An Algorithm for Detecting Sentence Validity

Charles DeGennaro

SUNY New Paltz, USA, degennac1@newpaltz.edu, https://orcid.org/0009-0002-1547-9432

Andrew McDonald

SUNY New Paltz, USA, mcdonala12@newpaltz.edu, https://orcid.org/0009-0009-8157-6661

Ashley Suchy

SUNY New Paltz, USA, suchya1@newpaltz.edu, https://orcid.org/0000-0002-9673-7570

Abstract: In this paper, we introduce an algorithm for determining the grammatical validity of a sentence. We take a similar approach as in (Preller, 2007) and (Lambek, Type Grammar Revisited, 1999) (Lambek, From word to sentence: a computational algebraic approach to grammar, 2008) by encoding the English words based on word type which we call *components*. A sentence can be described both algebraically and geometrically. Our algorithm generates the geometric portion called *underlinks* from the generalized reductions of the algebraic portion. Underlinks uniquely determine the reduction of the components leading to the empty string. This is the mathematical basis for determining if a sentence is valid. We also provide a proof for the algorithm's time complexity of $O(n^2)$ along with a Python implementation.

This paper is part of a bigger project based on (Coecke, Mehrnoosh, & Clarky, 2010) where we explore the combination of a sentence's grammar and meaning. This is done by combining two compact closed categories; pregroups represent the grammar of a sentence, and finite dimensional vector spaces describe the meaning of a sentence. Together one compact closed category is created, representing both aspects of the sentence.

Keywords: Algorithm, Language Processing, Pregroup Grammar

Citation: DeGennaro, C., McDonald, A., & Suchy, A. (2023). An Algorithm for Detecting Sentence Validity. In M. Shelley & V. Akerson (Eds.), *Proceedings of IConEST 2023-- International Conference on Engineering, Science and Technology* (pp. XXX-XXX), Las Vegas, NV, USA. ISTES Organization.

Introduction

The work got started by looking at (Coecke, Mehrnoosh, & Clarky, 2010) where the main objective is to combine category theory and pregroup grammars to look at the meaning and grammar of a sentence together. We then narrowed our scope to the grammatical portion of the paper, with a specific interest in checking sentence validity. With the ideas presented in Coecke, we set out to look for an algorithm that could determine sentence validity, as it was mentioned to be possible, but never shown. After reading through the work of

(Preller, 2007) and (Lambek, From word to sentence: a computational algebraic approach to grammar, 2008), the foundations of our algorithm were discovered. These ideas were expanded upon in this paper to verify and prove our algorithm works as intended. To be more inclusive to members outside of this field, we provide some important definitions and background in this paper to ease the knowledge gap. Next, we will discuss the construction of our algorithm, and prove its time complexity of $O(n^2)$. Examples of sentences accepted and rejected by our algorithm are presented, and the full code is available for others to download and modify. Some improvements to be made in the future are mentioned, which are not yet present in our current version of the algorithm.

Definitions

In this section, we present definitions essential to the rest of this paper.

A *basic type* is the definite meaning of a word. For example, "John" is a subject, so when encoding "John" into component form, its basic type would be the symbol used to represent a subject, π. (Technically, John is a third person subject, so the actual symbol used is π_3) Note: Words can be broken into multiple basic types, as well as multiple different encodings based on their context in a given sentence. The symbols of basic types used in this paper and their meanings are described in Table 1.

Adjoints are used to denote valid positions of elements within a sentence. More generally, left adjoints and right adjoints (represented with a superscript "l" and "r" respectively) are defined for every basic type, *a*, and the following reduction rules for each hold:

$$a^l a \to 1 \qquad aa^r \to 1$$

Transitions allow a basic type to transform into another basic type, as described by the transition table provided in Table 3.

A *reduction* combines two elements into a unit element, essentially removing both elements from the sentence. For example, $ss^r \to 1$ is a reduction.

Components are symbols that form an encoding for words. Components are broken up into two parts: base (basic type) and precedence (adjoint). For example, the component x^r consists of the base "x", and the precedence "r". Furthermore, we convert the superscript "r" and "l" into the integer values of 1 and −1 respectively in the algorithm. If a component has no adjoint, this is represented with a 0 and the basic type is represented without a superscript. The program follows the reduction rules of $x^l x \to 1$ and $xx^r \to 1$.

A *dictionary* is a data structure where items can be accessed using a specific key.

An *underlink* is a geometric representation of a valid pairing. Underlinks are drawn under a given string of components to visually show the valid structure of a given sentence. In other words, underlinks connect components that reduce to 1. For example,

Figure 1. Example of an underlink

A full listing of basic types and type assignments can be found in Chapters 31 and 32 of (Lambek, From word to sentence: a computational algebraic approach to grammar, 2008). Here we present a small subset of the basic types and type assignments that we will use in this paper.

Table 1. Basic types used in this paper

π - Subject
π_1 - First Person Singular Subject
π_3 - Third Person Singular Subject
$\hat{\pi_3}$ - Pseudo-Subject
s - Declarative Sentence (Statement)
s_1 - Statement in the Present Tense
$\bar{s}$ - Indirect Statement
i - Infinitive of Intransitive Verb
j - Infinitive of Complete Verb Phrase
o - Direct Object
$\hat{o}$ - Pseudo-Object
n_1 - Count Noun

This paper utilizes the dictionary described by Table 2, which converts a word to its component form. Note that one may find many different valid component forms for a given word, based on its context in a sentence (for example, likes has a second form in parenthesis). This property will be mentioned briefly in the conclusion.

Table 2. Sample dictionary

Tom - π_3
$John$ - π_3
$Marie$ - π_3
I - π_1
him - o
she - π_3
$will$ - $\pi_1^r s_1 j^l$
$come$ - i
$doesn't$ - $\pi_3^r s o^l$
$matters$ - $\pi_3^r s o^l$
$matter$ - o
not - $o o^l$
see - $\pi^r s o^l \hat{\pi_3}$
$likes$ - $\pi^r s_1 o^l \hat{\pi_3}$, $(\pi_3^r s \pi^l)$
a - $\hat{\pi_3}^r o n_1^l$
$book$ - n_1
$watch$ - n_1
$which$ - $n_1^r n_1 \hat{o} \bar{s}^l$
$detests$ - $\pi^r \bar{s} \hat{o}^r$
$.$ - s^r

Table 3 provides a list of the valid transitions of basic types used in this paper. A more extensive list is provided in (Lambek, From word to sentence: a computational algebraic approach to grammar, 2008). This list is used in Algorithm 2 (The Match Algorithm) to determine if two components can form a valid pair.

Table 3. Valid Transitions

$$\begin{aligned}
\pi &\to \pi \\
\pi_1 &\to \pi_1,\ \pi \\
\pi_3 &\to \pi_3,\ \pi \\
\hat{\pi_3} &\to \hat{\pi_3},\ \pi_3,\ \pi \\
s &\to s \\
s_1 &\to s_1,\ s \\
\bar{s} &\to \bar{s},\ \pi_3,\ \pi \\
i &\to i,\ j \\
j &\to j \\
o &\to o \\
\hat{o} &\to \hat{o},\ o \\
n_1 &\to n_1
\end{aligned}$$

The Algorithm

Algorithm 1 (The Sentence Validity Algorithm) is a recursive algorithm that determines whether an English sentence is grammatically correct. To check validity, we start with an English sentence. This paper focuses on the present tense, with plans to include more sentence types in the future. Given a sentence, we encode each word into its components using the dictionary provided in Table 2. We refer to this list of components as the component sentence. Each component is then given an index from 0 to n − 1, where n is the length of the component sentence. Once the component sentence is generated, we check if a valid match exists for a given right component. We can find the locations of right components through Theorem 2, with the first iteration of our algorithm running from the trivial fact that the last component in a sentence must be a right component, and with the criteria for a valid match being described in Lemma 2. If a corresponding left component does exist, an underlink connects the two paired components, and the problem is recursively split into two new subsections, referred to as the outer and inner subsections. This naming convention comes from the nature of one subsection being outside, and the other inside the new underlink. This property can be found in more detail in Lemma 1.

If we look at Figure 2 for an example of the outer and inner subsections, we can choose the indexes 2 and 9 to be the pair found in a given iteration of the algorithm. The indexes from 0 to 1 would produce the outer subsection, and the indexes 3 to 8 would produce the inner subsection. If the algorithm successfully determines that each component has a valid pair, then the sentence is grammatically correct, and the geometric view of underlinks can be generated. If one component cannot find a valid pair, then the entire sentence is grammatically incorrect, and underlinks cannot be created. Note that the parenthesis used in each example are only used as a visual aid to show the breakup of which components are part of each word, they serve no function pragmatically and are not present within our program.

For our purposes, the geometric view of components is purely used as a visual aid to display why a given sentence is grammatically correct. For a more detailed explanation of the properties observed with the geometric structure, we encourage the reader to look into (Preller, 2007), and for a more mathematically rigorous exploration, (Coecke, Mehrnoosh, & Clarky, 2010). In Figure 2, the top row is the English sentence, the second row is the components making up each word, the third row is the index labels for each component, and the bottom row is the underlinks.

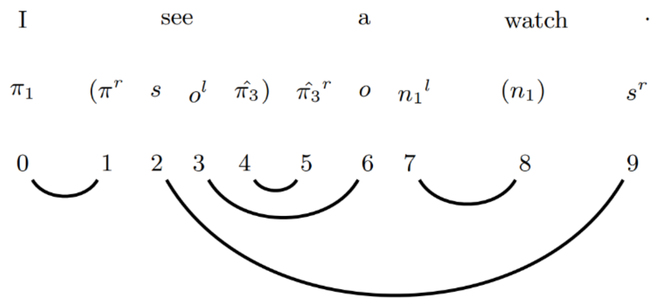

Figure 2. Underlink structure of a valid sentence

Our algorithm is based on a fact stated in (Preller, 2007), which we describe as Lemma 1, that if the location of a given right component is known, then its matching left component can be found. With the location of the last component in the string being a known right component, we work through the string backward, matching the current end component of our subsection to its matching left component.

Note that there is a pre-processing step done that rules out any odd length list of components before we get to Algorithm 1, as an odd number of components guarantees that one component will not find a valid pair.

Theorem 1. Algorithm 1 can be made to run in polynomial time, more specifically $O(n^2)$.
Proof. The following recurrence relation describes the runtime of Algorithm 1,

$$T(n) = \begin{cases} 0, & n = 0 \\ T(n-2) + \frac{n}{2}, & n > 0 \end{cases}$$

where n is the length of the component sentence.

For simplicity, the constant terms are omitted from the second case of the above function definition. These constant terms come from the Match Algorithm, with Steps 4 and 5 taking the longest to complete due to the transition table lookup and check.

Now we present the proof of the runtime in detail:

$$
\begin{aligned}
T(n) &= T(n-2) + \frac{n}{2} \\
&= T(n-4) + \frac{n-2}{2} + \frac{n}{2} \\
&= T(n-6) + \frac{n-4}{2} + \frac{n-2}{2} + \frac{n}{2} \\
&\quad\vdots \\
&= \frac{0}{2} + \frac{2}{2} + \frac{4}{2} + \cdots + \frac{n-2}{2} + \frac{n}{2} \\
&= 0 + 1 + 2 + \cdots + \frac{n}{2} - 1 + \frac{n}{2} \\
&= \sum_{i=0}^{\frac{n}{2}} i \\
&= \frac{\frac{n}{2}(\frac{n}{2}+1)}{2} \\
&= \frac{n^2+2n}{8}
\end{aligned}
$$

We now summarize two lemmas from (Preller, 2007) that were essential in developing the algorithm. Lemma 1 establishes the idea of working backward in Algorithm 1 given that we know a right endpoint because s^r (which is a period) is always the rightmost component of a sentence. Lemma 2 guarantees us that the left endpoint of underlink will always come before the right endpoint and that the two components at the endpoints must be compatible, i.e., reduce to 1.

Lemma 1. If $R: s_1 \cdots s_n \Rightarrow 1$ is a transition, then the iterator of a right endpoint of a link is the successor of the iterator of its left endpoint. (Preller, 2007)

Lemma 2. If $\{i, k\} \in R$ *and* $i < k$, then the algebraic condition $s_i s_k \to 1$ implies that $s_i = a^z$ and $s_k = b^{(z+1)}$ for some integer z and appropriate basic types a, b. (Preller, 2007)

Now we present a new lemma and theorem that are the foundation of Algorithm 1. Lemma 3 is a crucial intermediary step in proving Theorem 2. If we know that a left component is at position i and its corresponding right component is at position k, this allows us to infer the location of the adjacent component at $i-1$. This component must be either a right component within the range of 0 to $i-2$, or a left component within the range of $k+1$ to $n-1$. This is due to the inability of underlinks to intersect, as explained in (Preller, 2007).

Lemma 3. For two pairs of components $\{s_i, s_k\}$ and $\{s_{i-1}, s_r\}$, if both s_i and s_{i-1} are left components, then the position of s_r is greater than the position of s_k.

Proof. By definition we know $i < k$ and $i-1 < r$. Since $i-1 < i < k$ we know $r > k$ because underlinks can't cross. If r is anywhere from $i+1, \cdots k-1$ we would break that property.

A visual example of Lemma 3 can be seen in Figure 2, where labeling indexes $i = 4$, $i - 1 = 3$, $k = 5$ we see that the right component of $i - 1$ must be from $6 \cdots 9$ and it happens to be $r = 6$ in this example.

Theorem 2 shows that when processing the components from right to left if we consider a point where we just found a left component at i then $i - 1$ must be a right component. This idea can be seen below, where all indices in blue have been paired already, $i - 1$ in red is the index we are considering, and all indices in black may or may not have been paired.

$$0 \cdots (i-2)\ {\color{red}(i-1)}\ {\color{blue}i}\ (i+1) \cdots (k-1)\ {\color{blue}k \cdots n-1}$$

Theorem 2. Assume all components from s_{k+1} to s_{n-1} have already been paired. If a pair (s_i, s_k) exists where s_i is the left component, then s_{i-1} must be a right component.

Proof. Assume s_{i-1} is a left component. By Lemma 3, the corresponding right component of s_{i-1} must be somewhere to the right of s_k. However, based on the assumption declared in the theorem, there are no valid right components to be paired with, since every component is already paired. Therefore, s_{i-1} must be a right component.

Algorithm 1 The Sentence Validity Algorithm

Input: An even-length ordered list of components *components*, a starting index *start*, and an ending index *end*.

Output: True if a sentence portion is valid, False otherwise.

Steps:

1. If $end - start \leq 0$, return True
2. Let $index = end$-1
3. If $components[index]$ and $components[end]$ are a valid pair (determined via **Algorithm 2**), skip to Step 5, otherwise continue to Step 4.
4. If index $\geq start$, go back to Step 3 and subtract the value of index by 2, otherwise return False.
5. Recursively call **Algorithm 1** with $components = components$, $start = start$, and $end = index - 1$. Save this result as *outer*.
6. Recursively call **Algorithm 1** with $components = components$, $start = index + 1$, and end=end-1. Save this result as *inner*.
7. Return True if both *outer* and *inner* are True, otherwise return False.

Algorithm 2 Match Algorithm

Input: Two components *left* and *right*.

Output: True if these components form a valid pair, False otherwise.

Steps:

1. Check that the precedence of *left* is exactly one less than the precedence of *right*. If not, return False.
2. Check if the base of *left* and *right* are the same base. If True, return True, otherwise, continue to Step 3.
3. If *right* has a precedence of 0, continue to Step 4. Otherwise, *left* must have a precedence of 0, and skip to Step 5.
4. Check the transition table for the base of *right*. If any of the transitions are the same base as *left*, return True, otherwise return False.
5. Check the transition table for the base of *left*. If any of the transitions are the same base as *right*, return True, otherwise return False.

The Implementation

In this section, we explain some of the main components of our program, which implements the ideas discussed in this paper. The full code can be found at (DeGennaro, 2023). Below we go over the imports in the main file, which are all original code stored in separate files for easy understanding of the program.

Basic:

— A format to store each basic type as a unique integer id, accessible through the basic types common name. Ex: Basic.PI = 0, Basic.PI1 = 1, etc. The basic types used are accounted for in Table 1.

transitions:

— Used by the Match Algorithm to determine if a base can be transitioned to another base. These transitions are described in Table 3.

dictionary:

— Used in "sentence_to_components()" to convert each English word into its component form, as described in Table 2.

display_enums:

— Converts the integer form of a basic type back into a String representation for viewing as an output.

get_raw_pairs:

— Used after the Sentence Validity Algorithm runs, it outputs the set of index pairs of the matched components.

draw_underlinks:

— Takes in extra information we can extract from the running of the algorithm and uses it to display the underlinks in the terminal output.

Before reading the code, it is important to understand some of the data structures in place. To store our components, which are comprised of a base and a precedence, we create a tuple of the form (base, precedence). The base is the integer id discussed above (enum), and the precedence is an integer discussed in the definition of components previously. To draw the underlinks, the algorithm keeps track of the current recursion depth, which allows us to draw underlinks in the terminal that do not cross. Some underlinks can be drawn closer together, but this would require extra processing. Since the main goal of the algorithm is to determine sentence validity, the underlinks were added as a visual aid, and are not the priority output of the algorithm.

Some Examples

This section will present some example sentences to help the reader. We demonstrate a few examples of valid and invalid sentences based on our algorithm. Diagrams without underlinks are invalid, as underlinks can only be drawn on valid sentences. Figure 6 shows an invalid sentence and Figure 7 shows the valid correction of Figure 6. Figure 6 will be detected as invalid at the pre-processing stage since it has an odd number of components. Figure 8 shows another invalid sentence, and it will be detected as invalid when we attempt to find a match for the component s^r in the first step.

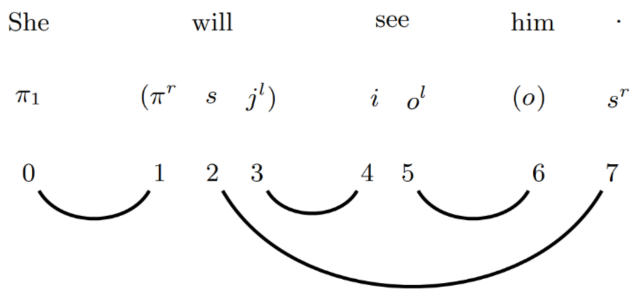

Figure 3. Valid sentence

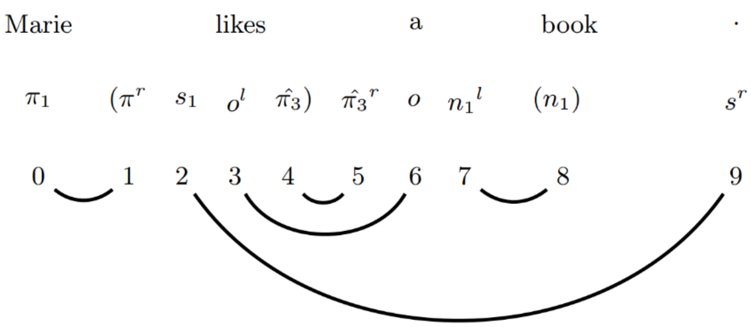

Figure 4. Valid sentence

John likes a book which Marie detests .

$\pi_3 \quad (\pi^r \; s_1 \; o^l \; \hat{\pi_3}) \quad \hat{\pi_3}^r \; o \; n_1{}^l \quad (n_1) \quad n_1^r \; n_1 \; \hat{o} \; \bar{s}^l \quad (\pi_3) \quad \pi^r \; \bar{s} \; \hat{o}^r \quad (s^r)$

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17

Figure 5. Valid sentence

Tom matters not .

$\pi_3 \quad (\pi_3{}^r \; s \; o^l) \quad o \; o^l \quad (s^r)$

0 1 2 3 4 5 6

Figure 6. Invalid sentence

Tom doesn't matter .

$\pi_3 \quad (\pi_3{}^r \; s \; o^l) \quad o \quad (s^r)$

0 1 2 3 4 5

Figure 7. Corrected sentence (Sorry to all Tom's)

John likes Marie .

$\pi_3 \quad (\pi_3{}^r \; s_1 \; o^l \; \hat{\pi_3}) \quad \pi_3 \quad (s^r)$

0 1 2 3 4 5 6

Figure 8. Invalid sentence due to encoding of "likes"

John likes Marie .

$\pi_3 \quad (\pi_3{}^r \; s \; \pi^l) \quad \pi_3 \quad (s^r)$

0 1 2 3 4 5

Figure 9. Corrected sentence with alternate "likes" encoding

Conclusion

We hope the algorithm presented here can be expanded upon to accept more valid sentences than the current solution poses. One example of a sentence that our current model cannot accept is “John likes Marie .”, even though this is a grammatically correct sentence. This is a result of the limit in our current transition table, where each word can only have one component encoding (see Figure 8). Using an alternative encoding of likes, “$\pi_3^r s \pi^l$”, we can use this in our program and see a successful acceptance of the sentence (see Figure 9).

A possible method to solve this problem would be to change the dictionary in Table 2 to store lists of component encodings, instead of a single component encoding for each word. After that, we would generate all permutations of the component sentences based on each word with multiple encodings. If any of the component sentences is determined as valid, then the sentence can be determined as grammatically correct. This would create an exponential increase in runtime but seems like a valid first step to solve this problem. Future work may look into this method and ways to optimize it.

It would be worth exploring another intriguing feature, which is the ability to correct improper sentences. This is demonstrated in Figures 6 and 7. Fixing a sentence may have to include a model for finding the semantic structure of the sentence, which is not possible with our current system.

References

Coecke, B., Mehrnoosh, S., & Clarky, S. (2010). Mathematical Foundations for a Compositional Distributional Model of Meaning. *CoRR*.

DeGennaro, C. (2023). GitHub code. *https://github.com/ItBeCharlie/SentenceValidity*.

Lambek, J. (1999). Type grammar revisited. *Logical Aspects of Computational Linguistics*.

Lambek, J. (2008). From word to sentence: a computational algebraic approach to grammar.

Preller, A. (2007). Toward Discourse Representation via Pregroup Grammars. *Journal of Logic, Langauge, and Information*, 173-194.