# Application of the Fibonacci Series in Natural Language Processing

Radu Bucea-Manea-Tonis[1], Adrian Beteringhe[2]

**Abstract:** The article investigates some possibilities of using the new Javascript programming language for generating parsing trees and histograms for natural language processing (NLP). At the same time, we are trying to break away from the AI-type paradigm ("stochastic parrot") applied in language theory considering it too limited and focusing more on the rational approach, exploring the functional characteristics of Javascript language in the key of predicate logic. Two of Chomsky's ideas were crucial for the development of our research: first, there is a basal grammar innate to the child that provides the very structures the language is built upon, and second, the recursive nature of the human language Chomsky had noticed that allows us to build an indefinite amount of statements from a finite set of grammar rules, plus the composable character of grammar that begets infinite long verbal structures. The last observation gave us the idea to establish isomorphic relations between natural language and formal systems to prove our theorems (future work). The agglutinative mechanism of making both sensical or not-sensical verbal content drew our attention to the Fibonacci series of numbers that is fundamental for developing living structures both at the molecular and macro level (e.g. breeding of rabbits, use of the phi constant in architecture, and so on). This way the bigrams (from n-grams) can result from concatenating strings one after another creating entities in a coherent, linked-list style, rational manner. These collocations may be interpreted either as new concepts (e.g. military-doctor) or camouflaged predicates built upon identity principles (e.g. is or exists). The practical way we decided to test our hypothesis was to employ the functional capabilities of Javascript programming language that brings us even closer to the logical nature of the human language. The new ES6 streaming process of transforming a text was another aspect similar to the pipeline style of the human brain in processing data. The two-way parsing on texts calculating the frequency of pairs' appearance proved to be of significant importance in dead language studies or searching for anagrams. The use of the Wink software package language model allowed us to create

[1]Senior professor, Danubius University of Galati, Romania, Address: 3 Galati Blvd., 800654 Galati, Romania, Tel.: +40727888619, 0372 361 226, E-mail: radumanea@univ-danubius.ro.
[2]Associate Professor, Danubius University of Galati, Romania, Address: 3 Galati Blvd., 800654 Galati, Romania, Tel.: 0372 361 226, Corresponding author: adrianbeteringhe@univ-danubius.ro.

predicates like verbs (subject, object) based on the SVO structure of IE languages, the future knowledge base for our next proofing language system.

**Keywords:** functional programming; language analysis; mathematical logic; parsing trees

## 1. Introduction

Automatic natural language processing (NLP) began in the 1950s when Alan Turing published Computing Machinery and Intelligence, a pioneering paper on artificial intelligence. Related to the first grasp on language learning, was the binary generation of verbal phrases, the peripheral mechanism being the most notable example in this matter. *Colorless green ideas sleep furiously* was Chomsky's example of a sentence that is grammatically well-formed, but semantically nonsensical used in his 1957 Syntactical Structures book. It is developed almost like a balanced binary tree, and the following year (1958), the linguist and anthropologist Dell Hymes demonstrated how that nonsense words can develop into something meaningful when in the right sequence: *Hued ideas mock the brain,/ Notions of color not yet color,/ Of pure, touchless, branching pallor/ Of invading, essential Green...*

Two of the successful NLP systems developed in the 1960s were SHRDLU and ELIZA. Until the 1980s, most natural language processing systems relied solely on complex sets of manually coded rules. NLP began to develop after the introduction of machine learning algorithms starting in the 1980s. (Brownlee, 2017)

It has been observed that certain pairs of words have a higher frequency than others accordingly to the principle of semantic localization. For example, the following combinations (collocations) are encountered with a high frequency: doctor engineer, civil engineer, head of service, military doctor, etc. In addition, the direction of parsing the word tree may indicate a variable number of such combinations, an aspect of scientific importance in the case of extinct languages research, where the meaning of writing can be obtained from left to right, from right to left, or alternately in both directions (gr. boustrophedon), which will broaden the basis of analysis. Historically, the invertible computational approach has proven useful in understanding energy conservation, studying entropy, and understanding information transformation and transmission (Kristensen, Kaarsgaard & Thomsen, 2022).

Final submissions not following the required format will be returned to the authors for modification and compliance. All scientific papers should be written in English or French. The abstract and the keywords must be written in English.

## 2. Materials

The increasing frequency of the CPU clock, the increasing number of interconnected devices, and the demand for more computing power have led to a change in the current programming paradigm. All of this poses notable challenges for software developers to overcome these constraints. Added to these limitations are the lack of programming expertise among researchers and ordinary users, the high cost of automated reasoning systems (e.g. Rational Rose Enterprise from IBM), poor accessibility to source code, and distributed access to different system components that raise problems in the enforcement of intellectual property rights (Khanfor & Yang, 2017; Ljunglof, 2002).

In the following table, we analyze several programming languages according to their suitability for language analysis: object-oriented (OO), functional (FP), and imperative programming (IL), according to Table 1:

**Table 1. Suitability of Programming Languages in Distributed Computing Systems, According to (Khanfor & Yang, 2017)**

| | Weaknesses | Strengths |
|---|---|---|
| OO | | The simplicity of OOP makes it easy to represent many problems;<br>Numerous OOP tool frameworks;<br>A large number of programmers are familiar with OOP. |
| FP | Pure functional style only in large and very large projects;<br>Lack of expert developers in this paradigm;<br>Complexity and cost of software design tasks;<br>Small changes may require extensive program restructuring to cope with these changes;<br>A complex register of modules. | Writing programs using functional languages increases operational safety;<br>Mitigates security risks by prohibiting stateless changes;<br>Top-level features encourage and promote reuse;<br>Recursive calls promote code and feature reuse;<br>Plugin management systems and professional execution environments (e.g. Node) are used<br>Highly efficient immutable data structures for distributed systems; |

| | | |
|---|---|---|
| | | No side-effects. |
| IL | They tend to be more extensive in the number of lines of code; Increased estimates relative to software development efforts; Mutable data structures can lead to semantics changes in the execution results. | |

The advantages made available to programmers by functional programming languages (Haskel, Scala, Clojure, F#, etc.), of which we mention, especially modularity, recommend this paradigm in natural language analysis in the following directions (Khanfor & Yang, 2017; Ljunglof, 2002).

- Functional morphology - higher-order functions, static and dynamic type resolution;
- Morphological and syntactic analysis - defining combinators to describe rule templates;
- Semantic parsing - taxonomies handling algorithms are intuitive and allow direct implementation in functional languages.

In addition, even the order of function composition may have ontological value, e.g. belief (‘he’, do (‘a job’, is (‘good’))), where belief, do, and is are predicates with one or more arguments. It should be noted that functional languages make no distinction between a fundamental parameter and a functor, moreover, most of the time (e.g. Javascript) the type is dynamically inferred. Thanks to this observation, language models have been created for the NLP based on pairs of words (bigrams), which by the order of their chaining within sentences justify a grammatical structure (de Kok & Brouwer, 2011).

Javascript immutability (eg. Freeze () method for objects) and local scope for variables it uses (declared with let keyword) are natural consequences of this philosophy. Generalizing the use of functors was another functional concept that merged data and functions into a single parameter, suitable for both data & behaviour interchange between objects. The arrow functions facilitated this even more by declaring a functor and initializing it in one single line of code. Another major role played here is anonymous and immediate functions.

Nesting functions, a feature available before ES5, were gradually replaced by currying, a special way to employ binding, a concept introduced first by the Haskel programming language. In this respect, parameters are introduced one after another inside individual parenthesis "()" suffixing the called function name, and being passed in the same order as arguments of nested anonymous functions.

All these premises made way for callback style use of Javascript that made even more room for asynchronous/event use with the advent of Promises. In this style, the nesting is achieved at the args level, explicitly providing arrow functions with anonymous implementations instead of parameters. Implicit arguments (e.g. "a=1"), a variable number or parameters (e.g. "…args"), and memoizing, a technique enabled by closures (accessing out-of-context variables by functions) and higher order functions, which, alongside tail-calling, dramatically improve the performance of recursive functions.

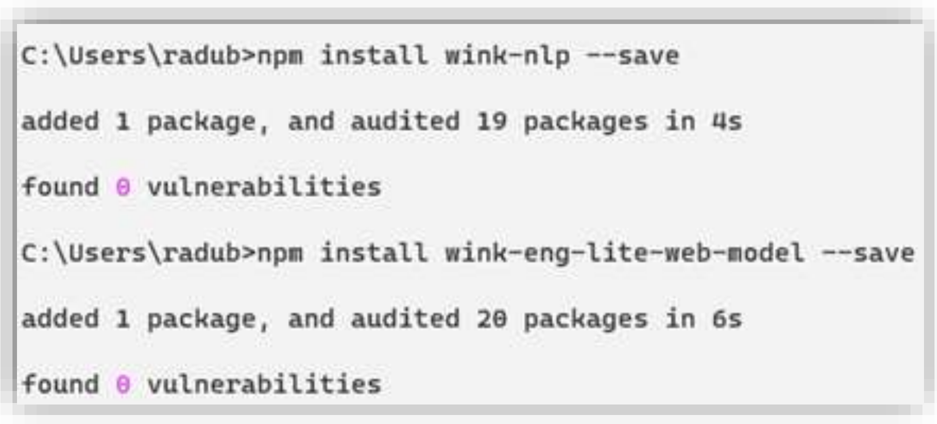
Corroborating data with Popularity of Programming Language Index (PyPL) - Python, 27.7%; Java, 16.79%; Javascript, 9.65%, shows that the multi-paradigm Javascript language meets the qualities necessary for an Open source approach to natural language analysis (Krill, 2023).

There are many useful NLP libraries available like NLTK (Python), and Apache OpenNLP (Java). Most of these libraries are not available in Javascript, so there is a daunting task to find proper NLP libraries in Javascript. The following are the libraries found by research and testing, after (Avinash, 2020):

- NLP.js is an NLP library for building bots developed by the AXA group. It provides entity extraction, sentiment analysis, automatic language identification, and more, supporting 40 languages. (https://github.com/axa-group/nlp.js)
- Natural is a general language facility for Node.js and a well-known NLP library. It currently supports tokenizing, stemming, classification, phonetics, term frequency–inverse document frequency (tf-idf), WordNet, string similarity, and some inflections (https://github.com/NaturalNode/natural).
- Compromise.cool is a lightweight library that can be used to run NLP on main browsers, and works with the English language only. (https://github.com/spencermountain/compromise/)
- Wink.js provides NLP functions for amplifying negations, managing elisions, creating n-grams, stems, phonetic codes to tokens, and more (https://github.com/winkjs/wink-nlp-utils).

## 3. Methodology

For start, we decided to use the Wink facility to generate bigrams (https://winkjs.org/wink-nlp-utils/tokens-bigrams.js.html). For this purpose, we'll install the wink-nlp package. Next, we'll use that package to install the wink-eng-lite-web-model https://winkjs.org/wink-nlp/getting-started.html, please see the figure 1:

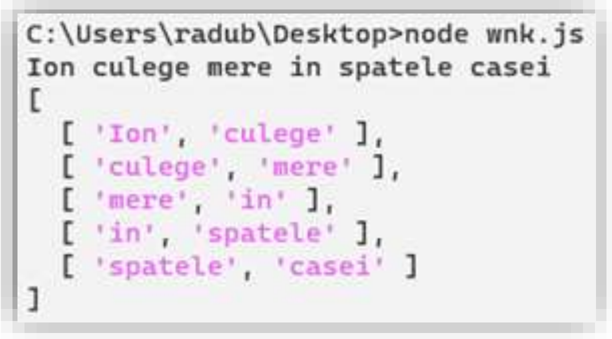


```
C:\Users\radub>npm install wink-nlp --save

added 1 package, and audited 19 packages in 4s

found 0 vulnerabilities

C:\Users\radub>npm install wink-eng-lite-web-model --save

added 1 package, and audited 20 packages in 6s

found 0 vulnerabilities
```

**Figure 1. Installing wink-nlp package under Node with npm@9.8.0**

We start our code by loading the wink-nlp package, some helpers, and the language model. After that, we instantiate winkNLP using the language model:

```
 // Load wink-nlp package.
const winkNLP = require ('wink-nlp');
// Load English language model — light version.
const model = require ('wink-eng-lite-web-model');
// Instantiate winkNLP.
const nlp = winkNLP (model);
const text = 'Ion culege mere in spatele casei';
const doc = nlp.readDoc(text );
console.log( doc.out() );
var bigrams = function (tokens) {
 // Bigrams will be stored here.
 var bgs = [];
 // Helper variables.
 var i, imax;
 // Create bigrams.
for (i = 0, imax = tokens.length - 1; i < imax; i += 1 ) {
bgs.push( [ tokens[ i ], tokens[ i + 1 ] ] );
}
return bgs;
}; //bigrams()
console.log(bigrams(doc.tokens().out()));
```



```
C:\Users\radub\Desktop>node wnk.js
Ion culege mere in spatele casei
[
  [ 'Ion', 'culege' ],
  [ 'culege', 'mere' ],
  [ 'mere', 'in' ],
  [ 'in', 'spatele' ],
  [ 'spatele', 'casei' ]
]
```

**Figure 2. Executing bigrams () token function with winkNLP**

To avoid using a dependent token-generating function (e.g. tokens ()) on the vocabulary specific to each language – we have only wink-eng-lite-web language model - we'll use a vector of strings, neutral in terms of language and unequivocal in the case of lexical atoms. We intend to start from a simple sentence like "Ion culege mere in spatele casei" which we will turn into a linear vector of strings.

We will then apply the series of Fibonacci numbers to direct access indexes to obtain the pairs of chained words in the form of an unbalanced binary list or tree:

['Ion culege', 'culege mere', 'mere in spatele', 'in spatele casei']

The algorithm simulates dragging a window of n × words from left to right, as in the case of a finite state automaton, except that the symbols are read two at a time, and the dragging is conditioned by every second token read. Please note that every second term of the list items plays the role of reference (pointer) to the first term in the next item.

In 1957, Noam Chomsky noticed this would be the generating principle of sentences with the famous example “Colorless green ideas sleep furiously”. It follows that pairs of words have a meaning taken separately and provide grammatical structure to the sentence, even if the whole is meaningless, as shown in the following sequence:(de Kok & Brouwer, 2011)

[[“Colorless”, “green”],[“green”, “ideas”],[“ideas”, “sleep”],[“sleep”, “furiously”]]

To obtain such a list, we apply split () function to the original text variable with a space character as the input argument, initializing the arg vector with the following elements (unigrams):

```
let text = ‘Elena creste o mandra floare albastra albastra floare albastra floare’;

let arg = text.split(‘ ‘);
```

, then complete the following sequence of steps:(de Kok & Brouwer, 2011)

I. Place the window of two words at the beginning of the vector and generate the first pair [“Elena creste”];

II. Then we slide the window’s position one at a time and save the next pair [“creste o mandra”];

III. Repeat the second step until the end, thus extracting all available pairs in the new vector (bigram):

```
let bigram = n => {return Array(n). fill (0).reduce((arr,_,i)=>{

arr.push((i>1)?arg[i-2].concat(“ ”, arg[i-1]):i);

return arr;

},[]);

};
```

## 4. Results

The combinatorial properties of words have always benefited from the interest of linguists, so we need to make a statistic (histogram) that orders combinations of words according to the frequency of their appearance in a text. First, we will extract all pairs of words into a new vector (arr), please see Figures 3 and 4:

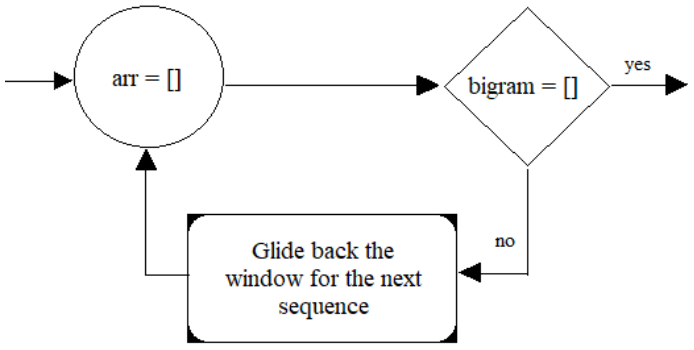
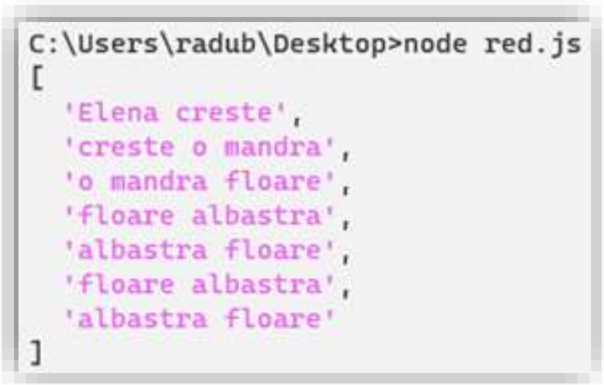


**Figure 3. Janus Style Loop for Bigram Vector Generation**

let arr = bigram(arg.length+1);

arr.splice(0,2);



```
C:\Users\radub\Desktop>node red.js
[
  'Elena creste',
  'creste o mandra',
  'o mandra floare',
  'floare albastra',
  'albastra floare',
  'floare albastra',
  'albastra floare'
]
```

**Figure 4. The Result of Obtaining a New Linear Vector by Eliminating the First Two Values of the Fibonacci Series**

In the next step we'll increment the total occurrences of each pair of tokens in the previously resulting bigram vector and map each pair in the vector with its frequency of occurrence, please see Figure 5:

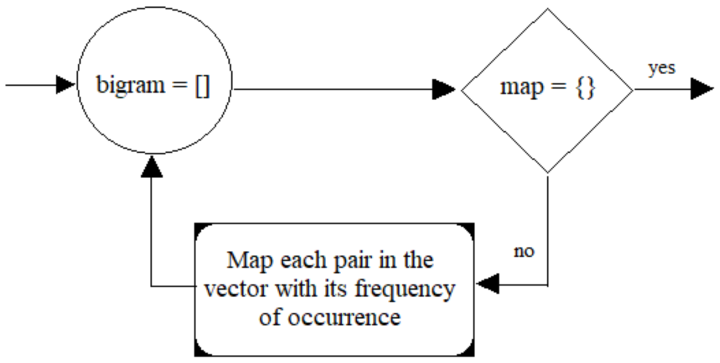
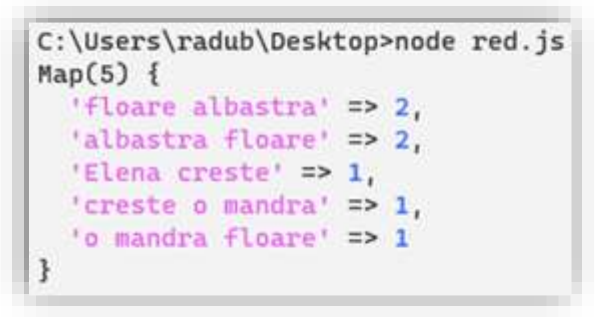

bigram = []
map = {}
yes
no
Map each pair in the vector with its frequency of occurrence

**Figure 5. Janus Loop Style for Mapping the Vector Pairs with the Frequency of Occurrence**

```
let count = arr => {
    return arr.reduce((total, colloc) => {
    total[colloc] ? total[colloc]++ : total[colloc] = 1;
    return total;
    }, {});
};
const map = new Map(Object.entries(count(arr)));
```

Finally, we order the entries of the associative table (map) by values instead of keys, please see Figure 6:

```
var mapDesc = new Map([...map.entries()].sort((x,y)=>y[1]-x[1]));
console.log(mapDesc);
```



```
C:\Users\radub\Desktop>node red.js
Map(5) {
  'floare albastra' => 2,
  'albastra floare' => 2,
  'Elena creste' => 1,
  'creste o mandra' => 1,
  'o mandra floare' => 1
}
```

**Figure 6. The Result of Composing the Bigram () and Count () Functions**

According to tf-idf numerical statistics, the application logic can be interpreted in two different ways. In the first case, when the text is parsed in the conventional direction (from left to right), a pair is assigned a value according to the frequency of occurrence. In the second case, when the text is parsed in the opposite direction, a value (frequency) associated with a specific pair within the associative dictionary (map) is compared with the new frequency of occurrence of the inverted pair, see diagram in Figure 7:

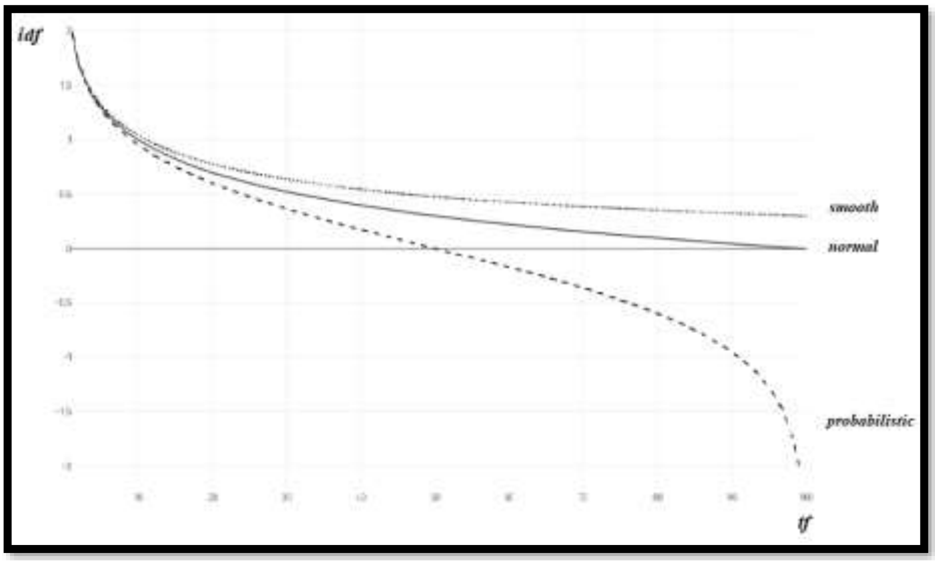


**Figure 7. Plotting Inverse Document Functions, After Wikipedia, URL: https://en.wikipedia.org/wiki/Tf-idf**

The convergence of the string towards the necessary and sufficient condition of fulfilling the anagram quality of a string in the vector is verified, applying the Squeeze theorem within the series of absolute frequencies, please see Figure 8:

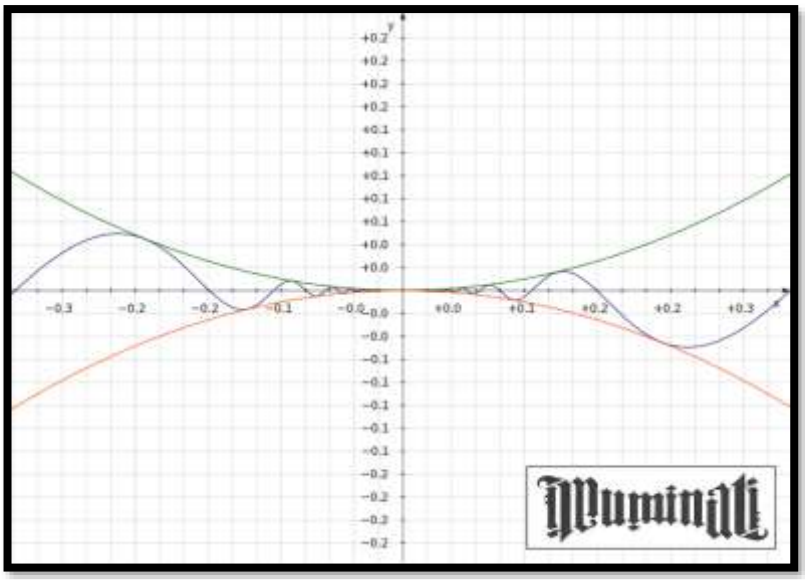


**Figure 8. Anagram Analogy with Plotted Mathematical Function, by KmPlot, URL: https://edu.kde.org/**

## 3. Conclusion

It follows from various studies, including our own, that use of pure FP languages ensures the safe use of programs by guaranteeing there are no state changes in software execution.(Khanfor & Yang, 2017) Since the programs' invertibility is generally undecidable (Kristensen & Kaarsgaard & Thomsen, 2022), all we can get from our analysis of the text is a reasonable approximation, thus any analysis of the text will divide the expressible pairs (collocations) into three categories: those that turn out to be invertible, those that turn out not to be invertible, and those for which analysis cannot be given a clear answer.

Our approach is to make this class of programs as accessible and useful as possible to achieve the intended purpose. It is a fact that Javascript multi-paradigm language has provided us with an invertible, more concise, more familiar, and easier language to program in, allowing the writing of NLP algorithms in a style much closer to that of conventional functional programming languages.

## References

Avinash, (2020). 4 Best NLP Libraries for Node.js and JavaScript. *Dialogflow*. https://www.kommunicate.io/blog/nlp-libraries-node-javascript/.

Brownlee, J. (2017). What Is Natural Language Processing?, *Deep Learning for Natural Language Processing*. https://machinelearningmastery.com/natural-language-processing/.

de Kok, D. & Brouwer, H. (2011). *Natural Language Processing for the Working Programmer*. https://www.researchgate.net/publication/259572969_Draft_Natural_Language_Processing_for_the_Working_Programmer.

Khanfor, A. & Yang, Y. (2017). An Overview of Practical Impacts of Functional Programming, *24th Asia-Pacific Software Engineering Conference Workshops*. https://www.researchgate.net/publication/323714122.

Krill, P. (2023). C++ still shining in language popularity index. *InfoWorld*. https://www.infoworld.com/article/3687174/c-still-shining-in-language-popularity-index.html.

Kristensen, J. T. & Kaarsgaard, R. & Thomsen, M. K. (2022). Jeopardy: An Invertible Functional Programming Language, *34th Symposium on Implementation and Application of Functional Languages*, DOI: 10.48550/arXiv.2209.02422.

Ljunglof, P. (2002). *Functional programming and NLP*. https://publications.lib.chalmers.se/records/fulltext/local_10778.pdf.

Shaikh, M. (2023). Top 5 Programming Languages for AI and Natural Language Processing. *Insider News*. https://www.insidermonkey.com/blog/top-5-programming-languages-for-ai-and-natural-language-processing-1167657/.