MDL Learner of CCG Grammars for Mildly Context Sensitive Languages

Mildly Context Sensitive Languages

The Chomsky Hierarchy, described in Chomsky (1956) introduces a containment hierarchy of classes of formal grammars:

Regular $\subsetneq$ Context-Free $\subsetneq$ Context-Sensitive $\subsetneq$ Recursively Enumerable

While several theories attempted to compare the natural languages to the context-free class of languages; by 1985, it became a consensus among linguists, that context-free grammar is too restrictive to be able to describe natural languages.

A formal proof of that 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 –

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

Another property of natural languages that exceeds the bounds of context-freeness is the constant growth property: every word in the language is longer than the next shorter word by a specifically defined constant at most.

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.

He described the set of mildly context sensitive languages as having the following properties:

1) Support cross-serial dependencies
2) Constant growth
3) Can be parsed in deterministic polynomial time

As well as all the other properties of the context-free grammar.

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.

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.

One formalism in particular is the Combinatory Categorial Grammar (CCG) defined by Steedman (1985, 2000), and will be discussed further in the next chapter.

In a previous work, Etzion (2020), I have created a Python implementation of a Polynomial time parser for CCG, based on the algorithm developed by Kuhlmann and Satta (2014).

In this work I have attempted to create an MDL learner for mildly context sensitive languages, using my implementation of the CCG parser.

In the following chapters, I will introduce the formalism CCG for mildly context sensitive grammars, as well as my own Python implementation for a polynomial-runtime parser, for CCG. Afterward I will discuss learning and the methods I selected for implementing a CCG learner, based on the CCG parser.

CCG

Combinatory Categorial Grammar (CCG) is a grammar formalism, first introduced by Ades and Steedman (1982). 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 $X\backslash Y \ \ Y \rightarrow_{<} X$

The application rules state that if a constituent with a category X/Y is immediately followed by a constituent Y (or analogously, if X\Y is immediately followed by Y), They can be combined to form a constituent with category 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 $Y/Z \ \ X\backslash Y \ \rightarrow_{<B} \ X/Z$

The composition rules allow two functor categories to combine to form another functor.

(3) a. Forward type-raising $X \rightarrow_{>T} \ T/(T\backslash X)$
b. Backward type-raising $X \rightarrow_{<T} \ T\backslash(T/X)$

Type-raising rules allow an argument category X to change into a functor category T/(T\X) or T\(T/X) where T\X and T/X are parametrically licensed categories by the lexicon.

(4) conjunction $\alpha \ \ \alpha\backslash\alpha/\alpha \ \ \alpha \ \rightarrow_{<\Phi>} \ \alpha$

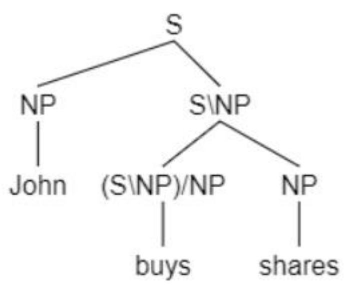
The conjunction rule allows to conjunct two identical functors represented as α if they are separated by the conjunction "and".

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 enable the analysis of long-range dependencies in natural grammars, that involve extraction and coordination constructions.

Through the use of these rules, it was demonstrated in Vijay-Shanker and Weir (1994) that CCG is equivalent to the family of mildly context sensitive languages and abide by their properties.

CCG Polynomial Parser

Several polynomial-time algorithms for CCG Parsers were introduced in the literature. The first breakthrough in this matter was the algorithm developed by Vijay-Shanker and Weir (1993). This algorithm provides a method to extend the CKY algorithm, to support the decomposition of CCG derivations into smaller pieces.

Later, 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.

Both the Vijay-Shanker and Weir, and The Kulmann and Satta 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.

In my previous work, I decided to implement the Kuhlmann and Satta algorithm in Python and have created an implementation of a CCG parser based on the rules presented in their paper.

I adapted the inference rules presented in the paper into a designated class containing 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.

For further explanations and examples about the algorithm and its implementation in Python, See Etzion (2020).

Learning

After developing a parser that is able to recognize grammatical sentences in a mildly context sensitive grammar; the next question we should ask is whether we can develop a program that is able to recognize the correct grammar, after receiving enough correct utterances.

This is the question of learning and it is dates far back to the Classical Period. Phillips and Soltis (2015) explain the origins of this question going back as far as the writings of Plato, the question he is posing is:

> *"How is that a learner is able to understand something new? Consider a person who was absolutely and completely ignorant; how could this person understand, and learn, something that was totally incomprehensible? (Could a computer, with completely empty data banks and no internal program acquire a piece of factual information without any proper preparation?)"*

The answer presented in the book is that learning depends upon the learner having some prior knowledge or experience. Several theories as to what this "prior knowledge" is or where it comes from have been presented over the years.

For this work, I will rely on a few commonly accepted assumptions:

- The existence of Universal Grammar (UG) – A concept that have been suggested by Chomsky and Halle (1965) and in this case, represents an initial grammar the child biologically inherits, and uses to create a specific grammar, after being exposed to linguistic environment.
- Lack of negative evidence – It has been reported in the literature that children have no access to systematic negative evidence while learning a natural language (Marcus 1993). Therefore, the learning process may only rely on correct grammatical utterances as systematic input.

According to these assumptions, a program that can emulate a close resemblance to the human leaner must be provided with the same resources – an initial grammar hypothesis and a stream of correct grammatical sentences.

Avraham (2017) has stated in his work on head-complement order learner that to implement a syntactic grammar, several models are required:

- Formalism – a description of the grammar.
- Parser – an algorithm that tests whether a given grammar generates the input sentence.
- Metric – a method to compare different grammars in order to choose the "best" one.
- Learning algorithm.

As discussed earlier, I have chosen the formalism of CCG that can represent a mildly context sensitive grammar and have already created a parser that is able to recognize sentences given a CCG grammar.

The next step, according to Avraham, is to select a metric for comparing the grammars. This metric is the Minimum Description Length.

Minimum Description Length (MDL)

Generating some grammar for a given data is a fairly easy task, we may begin by constructing the simplest grammar that accounts for all possible inputs. However, as we continue to receive more sentences that are different from each other, we would have to enlarge our lexicon considerably to account for each. That is an ineffective method if there are many generalizations that could be made, regarding the input.

We would then like a method of grading a specific grammar, to determine if it is a "good" grammar, and by means of this method, to eventually settle on the "best" grammar for the input.

Consider the following example of data discussed in Katzir (2014):

```
pabikugolatudaropitibudogolatutibudogolatugo
latupabikutibudodaropitibudopabikugolatudaro
pipabikutibudogolatu
```

This data is based on an experiment that was presented in Saffran et al. (1996) to examine generalization in 8-month-old babies.

An observation on the data:

It is apparent that this sequence is comprised of 4 recurring strings: pabiku, golatu, daropi and tibudo. This seems to be the most important observation.

However, we can make several other observations:

- The strings kugo, pa and pabikugolatu also seem to repeat in the text.
- The string pabikudarpi never appears
  And more.

These do not seem to be quite as important as the first observation. If we were to attempt to decide on the "best" grammar to account for the data, we would want a grammar that takes into account the first observation, and refers to pabiku, golatu, daropi and tibudo as lexicon items. But we would probably not want to account for the latter observations in the grammar, which will create an overly complex rule system.

The methodology I adopted is the Minimum Description Length (MDL), demonstrated by Katzir (2014). It follows the work of Rissanen (1978) and offers a method to grade a certain grammar based on its complexity and its efficiency regarding a given input.

MDL is the sum of the length two elements:

- G – The grammar itself. How economical is our hypothesis?
- D:G – The encoding of the given data, using the grammar

The goal would be to discover the grammar where this sum: |G|+|D:G| is minimal. That would be considered the best grammar for the given data.

Returning to our previous example, we will consider several possible grammars for the data given above.

1) $S \rightarrow W\, S$
   $W \rightarrow a \mid b \mid d \mid g \mid i \mid k \mid l \mid o \mid p \mid r \mid t \mid u$

   This is a simple grammar that could generate the data above, as well as many other different sequences. However, it might be too simplistic as it does not rely on the lexical generalization we were able to make, and so the lexicon available is rather long and comprised of 12 different segments. Assuming we use binary digits to encode our grammar, we need $\lceil \log 12 \rceil$ digits to describe the entire lexicon.
   Therefore |G| $\approx \lceil \log 12 \rceil \times 12$ = 48.
   |D:G| is much longer. To create the exact data given above, we must choose the correct segment out of the 12 segments available in the lexicon, and to do so for the entire 108 segments in the data. Therefore |D:G| $\approx \lceil \log 12 \rceil \times 108 = 432$
   We received the sum MDL = 48 + 432 = 480

2) $S \rightarrow W\, S$
   $W \rightarrow pabiku \mid golatu \mid daropi \mid tibudo$

   This grammar does capture the generalization we have made earlier. The lexicon has only 4 words and it is smaller, but the words are 6 segment strings, rather than 1 segment strings. Therefore, the encoding is longer than that of the last grammar.
   |G| $\approx \lceil \log 12 \rceil \times 6 \times 4 = 96$
   But here, generating the data from the grammar is easier: we have 4 lexicon items to choose from, and since each is 6 segments long, we must only make this choice for sixth of the data: |D:G| $\approx \lceil \log 4 \rceil \times \frac{108}{6} = 36$
   We received MDL = 96 + 36 = 132
   This is a much better result, which fits with our previous assumptions.

3) $S \rightarrow W\, S$
   $W \rightarrow pabikugolatu \mid daropitibudo \mid golatutibudo \mid$
   $golatugolatu \mid pabikutibudo \mid daropipabiku \mid tibudogolatu$

   In this grammar, generating the data is simpler, we have 7 lexicon strings to choose from but only $\frac{108}{12} = 9$ choices must be made. Therefore |D:G| $\approx \lceil \log 7 \rceil \times 9 = 27$
   However, encoding the grammar is much more complicated, with 7 strings of 12 segments each: |G| $\approx \lceil \log 12 \rceil \times 12 \times 7 = 336$
   We received MDL = 336 + 27 = 363
   And this is not a good outcome compared to the former.
   We can conclude that this grammar is too complex given this data and choose the second grammar suggested above, as the "best" grammar out of the three.

Simulated Annealing Algorithm

The final model we are required to implement is an algorithm that can determine the best grammar for a given input in an efficient way.

I followed the search procedure used by Katzir (2014) and Rasin and Katzir (2016), the strategy of the Simulated Annealing Algorithm (Kirkpatrick et al. 1983).

Simulated Annealing is a probabilistic technique for locating the global optimum of a given function in a large search space.

Its name stems from the annealing in metallurgy, a process that involves heating and controlled cooling of a material to increase the size of its crystals and reduce their defects.

The process relies on two variables: temperature and energy.

The energy is the function we are attempting to minimize to an optimum state, and the temperature is our probabilistic margin of error.

The general process works as follows: We start with an initial hypothesis; the temperature decreases progressively from an initial positive value to zero. At each iteration, the algorithm randomly selects a neighbor hypothesis, measures its energy to determine if it is an improvement over the latter, and moves to the next hypothesis according to the temperature-dependent probabilities of selecting a better or worse hypothesis.

When the temperature reaches zero, an optimum has been achieved.

In the case of language learning with MDL, we would use a simulated annealing algorithm that generates an initial hypothesis of a possible grammar for an input. The learner will generate a neighbor grammar at each iteration, that is a different but closely related grammar of the first one, that is also able to generate the input.

The Energy would be calculated through MDL. If the MDL of the new hypothesis is smaller, it is considered a more efficient grammar and would be selected. Otherwise it could be selected with a probability depending on the current temperature and on the difference between the MDL score of the current hypothesis. The temperature would decrease by a constant value in each iteration.

Once the temperature is low enough, an optimal grammar would be chosen for the input.

A pseudocode of the search procedure presented by Rasin and Katzir (2016):

```
$D \leftarrow \textit{input string in } \Sigma$
$G \leftarrow \textit{initial\_grammar}(\Sigma)$
$T \leftarrow \textit{initial temperature}$
**while** $T > \textit{threshold}$ **do**
    $G' \leftarrow \textit{random\_neighbor}(G)$
    $\Delta \leftarrow [|G'| + |D{:}G'|] - [|G| + |D{:}G|]$
    **if** $\Delta < 0$ **then**
        $p \leftarrow 1$
    **else**
        $p \leftarrow e^{-\frac{\Delta}{T}}$
    **end if**
    choose $G \leftarrow G'$ with probability $p$
    $T \leftarrow \alpha T$
**end while**
**return** $G$
```

Implementation Details:

I have followed the simulated annealing algorithm presented above and adapted it for CCG grammars that are compatible with my CCG parser.

The Algorithm receives an input string and an initial grammar and in each iteration the algorithm generates a random grammar and validates the parsing of the input by this grammar. It then calculates the MDL and compares it to the previous to select the new grammar according to the probability function presented above (in the pseudo code).

Following Rasin and Katzir (2016), I have selected the initial temperature to be 100.0 and the cooing factor α to be 0.999985.

I implemented several methods in 2 Python files:

Simulated algorithm learner:

- The simulated annealing algorithm as described above
- MDL calculation method for calculating MDL of a given grammar in CCG and input string
- A main program that tests the algorithm, with the examples detailed below

Grammars:

- a grammar pool – a collection of CCG grammars compatible with the examples detailed below

Additionally, submitted with this work are 3 more Pthon files:

- ccg_algorithm
- rules
- lexicon

These files are the implementation of the CCG parser I used, that Ihave implemented in my previous work.

Examples:

I have constructed the following example to be tested by the learner:

Consider the string "abaababaab"

To simplify the idea, we can assume that the grammar that generates this string can only generate a's and b's. Even so, there are many possible grammars that can generate it.

Grammar 1:

This grammar stems from the observation that "abaab" repeats itself in the string. The grammar that may generate it is of the copy language: $\{ww \mid w \in \{a, b\}\}$. In CCG we will construct this grammar as follows:

a: A, S\A/S, S\A

b: B, S\B/S, S\B

This could be our initial hypothesis grammar.

Calculating MDL for CCG:

I have decided to approach the matter of calculating MDL in the following manner:

- The size of G is the how many binary digits it would supposedly take to encode the grammar. In this case we have 5 different segments to encode: "A","B","S","/","\". This would require $\lceil \log 5 \rceil = 3$ digits.
  We have 6 different rules:
  - two rules with the length of 1 segment: A, B
  - two rules with the length of 5 segments: S\A/S, S\B/S
  - two rules with the length of 3 segments: S\A, S\B

  Therefore, we conclude $|G| = 3 \times (2 \cdot 1 + 2 \cdot 5 + 2 \cdot 3) = 54$

- The Size of D:G is the choices me must make to achieve the input string from the grammar. We have 6 rules to choose from, which would cost $\lceil \log 6 \rceil = 3$ to encode, and since each rule must be chosen for each letter, the choice must be made 10 times (the size of the input string).
  Therefore, we conclude $|D{:}G| = 3 \times 10 = 30$

We received $MDL = 54 + 30 = 84$

Grammar 2:

We may approach the input string from a different angle, and notice that the string is comprised of these pairs:

ab aa ba ba ab

Without the pair "aa" it would have been a palindrome. An approach to this could be to determine a grammar where the strings "ab" and "ba" are written as a palindrome, and the string "aa" can come anywhere:

ab: A, S/A\A, B\A

ba: B, S/B\S, A\B

aa: A\A, A/A, B\B, B/B

This is a more complicated idea that would seem strange to choose above the first grammar, and calculating MDL show:

- $|G| = \lceil \log 5 \rceil \times (2 \cdot 1 + 2 \cdot 5 + 6 \cdot 3) = 90$
- $|D{:}G| = \lceil \log 10 \rceil \times 5 = 20$

We received MDL = 110

This is much worse. Even though there are half as many choices for |D:G| the encoding of the grammar is too complex to make it a good hypothesis, as shown by the MDL result.

Grammar 3:

Another approach to the string is that it is combined of the words:

aba aba baab

A possible grammar could be one that generates "aba" as many times as we want, and afterwards generates "baab" as many times as we want: $\{(aba)^n (baab)^m |\, n, m \geq 1\}$

aba: A, S\A/B

baab: B, B/B

- $|G| = \lceil \log 5 \rceil \times (2 \cdot 1 + 5 + 3) = 30$
- $|D{:}G| = \lceil \log 4 \rceil \times 3 = 6$

We received MDL = 36

This is a much better result, and it is not surprising, as the grammar is much simpler than the previous one and the construction of the input string is much easier as well.

Grammar 3b:

The last example is derived from grammar 3 we saw above. We might notice that our string has only one appearance of the string "baab". Maybe we can strict the grammar even more, so that "aba" can appear as many times as we want but "baab" must only appear once.

This would be acquired by removing the rule b: B/B from grammar 3, that allows "baab" to repeat. We will then receive the grammar:

aba: A, S\A/B

baab: B

- $|G| = \lceil \log 5 \rceil \times (2 \cdot 1 + 5) = 21$
- $|D{:}G| = \lceil \log 3 \rceil \times 3 = 6$

We received MDL = 27

This is an even better result for the given string.

Conclusion

In this work, I attempted to use my previous implementation of a Python CCG parser and extend it into a simulated annealing learner that is able to use the methodology of MDL in order to decide on the best grammar for a given input string.

I have created an example that demonstrates several possible grammars and have showed that the algorithm does eventually choose the favorable outcome.

This example can be broadened further with many more different grammars that are able to parse the same string.

Other examples can be created as well and should all result in the choice of grammar with the optimal MDL for each string given.

Several ideas on extending the learner further:

- Generating grammars: At this point, the learner does not have means to generate infinite grammars for an input string, rather it can choose randomly from a pool of grammars already given. The next step would be to attempt to create an algorithm that takes an initial grammar and is able to generate a neighbor grammar, that can still parse the string.
- Adding more sentences to the input: I have tested my learner with a single string and demonstrated dividing it to words in different methods by different grammars. The next step would be to add more sentences to the input and see how that affects the grammar being chosen.
- Extending the parser to be able to handle grammars with multiple rules per word: My initial implementation of the parser does not support having a lexicon word with more than one rule. Therefore, for rules such as a: A, S/A, I had to rewrite the grammar as – a1: A, a2: S/A, and I had to make the same adjustment to the input string, so it would comply with the grammar. Extending the parser to contend with this problem, would allow an easier transition from the input string to different grammars in the learner.

References

Ades, A. E., & Steedman, M. J. (1982). On the order of words. *Linguistics and philosophy*, *4*(4), 517-558.

Avraham, T. (2017). Learning head-complement order with Minimalist Grammars. MA thesis, Tel Aviv University

Chomsky, N. (1956). Three models for the description of language. *IRE Transactions on information theory*, *2*(3), 113-124.

Chomsky, N., & Halle, M. (1965). Some controversial questions in phonological theory. *Journal of linguistics*, *1*(2), 97-138.

Etzion, H. (2020) Python Implementation of a Polynomial-Runtime Algorithm for CCG. Parsing Seminar, Tel Aviv University

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?.

Katzir, R. (2014). A cognitively plausible model for grammar induction. *Journal of Language Modelling*, *2*.

Kirkpatrick, S., Gelatt, C. D., & Vecchi, M. P. (1983). Optimization by simulated annealing. *science*, *220*(4598), 671-680.

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

Marcus, G. F. (1993). Negative evidence in language acquisition. *Cognition*, *46*(1), 53-85.

Phillips, D., & Soltis, J. F. (2015). *Perspectives on learning*. Teachers College Press.

Rasin, E., & Katzir, R. (2016). On evaluation metrics in Optimality Theory. *Linguistic Inquiry*, *47*(2), 235-282.

Rissanen, J. (1978). Modeling by shortest data description. *Automatica*, *14*(5), 465-471.

Saffran, J. R., Aslin, R. N., & Newport, E. L. (1996). Statistical learning by 8-month-old infants. *Science*, *274*(5294), 1926-1928.

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

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.

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

Vijay-Shanker, K., & Weir, D. J. (1994). The equivalence of four extensions of context-free grammars. *Mathematical systems theory*, *27*(6), 511-546.