

Article

Not peer-reviewed version

Making a Grammar Checker with Autocorrect Options Using NLP Tools

[Radu Bucea Manea Tonis](#)^{*} and Adrian Beteringhe

Posted Date: 23 August 2023

doi: 10.20944/preprints202308.1640.v1

Keywords: grammar; natural language; logic programming; syntactic analysis




Preprints.org is a free multidiscipline platform providing preprint service that is dedicated to making early versions of research outputs permanently available and citable. Preprints posted at Preprints.org appear in Web of Science, Crossref, Google Scholar, Scilit, Europe PMC.

Copyright: This is an open access article distributed under the Creative Commons Attribution License which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

Article

Making a Grammar Checker with Autocorrect Options Using NLP Tools

Radu Bucea Manea Tonis ^{1,*}, , Adrian Beteringhe ^{2,†}

¹ Danubius University, School of Behavioral and Applied Sciences; radumanea@univ-danubius.ro

² Danubius University, School of Behavioral and Applied Sciences; adrianbeteringhe@univ-danubius.ro

* Correspondence: radumanea@univ-danubius.ro; Tel.: +40-727-888-619

† Current address: 3 Galati Boulevard, Galati 800654, Romania.

‡ These authors contributed equally to this work.

Abstract: Our natural language approach concerns syntactic analysis using a dedicated Javascript library - wink-nlp - and semantic analysis based on Prolog programming language, facilitated by another Javascript library - tau-prolog - that allows defining logical programs, declaring rules and checking for goals inside Javascript language. Firstly, our program splits the original text into sentences, then into tokens and identifies each part of the sentence, dynamically maps entities into Prolog rules, then check the spelling accordingly to the definite clause grammar (DCG) by querying the pre-defined program for initial goals (the sentence itself). Basically, we let the parser infer its own rules from the syntactic point of view, then check the grammar from a semantic perspective against the DCG inside the same workflow or pipeline of steps. The provided article demonstrates the usage of the wink-nlp and tau-prolog packages for natural language processing (NLP) and logical programming, respectively.

Keywords: grammar; natural language; logic programming, syntactic analysis

1. Introduction

Grammar proofreaders fall into two categories, those that perform syntactic analysis of the sentence and ensure the identification of sentence parts in order to establish the correct relationship between them according to a predefined linguistic model, and those that are based on AI, e.g. grammarly, and which can learn step by step the correct structure of a sentence and transform a grammatically wrong sentence into a correct one. For learning, training sets are used, such as C4_200M made and provided by Google and which contains examples of grammatical errors along with their correct form. [1] Syntactic analysis shows the following aspects of the sentence: [2]

- Word order and meaning - syntactic analysis aims to extract the dependence of words with other words in the document. If we change the order of words, then it will be difficult to understand the sentence;
- Retention of stop words - if we remove stop words, then the meaning of a sentence can be changed altogether;
- Word morphology - stemming, lemmatization will bring words to their basic form, thereby changing the grammar of the sentence;
- Parts of speech of words in a sentence - identifying the correct speech part of a word is important.

Identifying entities and their relationships in text is useful for several NLP tasks, for example creating knowledge graphs, summarizing text, answering questions, and correcting possible grammatical mistakes. For this last purpose, we need to analyze the grammatical structure of the sentence, as well as identify the relationships between individual words in a particular context. Individual words that refer to the different topics and objects in a sentence, such as names of places and people, dates of interest, or other the same, are referred to as "entities", see Figure 1:[3]

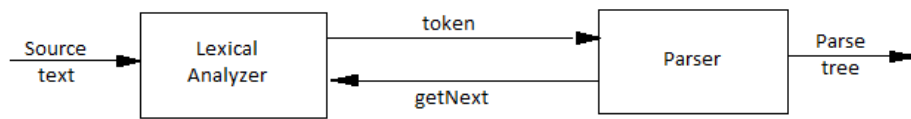


Figure 1. The interaction between lexical analyzer and parser, after [2].

Relationships are established by means of verbs or simple joining, as is the case with collocations. In the case of the latter, bigram trees can be used in the form of linear development on the agglutinative principle or the Fibonacci sequence, resulting in simply chained lists, please see Figure 2:

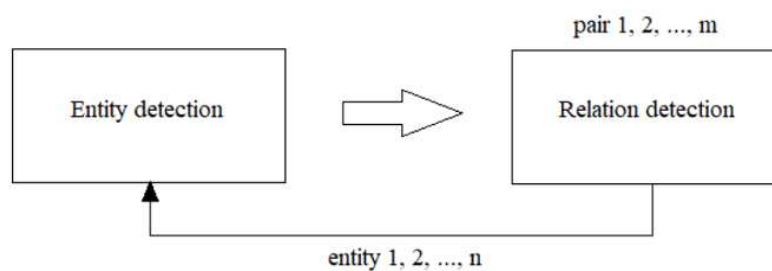


Figure 2. Inference pipeline architecture, after [3].

The main unit of content mapping is the sentence or statement. In the case of natural languages, the sentence structure is SVO in the case of Indo-European languages. Other primitive structures like the agglutinative language of the Minoans highlighted in the Linear A script (partially deciphered) seems to follow a VSO structure and the ancient Germanic languages a curious OSV (pre-Celtic?) order. There are some problematic considerations about rendering sentences in predicate logic [4], but since we address our parser only to simple, straight forward English texts, we hope not to encounter ambiguous situations like the following, where is not clear why every farmer should beat every donkey they own, if Pedro, for instance, beats regularly his donkeys:

$$\forall x[[donkey(x) \& own(Pedro, x)] \rightarrow beat(Pedro, x)] \quad (1)$$

$$\forall x \forall y[[farmer(x) \& donkey(y) \& own(x, y)] \rightarrow beat(x, y)] \quad (2)$$

2. Materials

Factors such as openness, simplicity, flexibility, full browser integration, and attention to the security and privacy concerns that naturally arise in executing untrusted code have helped the Javascript language gain very significant popularity despite its low initial efficiency. Overall, it allows for a disruptive paradigm shift that gradually replaces the development of OS-dependent applications with web applications that can run in a variety of devices, some completely portable.[5] WinkNLP is a JavaScript library for natural language processing (NLP). Specifically designed to make NLP application development easier and faster, winkNLP is optimized for the right balance between performance and accuracy. It is built from the ground up with a weak codebase that has no external dependence. The .readDoc() method, when used with the default instance of winkNLP, splits text into tokens, entities, and sentences. It also determines a number of their properties. They are accessible by the .out() method based on the input parameter — its.property. Some examples of properties are value, stopWordFlag, pos, and lemma, see Table 1:

Table 1. Common its.properties that become available at each stage, after <https://winkjs.org/wink-nlp>.

Stage	Description
tokenization	Splits text into tokens.
sbd	Sentence boundary detection — determines **span** of each sentence in terms of start and end token indexes.
negation	Negation handling — sets the negationFlag for every token whose meaning is negated due a "not" word.
sentiment	Computes sentiment score of each sentence and the entire document.
ner	Named entity recognition — detects all named entities and also determines their type and span.
pos	Performs part-of-speech (pos) tagging.
cer	Custom entity recognition — detects all custom entities and their type and span. The detection is carried out on the basis of training carried out using learnCustomEntities() method.

The .readDoc() API processes input text in several stages. All steps together form a processing channel/flow, also called pipes. The first stage is tokenization, which is mandatory. Later steps such as sentence limit detection (SBD) or part-of-speech (POS) tagging are optional. Optional steps are user-configurable. The following figure and table illustrate the actual Wink flow, see Figure 3:

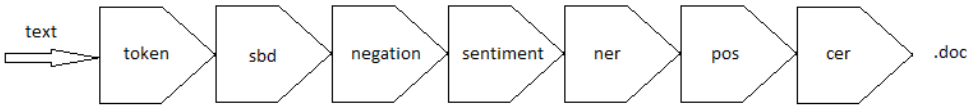


Figure 3. Wink processing flow, after <https://winkjs.org/wink-nlp/processing-pipeline.html>.

Accordingly to [3], there is a need for a compiler from Prolog (and extensions) to JavaScript, that may use logical programming (constraint) to develop client-side web applications while complying with current industry standards. Converting code into JavaScript makes (C)LP programs executable in almost any modern computing device, with no additional software requirements from the user’s point of view. The use of a very high-level language facilitates the development of complex and high-quality software. Tau Prolog is a client-side Prolog interpreter, implemented entirely in JavaScript and designed to promote the applicability and portability of Prologue text and data between multiple data processing systems. Tau Prolog has been developed for use with Node or a seamless browser.js and allows browser event management and modification of a web’s DOM using Prolog predicates, making Prolog even more powerful. [6] Tau-prolog provides an effective tool for implementing LFG: a sentence structure rule annotated with functional schemes such as S → NP, VP. to be interpreted as: [7]

- the identification of the special grammatical relation to the subject position of any sentence analyzed by this clause vis-à-vis the NP appearing in it;
- the identification of all grammatical relations of the sentence with those of the VP.

The procedural semantics of the PROLOG are such that the instantiation of variables in a clause is inherited from the instantiation given by its subscope, if they succeed. Another way to deal with logic programming is using a dedicated library [8] allowing us to declare facts and rules functional style, a step further to constraint programming, an interesting paradigm we aim to explore in our future research.

3. Methodology

We see the process of understanding natural language as the application of a complex H function that achieves the transformation of an external form into a certain understanding in a particular field of knowledge. One strategy to define H is to decompose it into a linear sequence of functions h , which applies to intermediate structures S_i :

$$H(P) = h_n \circ h_{n-1} \circ h_{n-2} \circ \dots \circ h_1(P). \quad (3)$$

Decomposition is motivated by linguistic and mathematical considerations. Then, for computational reasons, h_i may again be decomposed or, conversely, integrated. The exact nature of each S_i and h_i is not yet completely clear to [9], yet, within the logical programming paradigm, we see h , as rewriting systems. After lexical analysis of the text and identification of words with the help of the token function, a first step is to identify the parts of the sentence. Extremely useful again is binary development, this time at the level of sentence, dividing the statement into noun phrase (NP) and verbal phrase (VP). Recursive development is done after the second term, decomposed into a new NP, VP and so on. For example, the process of syntactic analysis rewrites a sentence in a syntactic tree, please see Figure 4:

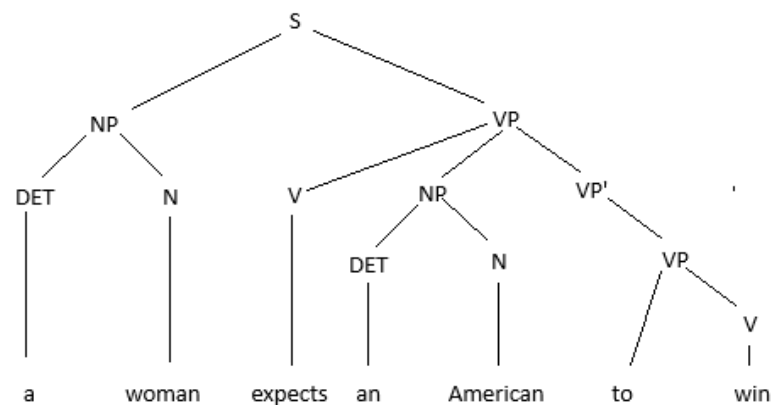


Figure 4. Syntactic tree.

Then (roughly speaking) a semantic interpretation process rewrites the syntactic tree into a logic formula. Finally, this logical formula is rewritten into a set of PROLOG clauses. The program loads the wink-nlp package, imports an English language model, creates a session with tau-prolog, and performs natural language processing tasks using wink-nlp. It also defines a Prolog program, extracts entities from a given text, and queries the Prolog program using tau-prolog against the rules obtained by syntactic analysis (previous step).

1. The required packages and modules are imported using the require function. The wink-nlp package is imported as winkNLP, and the English language model is imported accordingly:

```
// Load required packages and modules:
const winkNLP = require('wink-nlp');
const model = require('wink-eng-lite-web-model');
const pl = require("tau-prolog");
```

2. The tau-prolog package is imported as pl, and a session is created with pl.create(1000):

```
// Create a new session:
const session = pl.create(1000);
```

3. The winkNLP function is invoked with the imported model to instantiate the nlp object:

```
// Instantiate winkNLP:
```

```
const nlp = winkNLP(model);
```

4. The `its` and `show` variables are assigned to `nlp.its` and a function that logs the formatted answer from the `tau-prolog` session, respectively:

```
// Define helper functions:
const its = nlp.its;
const showAnswer = x => console.log(session.format_answer(x));
```

5. The `item` variable is assigned the value of the third argument passed to the Node.js script using `process.argv[2]`:

```
// Get command line argument:
const inputItem = process.argv[2]; //'the boy eats the apples.the woman runs the alley';
// in the back. a woman runs freely on the alley';
```

6. The `program` variable is assigned a Prolog program represented as a string. It defines rules for sentence structure, including noun phrases, verb phrases, and intransitive verbs. The program also includes rules for intransitive verbs, e.g. "runs" and "laughs":[\[10\]](#)

```
// Define the program and goal:
let program = `
s(A,B) :- np(A,C), vp(C,D), punct(D,B).
np(A,B) :- proper_noun(A,B).
np(A,B) :- det(A,C), noun(C,B).
vp(A,B) :- verb(A,C), np(C,B).
vp(A, B) :- intransitive_verb(A, B).
proper_noun([Eesha|A],A).
proper_noun([Eeshan|A],A).
intransitive_verb([runs|A],A).
intransitive_verb([laughs|A],A).
punct(A,A).
`;
```

7. The `nlp.readDoc` function is used to create a document object from the `inputItem`. The code then iterates over each sentence and token in the document, extracting the type of entity and its part of speech:

```
const doc = nlp.readDoc(inputItem);
let entityMap = new Map();
// Extract entities from the text:
doc.sentences().each((sentence) => {
  sentence.tokens().each((token) => {
    entityMap.set(token.out(its.value), token.out(its.pos));
  });});
```

8. The extracted entities and their parts of speech are stored in a `Map` object as Prolog rules:

```
// Add entity rules to the program:
const mapEntriesToString = (entries) => {
  return Array.from(entries, ([k, v]) => `n ${v.toLowerCase()}(S0,S) :- S0=[${k.toLowerCase()}|S].`).join("") + "n";
}
//console.log(mapEntriesToString([...entityMap.entries()]));
```

9. The generated Prolog rules are appended to the program string:

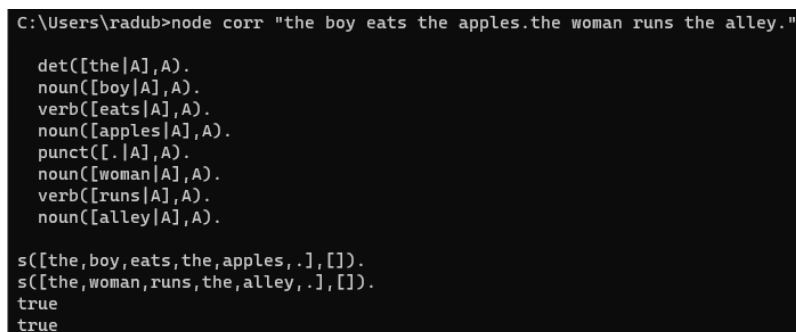
```
program += mapEntriesToString([...entityMap.entries()]);
```

10. The session.consult function is used to load the Prolog program into the tau-prolog session. Then, the session.query function is used to query the loaded program with the specified goals. The session.answers function is used to display the answers obtained from the query:

```
doc.sentences().each((sentence) => {
  let goals = 's([${sentence.tokens().out()}], []).';
  session.consult(program, {
    success: function() {
      session.query(goals, {
        success: function() {
          session.answers(showAnswer);
        }}}});
});
```

4. Results

Basically, the program measures the impedance between WinkNLP and Tau-Prolog language models. It is a matter of tuning both in order to get the optimum results, this is to map and filter the output of WinkNLP according to the DCG Prolog inference rules, since the lexicon is obtained by consuming its own WinkNLP results, see the results in Figure 5:



```
C:\Users\radub>node corr "the boy eats the apples.the woman runs the alley."

det([the|A],A).
noun([boy|A],A).
verb([eats|A],A).
noun([apples|A],A).
punct([.|A],A).
noun([woman|A],A).
verb([runs|A],A).
noun([alley|A],A).

s([the,boy,eats,the,apples,.],[]).
s([the,woman,runs,the,alley,.],[]).
true
true
```

Figure 5. The result of corr's execution.

This is most likely the field of AI (e.g. <https://sunilchomal.github.io/GECwBERT/#c-bert>) to choose the appropriate language model in order to get the minimum entropy or information loss.

5. Discussion

If the required packages (wink-nlp, wink-eng-lite-web-model and tau-prolog) are not installed, the code will throw an error. Also, if the Node.js script is not executed with a third argument, the item variable will be undefined, which may cause issues later in the code. In our future research will add error handling to gracefully handle any exceptions thrown during package imports or function invocations, and, eventually, implement additional natural language processing tasks using the wink-nlp package. Also, we aim to enhance the Prolog program to handle more complex sentence structures and semantic relationships using an extended DCG parser (e.g. <https://github.com/hfeky/definite-clause-grammar-parser/blob/main/dcgpl.pl>) and feel ready to take into account using a logic Javascript library (e.g. <https://github.com/mcsoto/LogicJS>) to replace the entire Prolog script from our source code.

Author Contributions: Conceptualization, R.M. and A.B.; methodology, R.M.; software, R.M.; validation, A.B.; formal analysis, A.B.; investigation, R.M.; resources, R.M.; data curation, A.B.; writing—original draft preparation,

R.M.; writing—review and editing, R.M.; visualization, A.B.; supervision, A.B.; project administration, R.M. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Conflicts of Interest: The authors declare no conflict of interest.

Abbreviations

The following abbreviations are used in this manuscript:

DCG	Definite Clause Grammar
NLP	Natural Language Processing
AI	Artificial Intelligence
SVO	Subject Verb Object
VSO	Verb Subject Object
OSV	Object Subject Verb
OS	Operating System
DOAJ	Directory of open access journals
LFG	Lexical-Functional Grammar
LP	Logic Programming

References

1. NLP: Building a Grammatical Error Correction Model. Available online: <https://towardsdatascience.com/nlp-building-a-grammatical-error-correction-model-deep-learning-analytics-c914c3a8331b> (accessed on 16 Aug. 2023).
2. Syntactic Analysis - Guide to Master Natural Language Processing(Part 11). Available online: <https://www.analyticsvidhya.com/blog/2021/06/part-11-step-by-step-guide-to-master-nlp-syntactic-analysis> (accessed on 16 Aug. 2023).
3. Relation Extraction and Entity Extraction in Text using NLP. Available online: <https://nikhilsrihari-nik.medium.com/identifying-entities-and-their-relations-in-text-76efa8c18194> (accessed on 16 Aug. 2023).
4. Discourse Representation Theory. Available online: <https://plato.stanford.edu/entries/discourse-representation-theory> (accessed on 16 Aug. 2023).
5. Jose F. Morales, Rémy Haemmerlé, Manuel Carro, and Manuel V. Hermenegildo. Lightweight compilation of (C)LP to JavaScript. *Theory and Practice of Logic Programming* **2012**, 12(4-5), 755–773, <https://doi.org/10.1017/S1471068412000336>.
6. An open source Prolog interpreter in JavaScript. Available online: <https://socket.dev/npm/package/tau-prolog> (accessed on 16 Aug. 2023).
7. Frey W.; Reyle U. A Prolog Implementation of Lexical Functional Grammar as a Base for a Natural Language Processing System. Conference of the European Chapter of the Association for Computational Linguistics (1983); URL: <https://api.semanticscholar.org/CorpusID:17161699>
8. Logic programming in JavaScript using LogicJS. Available online: <https://abdelrahman.sh/2022/05/logic-programming-in-javascript> (accessed on 16 Aug. 2023).
9. Saint-Dizier, P. An approach to natural-language semantics in logic programming; *Journal of Logic Programming*. *Journal of Logic Programming* **1986**, Volume 3, Issue 4), 329–356, [https://doi.org/10.1016/0743-1066\(86\)90010-5](https://doi.org/10.1016/0743-1066(86)90010-5).
10. Kamath, R, Jamsandekar, S., Kamat, R. Exploiting Prolog and Natural Language Processing for Simple English Grammar. In Proceedings of National Seminar NSRTIT-2015, CSIBER, Kolhapur, Date of Conference (March 2015); URL: https://www.researchgate.net/publication/280136353_Exploiting_Prolog_and_Natural_Language_Processing_for_Simple_English_Grammar.

Disclaimer/Publisher’s Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.