer. Therefore, the new algorithm introduces a new type of item to implement in a specific decomposition of long derivations into smaller pieces, these items are called derivation contexts:

Suppose a derivation *t* yields the *w[i,j]* part of the string with the category *X/Y* (an item of the form *[X/Y, i, j]*). Suppose this derivation has the special property that no combinatory rules that it uses pop the argument *X* from the stack, therefore, after popping */Y*, it may push new arguments and pop them again, but never touch *X*. This special proprty provides a derivation context, and it is useful because it can be carried out with any choice of X.

The paper refers to the excess derivation as β, the (possibly empty) sequence of arguments being pushed to the arguments stack of *X* in place of */Y*, the bridging argument. If we replace *t* by another derivation *t'* with the same yield but with a category type *X'/Y*, where *X≠X'*, since we are not able to touch *X'*, we obtain another valid derivation tree with the same yield as t, and the type of the tree will be *X'*β. The internal structure of *t'* is not important to record, the only important information is the extent of the yield of *t*, specified in terms of *i* and *j*, the extent of the yield of *t'*, specified in the terms of *i'* and *j'*, the identity of the bridging argument */Y* and the excess β.

We represent these pieces of information in a derivation context item of the form –

[/Y, β, i, i', j', j]. We have another type of item [\Y, β, i, i', j', j] with a backward slash and a similar meaning.

The paper specifies that both the original derivation items and the derivation context items are arity-restricted by the constant c. Therefore, the arity of Yβ must not be larger than c.

To maintain both derivation items, we must use additional inference rules to the rule represented in the exponential algorithm.

The new algorithm uses three inference rules, each with a forward and a backward application:

Rule 1:

This rule corresponds to the rules seen in the exponential algorithm; however, it only applies if the arity of the category in the product is small enough. The forward application is as follows:

$$\frac{[X/Y, i, j] \;\; [Y\beta, j, k]}{[X\beta, i, k]} \begin{cases} X/Y \;\; Y\beta \;\to X\beta \\ \;\; ar(X\beta) \leq c \end{cases}$$

Another similar version exists for each rule with the backward application, these will be shown later in the implementation chapter.

Rule 2:

This rule has the same antecedents as rule 1, but applies when the arity bound is exceeded, which would result in an impossible derivation with the use of rule 1. Rule 2 then triggers a new derivation context [/Y,β, i, i, j, k]:

$$\frac{[X/Y,i,j]\ \ [Y\beta,j,k]}{[/Y,\beta,i,i,j,k]} \begin{cases} X/Y\ \ Y\beta\ \rightarrow X\beta \\ ar(X\beta) > c \end{cases}$$

Further applications described below, extend the new context into new derivations.

Rule 3:

When a derivation context excess has become sufficiently small, it will be recombined with the derivation that triggered it. This is done by rule 3:

$$\frac{[X/Y,i',j']\ \ [/Y,\beta,i,i',j',j]}{[X\beta,i,j]} \ \{ar(X\beta) \leq c$$

Another set of 3 rules must be set to be able to extend derivation context further, until the excess is small enough to be put together with the original derivations. These rules correspond directly to the three rules showed above, but take derivation contexts as their antecedents, instead of normal derivation trees:

Rule 4:

This rule is parallel to rule 1 and it extends a derivation in the same manner:

$$\frac{[/Y,\beta/Z,i,i',j',j]\ \ [Z\gamma,j,k]}{[/Y,\beta\gamma,i,i',j',k]} \begin{cases} X/Z\ \ Z\gamma\ \rightarrow X\gamma \\ ar(Y\beta\gamma) \leq c \end{cases}$$

Rule 5:

This rule is parallel to rule 2, it triggers a new context derivation if the antecedent context derivation exceeds the arity bound, and thus cannot be further extended:

$$\frac{[/Y,\beta/Z,i,i',j',j]\ \ [Z\gamma,j,k]}{[/Z,\gamma,i,i,j,k]} \begin{cases} X/Z\ \ Z\gamma\ \rightarrow X\gamma \\ ar(Y\beta\gamma) > c \end{cases}$$

Rule 6:

This rule is parallel to rule 3, it is used to recombine a context derivation with the context that originally triggered it:

$$\frac{[/_1Y,\beta/_2Z,i'',i',j',j'']\ \ [/_2Z,\gamma,i,i'',j'',j]}{[/_1Y,\beta\gamma,i,i',j',j]}$$

Kuhlmann and Sata present a thorough proof of correctness and completeness of the algorithm in their paper, as well as a runtime analysis which concludes that the algorithm runs in time of $O(n^6)$ in respect to the length of the input string.

They have specified that although the algorithm is only meant for recognition in CCG, standard techniques of using back-pointers at each inference rule can be applied to obtain a derivation forest from the parsing table.

Python Implementation

I decided to implement the Kuhlmann and Sata algorithm in Python.

I adapted the inference rules presented in the paper into a designated class containing all six rules. Each of the rules has a forward and a backward application.

The algorithm receives a lexicon and a string to be parsed and translates the string into the lexicon categories.

A segment with a category that contains the "S" symbol is determined to begin the parsing process, and rule 1 is applied.

The rules are made to work recursively. Each rule examines the segment in turn and determines:

- The degree of the combinator used
- The direction of the slash
- The product of the derivation

Rules such as rule 1 and rule 4 assert that the category created by the derivation is within the constant bound. If it is, the rules are recursive and will work until the completion of the string, if the arity exceeds the bound, rules 2 and 5 will be called respectively to create a new derivation context. Then rules 3 and 6 will be used respectively to go back to the original derivation when the arity is sufficiently small.

If the parsing process was successful and the algorithm was able to achieve the derivation

[S, 0, n], it will return True.

Parsing examples:

1)

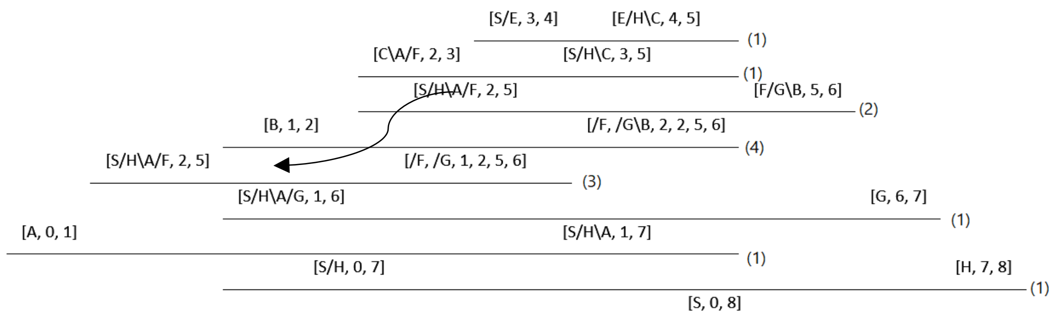
The first example is similar to a sample derivation, taken from Kuhlmann and Sata (2014), and it shows a very simple parse derivation with a toy grammar, that only operate rules 1-4:

Lexicon:

$w_0 \coloneqq A$ $w_4 \coloneqq E/H\backslash C$
$w_1 \coloneqq B$ $w_5 \coloneqq F/G\backslash B$
$w_2 \coloneqq C\backslash A/F$ $w_6 \coloneqq G$
$w_3 \coloneqq S/E$ $w_7 \coloneqq H$
The start symbol is S, The constant $c = 3$ and the input string: $w_0 \cdots w_7$

We start with the item *[S/E, 3, 4]* and rule 1 applies twice (once forward and once backward) to obtain *[S/H\A/F, 2, 5].* Combining this item with *[F/G\B, 5, 6]* is not possible using rule 1, because this would result in a category with arity 4, which exceeds the bound. Therefore, we use rule 2 to trigger the context item *[/F, /G\B, 2, 2, 5, 6].* We use rule 4 to obtain the item *[/F, /G, 1, 2, 5, 6]*, and at this point we use rule 3 to recombine the context item with the

tree item that originally triggered it; this yields the item *[S/H\A/G, 1, 6]*. This item's category has a sufficiently small arity.
Rule 1 now applies three times, to finally produce the goal item *[S, 0, 8]*.



The second example is an extension of the previous example, meant to show a parsing which would require rules 5 and 6 to apply as well:

Lexicon:

$w_0 \coloneqq A$	$w_5 \coloneqq B/C\backslash D$	$w_9 \coloneqq K/L$
$w_1 \coloneqq H$	$w_6 \coloneqq E/F/G$	$w_{10} \coloneqq L$
$w_2 \coloneqq J$	$w_7 \coloneqq G\backslash H/I$	$w_{11} \coloneqq F$
$w_3 \coloneqq D/E$	$w_8 \coloneqq I\backslash J/K$	$w_{12} \coloneqq C$
$w_4 \coloneqq S\backslash A/B$		

The start symbol is S, the constant $c = 3$ and the input string: $w_0 \cdots w_{12}$

This example is very much the same as the previous one and will not be thoroughly presented because it is a rather long process.

The main difference to the first example is that after applying rule 2 to trigger the derivation context *[/E, /F/G, 3, 3, 6, 7]* rule 4 is applied once to create *[/E, /F\H/I, 3, 3, 6, 8]*. However, instead of the excess arity becoming smaller, it becomes larger, so we are not able to use rule 3 to return to the original derivation.

We must continue with rule 4, except this time, if we apply use rule 4 to combine the context item with *[I\J/K, 8, 9]*, we will receive a derivation context item that also exceeds the arity bound; so we must apply rule 5 instead to create a new derivation context –

[/I, \J/k, 3, 3, 8, 9].

We proceed in the same manner, until the excess arity becomes sufficiently small to be combined with the original via rule 6, and later on combine with the derivation tree item via rule 3. Then we continue recursively until the completion of the string and the production of the goal item *[S, 0, 13]*.

2)

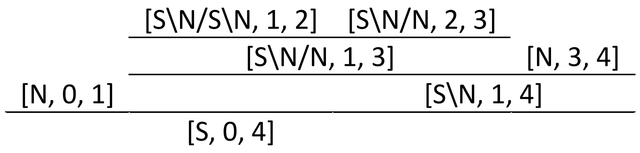
An example for an English sentence, also taken from Kulmann and Sata (2014). This example shows derivations with a higher degree:

Lexicon:

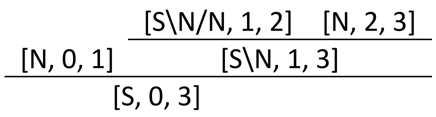
$Lousie := N$ $\qquad$ $marry := S\backslash N/N$

$might := S\backslash N/S\backslash N$ $\qquad$ $Harry := N$

The start symbol S, the constant $c = 5$ and the input string: $Louise\ might\ marry\ Harry$



Notice that there is another string that would derive a grammatical sentence in this lexicon. That is the string: *Louise marry Harry*



This makes some sense, as the lexicon does not seem to adhere to rules of tense and agreement in English. This could than have a similar meaning to "Louise married Harry" or "Louise is marrying Harry".

Therefore, it is not surprising that for the string: $Louise\ might\ Harry$, there is no possible derivation that would result in a grammatical sentence.

Implementation details

The attached Python implemntation for the polynomial-runtime algorithm developed by Kuhlmann and Sata (2014), demonstrates the algorithm's ability to parse sentences in mildly context-sensitive grammar with CCG formalisms: application and compositions of singular and multiple degress. This is done in a polynomial runtime complexity, with respect to the length of the input string.

The script demonstrates the examples above, and can be further enhabced to include additional rules for supporting more advanced features of mildly-context sensitive languages, such as type-raising. New lexicon examples can be added and examined to allow parsing of different sentences in English and in other languages.

The script is submitted in three Python files:

- ccg_algorithm.py – The main module which calls the algorithm. This is the execution file and it demonstrates the parsing of the strings in the examples above.
- Rules.py – The main algorithm implementation.
- Lexicon.py – Example lexicons for the script.

References

Bresnan, J., Kaplan, R. M., Peters, S., & Zaenen, A. (1982). Cross-serial dependencies in Dutch. In *The formal complexity of natural language* (pp. 286-319). Springer, Dordrecht.

Chomsky, N. (1963). Formal properties of grammars. *Handbook of Math. Psychology*, *2*, 328-418.

Hockenmaier, J. (2003). Data and models for statistical parsing with Combinatory Categorial Grammar.

Joshi, A. K. (1985). Tree adjoining grammars: How much context-sensitivity is required to provide reasonable structural descriptions?.

Kuhlmann, M., & Satta, G. (2014). A new parsing algorithm for Combinatory Categorial Grammar. *Transactions of the Association for Computational Linguistics*, *2*, 405-418.

Pollard, C. J. (1984). Generalized phrase structure grammars, head grammars, and natural language. *Ph. D. disertation, Stanford University*.

Shieber, S. M. (1985). Evidence against the context-freeness of natural language. In *Philosophy, Language, and Artificial Intelligence* (pp. 79-89). Springer, Dordrecht.

Shieber, S. M., Schabes, Y., & Pereira, F. C. (1995). Principles and implementation of deductive parsing. *The Journal of logic programming*, *24*(1-2), 3-36.

Steedman, M. (1985). Dependency and coördination in the grammar of Dutch and English. *Language*, 523-568.

Steedman, M. (2000). *The syntactic process* (Vol. 24). Cambridge, MA: MIT press.

Steedman, M., & Baldridge, J. (2011). Combinatory categorial grammar. *Non-Transformational Syntax: Formal and explicit models of grammar*, 181-224.

Vijay-Shanker, K., & Weir, D. (1993). Parsing some constrained grammar formalisms. *Computational Linguistics*, *19*(4), 591-636.

Weir, D., & Joshi, A. (1988, June). Combinatory categorial grammars: Generative power and relationship to linear context-free rewriting systems. In *26th Annual Meeting of the Association for Computational Linguistics* (pp. 278-285).