Article

Not peer-reviewed version

Making a Grammar Checker with Autocorrect Options Using NLP Tools

Radu Bucea Manea Tonis [*,‡] and Adrian Beteringhe [‡]

Posted Date: 8 September 2023

doi: 10.20944/preprints202308.1640.v4

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

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: radub_m@yahoo.com
[†] 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, than 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 work flow or pipeline of steps.The provided article combine the usage of wink-nlp and tau-prolog packages for natural language processing (NLP) and understanding (NLU), demonstrates the need of a supplementary logic layer based on beta reductions and provide a method to convert lambda abstractions into arrow Javascript functions.

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

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]

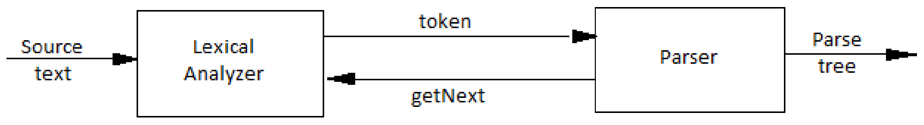


Figure 1. The interaction between lexical analyzer and parser, after [2].

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:

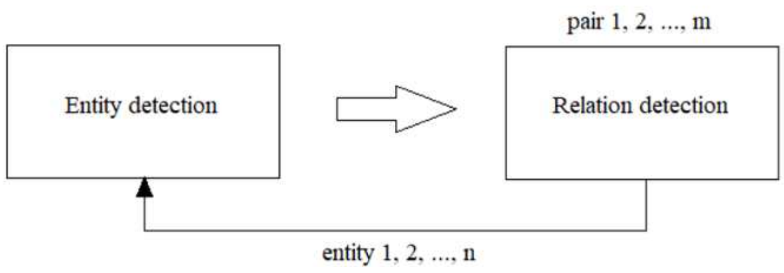


Figure 2. Inference pipeline architecture, after [3].

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:

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

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

In order to overrun this inconvenience we decide to improve our DCG syntax adding a new λ operator to bind free variables. There are several situations that will benefit from this approach: [9]

- Interpreting determiners, e.g. *a man* is not the same with *there is a man*;
- Type raising, e.g. the argument may become the function, like in the case of callback functions or inversion of control design pattern;
- Transitive verbs, e.g. the following expression $\lambda x.love(x,y)$ may improve the lexicon in this special case, and the expression $\lambda x.walk(x)$ will do the same for intransitive verbs;
- Quantifier and scope ambiguity, e.g. *In this country, a woman gives birth every 15 minutes.* and *Every man loves a woman* , respectively;
- Coordination or summing up, e.g. $walk(john) \wedge walk(mary)$.

This way we are getting closer to a Natural Language Understanding (NLU) component responsible for extracting information at a single step throughout a pipeline process consisting of several stages: [10] tokanization, syntactic analysis and generating the semantic grammar lexicon on the fly, based on the original term redexes, i.e. reduced forms.

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 code base 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 negation Flag 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 tagging.
cer	Custom entity recognition — detects all custom entities and their type and span.

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:

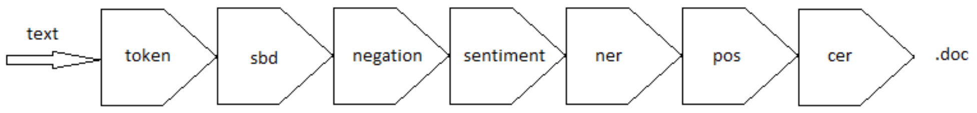


Figure 3. Wink processing flow, after https://winkjs.org/wink-nlp/processing-pipeline.html.

According 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 either 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 a Lexical-Functional Grammar (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 sub-scopes, 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:

$$H(P) = h_n \circ h_{n\text{-}1} \circ \ldots \circ h_1(P). \tag{3}$$

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 [11], yet, within the logical programming paradigm, he considers h_i 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 (NF) and verbal phrase (VF). Recursive development is done after the second term, decomposed into a new NF, VF and so on. For example, the process of syntactic analysis rewrites a sentence in a syntactic tree, please see Figure 4:

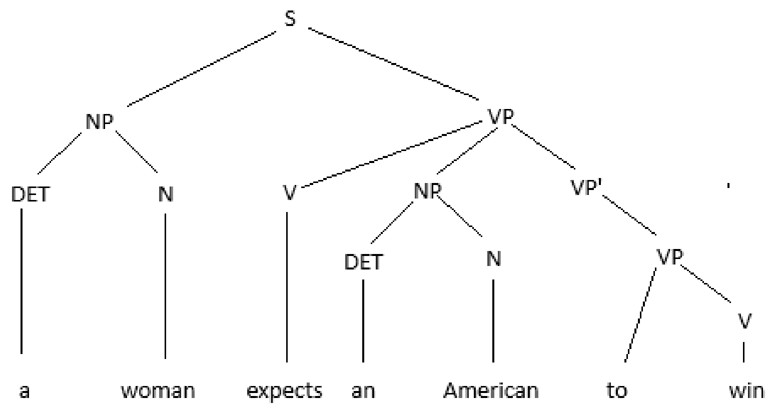


Figure 4. Syntactic tree.

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 winkNLP. 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":[14]

```
// 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). %λx.run(x)
intransitive_verb([laughs|A],A). %λx.laugh(x)
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:

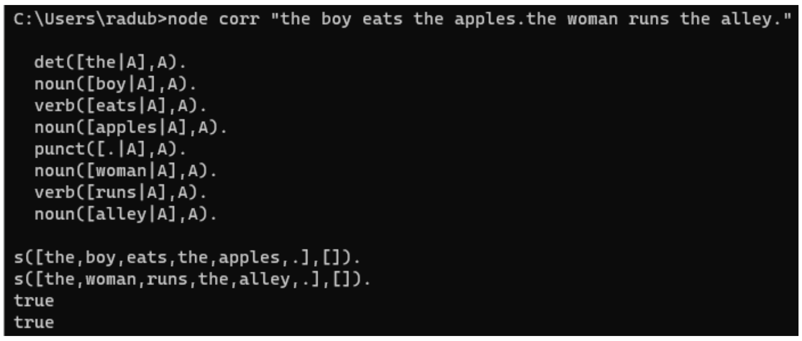
```
const doc = nlp.readDoc(inputItem);
let entityMap = new Map();
// Extract entities from the text:
doc.sentences().each((sentence) => {
sentence.tokens().each((token) => {
entityMap.set(token.out(its.value), token.out(its.pos));
});});
```

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);
}})}})});
```

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:



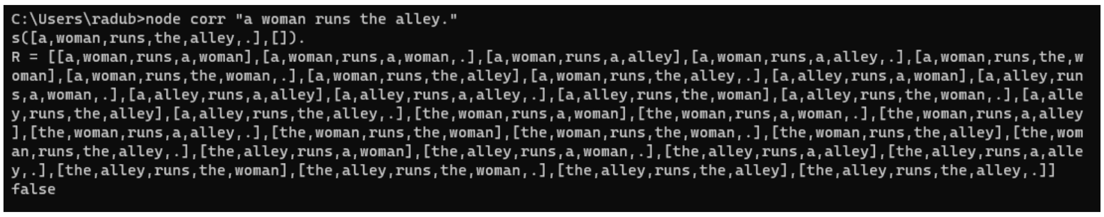
```
C:\Users\radub>node corr "the boy eats the apples.the woman runs the alley."

  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.

In order to show the possible valid combination of words, it suffice changing the program’s goal from ‘s([$sentence.tokens().out()],[]).‘ to ‘findall(M,s(M,[]),R).‘. The result will be a list of valid sentences according to the dynamic generated DCG lexicon, see Figure 6:



```
C:\Users\radub>node corr "a woman runs the alley."
s([a,woman,runs,the,alley,.],[]).
R = [[a,woman,runs,a,woman],[a,woman,runs,a,woman,.],[a,woman,runs,a,alley],[a,woman,runs,a,alley,.],[a,woman,runs,the,w
oman],[a,woman,runs,the,woman,.],[a,woman,runs,the,alley],[a,woman,runs,the,alley,.],[a,alley,runs,a,woman],[a,alley,run
s,a,woman,.],[a,alley,runs,a,alley],[a,alley,runs,a,alley,.],[a,alley,runs,the,woman],[a,alley,runs,the,woman,.],[a,alle
y,runs,the,alley],[a,alley,runs,the,alley,.],[the,woman,runs,a,woman],[the,woman,runs,a,woman,.],[the,woman,runs,a,alley
],[the,woman,runs,a,alley,.],[the,woman,runs,the,woman],[the,woman,runs,the,woman,.],[the,woman,runs,the,alley],[the,wom
an,runs,the,alley,.],[the,alley,runs,a,woman],[the,alley,runs,a,woman,.],[the,alley,runs,a,alley],[the,alley,runs,a,alle
y,.],[the,alley,runs,the,woman],[the,alley,runs,the,woman,.],[the,alley,runs,the,alley],[the,alley,runs,the,alley,.]]
false
```

Figure 6. The result of corr's *findall* execution.

It is important to notice that a determinant like 'a', i.e. ∃, almost triples the area of semantic field, thus emphasizes the importance of the semantic capabilities of the parser. It is obvious we have to run the *findall* method after each sentence not to combine the lexicon of the two sentences. Otherwise, the result is interesting, bring our program closer to generating AI features, e.g. chatGPT, rather than a normal grammatical corrector: *the boy eats the boy* , *the boy eats the apples* , *the boy eats the woman* , *the boy eats the alley* , *the boy runs the boy* , *the boy runs the apples* , *the boy runs the woman* , *the woman eats the apples* , and so on. 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.

We upgrade the syntactic tree with the lambdas constructs to obtain a semantic tree, please see Figure 7:

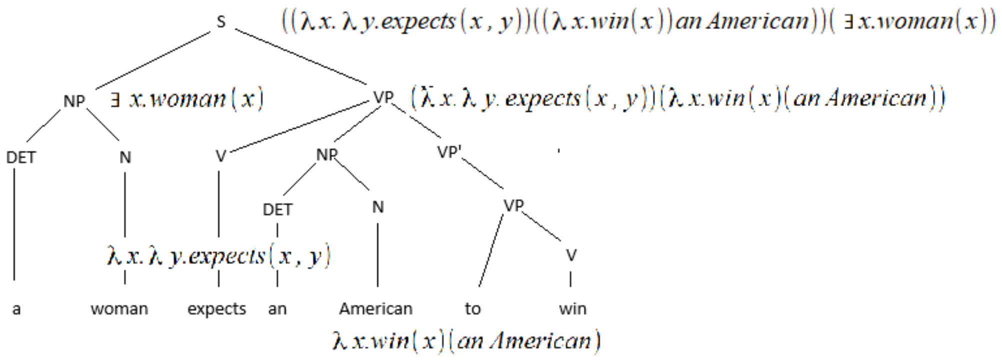

S
$((\lambda x.\lambda y.expects(x,y))((\lambda x.win(x))an\ American))(\exists x.woman(x))$
NP
$\exists x.woman(x)$
VP
$(\lambda x.\lambda y.expects(x,y))(\lambda x.win(x)(an\ American))$
DET
N
V
NP
VP'
DET
N
VP
V
$\lambda x.\lambda y.expects(x,y)$
a
woman
expects
an
American
to
win
$\lambda x.win(x)(an\ American)$

Figure 7. Semantic tree.

The most evident advantage is applying beta reductions when parsing the whole expression semantically:

$$(\lambda x.\lambda y.expects(x,y)((\lambda x.win(x))an\_American))(\exists x.woman(x)):$$

1. β

$$(\lambda x.\lambda y.expectsx,y((\lambda x.winx)an\_American))(\exists x.womanx)$$

2. β

$$\lambda y.expectsx,y((\lambda x.winx)an\_American)$$

3.

$$\lambda y.expectsx,y(winan\_American)$$

Roughly speaking, the semantic interpretation process rewrites the syntactic tree into a logic formula. It is an expression where the bound variables occur at several nesting depths, please see Figure 8:

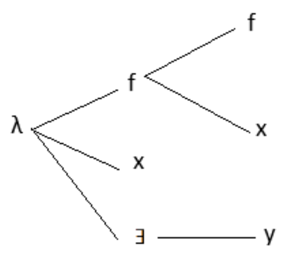


Figure 8. Viewing the term nesting structure.

In the name-free notation, no variable name appears after the λ symbol and bound variable indexes appear as numbers. The name x of a bound variable serves only to match each occurrence of x with its binding λ x so each occurrence of a bound variable is represented by an index, giving the number of abstractions lying between it and its binding abstraction. [12] In the name-free notation, the three occurrences of x are represented by 0 and 1, and 0 for the occurrence of y:

$$\lambda((f0)f1)(\exists 0)$$

In order to parse complex lambda expressions we have to compile a parser, according to antlr.org. ANother Tool for Language Recognition (ANTLR) is a parser generator for reading, processing, executing, or translating structured text. It is used to build languages and tools from grammars by generating parsers that can build and walk semantic trees. [13] In that matter we need to provide a specific grammar for lambda calculus and we started with these simple lines inspired from github.com/antlr/grammars-v4:

```
expression
:  VARIABLE | function_ | application
;
function_
:  'λ' VARIABLE '.' scope
;
application
:  '(' expression expression ')'
;
scope
:  expression
;
VARIABLE
:  [a-z] [a-zA-Z0-9]*
;
WS
:  [/t/r/n] -> skip
;
```

When we parse the expression $((\lambda ab.ab)((\lambda a.a)c))((\lambda a.a)d)$ with the command *antlr4-parse lambda.g4 file_ -gui* , we get the next tree, please see the Figure 9:

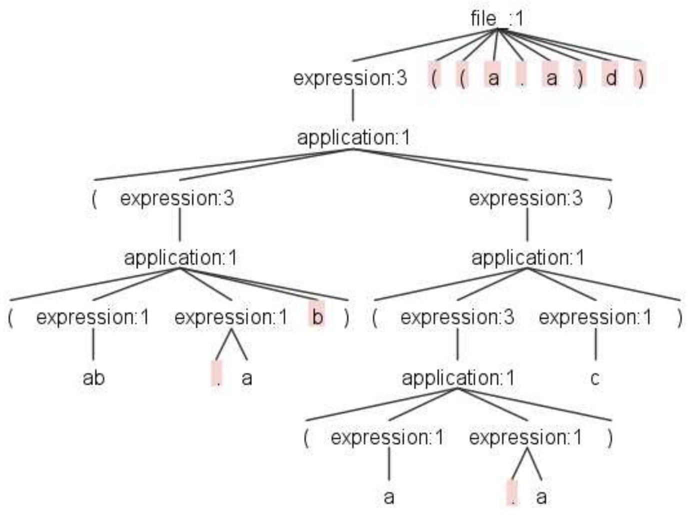


Figure 9. The ANTLR window showing the parse tree.

Finally, the lambdas are implemented in Javascript as abstractions, variables and applications:[15]

```
'use strict';
const walk = x => true;
const beat = (x,y) => true;
const exist = x => x;
const any = x => x;
const is = (x,y) => true;
const not = x => !x;
let Or = "||";
let And = '&&';
let John = "is('man','john')";
let Walk = "walk(any('man'))";
let Not = "not(true)";
let Beat = "beat('john','drums')";
let Any = "any('john')";
```

It is important to observe that variables are mapped into arrow functions, after being themselves mapped for each given token. The parser read each line of text at a time and recursively evaluates the current expression in two steps:

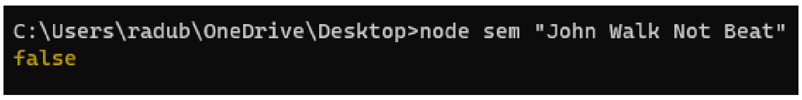
```
let s = process.argv[2];
console.log(eval(eval(s.split(' ').map(x=>return x;).join('+ And +'))));
```

4. Results

The expression *John Walk Not Beat* is logically evaluated as

$$is('man','john')\&\&walk(any('man'))\&\&not(true)\&\&beat('john','drums')$$

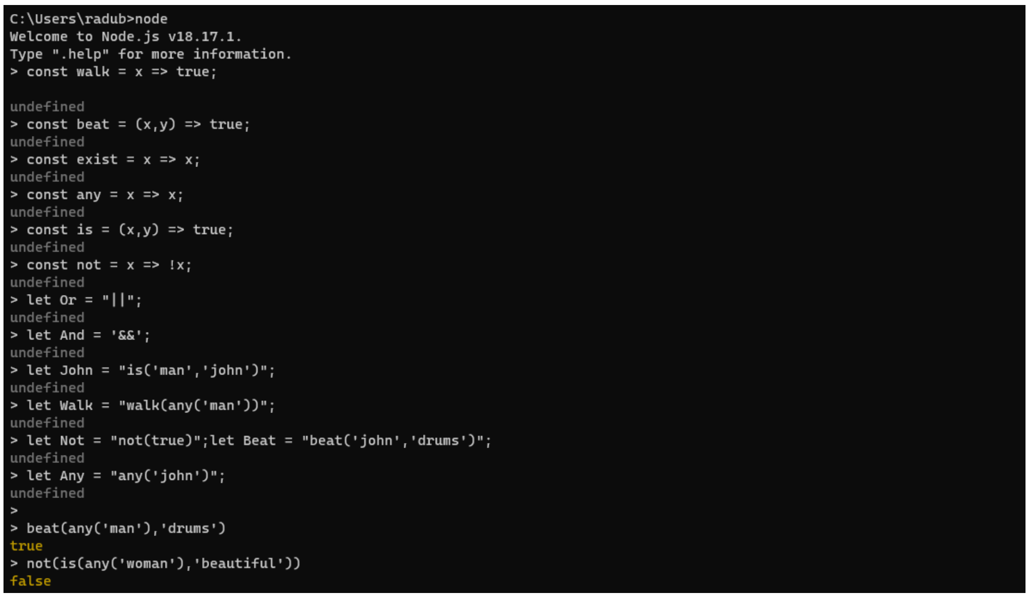
, as may be seen in Figure 10:



```
C:\Users\radub\OneDrive\Desktop>node sem "John Walk Not Beat"
false
```

Figure 10. The result of parsing the *John Walk Not Beat* expression.

The result is false because if John has the ability to walk and every person that walk can beat a drum, then John also can beat a drum. The expressions *beat(any('man'),'drums')* and *not(is(any('woman'),'beautiful'))* are evaluated in the same way when we load the entire program in Node and consult it like an ordinary Prolog database of facts and rules, see Figure 11:



```
C:\Users\radub>node
Welcome to Node.js v18.17.1.
Type ".help" for more information.
> const walk = x => true;

undefined
> const beat = (x,y) => true;
undefined
> const exist = x => x;
undefined
> const any = x => x;
undefined
> const is = (x,y) => true;
undefined
> const not = x => !x;
undefined
> let Or = "||";
undefined
> let And = '&&';
undefined
> let John = "is('man','john')";
undefined
> let Walk = "walk(any('man'))";
undefined
> let Not = "not(true)";let Beat = "beat('john','drums')";
undefined
> let Any = "any('john')";
undefined
>
> beat(any('man'),'drums')
true
> not(is(any('woman'),'beautiful'))
false
```

Figure 11. Prompt line evaluation in NodeJS.

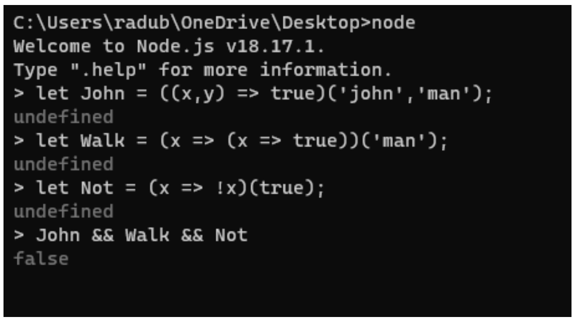
Another way to write the program is by using functors and merging together the abstractions and the applications:

```
let John = ((x,y) => true)('john','man');
let Walk = (x => (x => true))('man');
let Not = (x => !x)(true);
```

The expression is evaluated after a single step, since the tokens are not mapped into variables anymore:

```
John && Walk && Not
```

The result is evaluated to false because if John is a man and every man can walk, John should also been able to walk, please see Figure 12:



```
C:\Users\radub\OneDrive\Desktop>node
Welcome to Node.js v18.17.1.
Type ".help" for more information.
> let John = ((x,y) => true)('john','man');
undefined
> let Walk = (x => (x => true))('man');
undefined
> let Not = (x => !x)(true);
undefined
> John && Walk && Not
false
```

Figure 12. The result of parsing the *John Walk Not Beat* expression.

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 and lambda calculus. Also, we succeeded to replace the Prolog program and handle more complex sentence structures and semantic relationships using our own Javascript code base for NLP (i.e. github.com/radubm1/NLP).

Author Contributions: For research articles with several authors, a short paragraph specifying their individual contributions must be provided. The following statements should be used "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.", please turn to the CRediT taxonomy for the term explanation.

Funding: This research received no external funding.

Acknowledgments: In this section you can acknowledge any support given which is not covered by the author contribution or funding sections. This may include administrative and technical support, or donations in kind (e.g., materials used for experiments).

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
NLU	Natural Language Understanding
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
ANTLR	ANother Tool for Language Recognition

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. Introduction to semantic parsing. Available online: https://stanford.edu/class/cs224u/2018/materials/cs224u-2018-intro-semparse.pdf (accessed on 22 Aug. 2023).
10. Bercaru, G.; Truică, C.-O.; Chiru, C.-G.; Rebedea, T. Improving Intent Classification Using Unlabeled Data from Large Corpora. Mathematics 2023, 11, 769. https://doi.org/10.3390/math11030769.
11. 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.
12. Paulson, L.C. Writing Interpreters for the λ -Calculus. In *ML for the Working Programmer*; Cambridge University Press, Country, 2007; pp. 357 – 396; ; DOI: https://doi.org/10.1017/CBO9780511811326.011.
13. Parr, T. Introducing ANTLR and Computer Languages. In *The Definitive ANTLR 4 Reference*; Davidson Pfalzer, Country, 2013; pp. 328; ISBN: 9781934356999.
14. 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: URL: https://www.researchgate.net/publication/280136353_Exploiting_Prolog_and_Natural_Language_Processing_for_Simple_English_Grammar.
15. An introduction to the roots of functional programming. Available online: https://medium.com/@ahlechandre/lambda-calculus-with-javascript-897f7e81f259 (accessed on 28 Aug. 2023).