Drexel University
Department of Computer Science

Parsing Natural Language using CCG to Generate Executable Software Tests

Thesis Report

Research Committee:
Dr. Colin S. Gordon
Dr. Geoffrey Mainland

Submitted By:
Nikhil Parakh (np657)

Submitted On:
Friday, June 9, 2023

Table Of Contents

Introduction

This research was focused on developing a parser that parses Natural Language (English) using Combinatory Categorial Grammar (CCG). The parsed English Language would be used to generate executable software tests.

The existing machine learning implementations for parsing natural language are non-modular and difficult to repair. The other implementations use some type of probabilities and work with Context-Free Grammar. We did not have enough statistics to be able to work with probabilities with the current CCG lexicon as it does not include enough software and testing related terms.

We have applied classic ideas from linguistics to parse natural language. While there are existing parsers that do the work, they aren't very efficient. We used shared concurrency model in the parsing algorithm for more efficient results.

Literature Review

CCG

Combinatory Categorial Grammer (CCG) is a grammar formalism used to associate words in a sentence with a category during parsing. These categories are functions that define the type and directionality of its arguments and also defines the type of its results.

The "result leftmost" is a notation in which a rightward-combining functor over a domain β into a range α are written α/β, while the corresponding leftward combining functor is written $\alpha\backslash\beta$. α and β may themselves be function categories. For example, a transitive verb is a function from (object) NPs into predicates—that is, into functions from (subject) NPs into S:
(1) likes := (S\NP)/NP
(2) Forward Application: (>) X/Y Y => X
(3) Backward Application: (<) Y X\Y => X (Mark Steedman, 1996)

An example of this would be:

```
                Harry            likes            movies
                 (NP)          (S\NP)/NP           (NP)
                               —------------------------------>
                                                (S\NP)
              —---------------------------------------------------------<
```

(S)

There are various other rules but for simplicity only the forward-backward application rules have been demonstrated here.

Other Implementations

Currently there are multiple implementations of CCG Parsers. Mark Steedman in his book “*The Syntactic Process”* mentions how CCGs are directly compatible with the binary-branching bottom algorithms. Some of the applications of these algorithms are (1) Shift-Reduce Parser, and (2) Cocke-Kasami-Younger (CKY) Parser.

1) Shift-Reduce Parser

   A Shift-Reduce parser uses Stack data structure to parse the English sentence. Putting something into the stack is called Shifting and the application of combinatory rule on the top components of the stack is called reducing. We iterate over the words in the sentence and put the single items (words) to this stack and if it is possible to apply some combinatory rule to the top components of the stack, we reduce it and replace the components with the result until no more reductions can be performed. Then, we continue putting the words to the stack.

   When we have reached to the end of the sentence and all the words have been shifted, if the content of the stack is the starting symbol, then we can conclude that the parse is accepted. If not, a different parse should be used or the sentence is not grammatically correct.

2) CKY Parser

   The CKY Parser uses a chart to parse the English Language.
   We first initialize n x n lower triangular matrix for a sentence with n words. Let’s call it *chart*. Chart[0][0].....chart[0][n] are the word categories. Each step above the previous checks for possibility of a span. Like chart[1][i].....chart[1][n-1] checks for the possibility of 2-span words and at the end the algorithm checks if chart[n][0] is the starting symbol or not. If it is, then the parse is valid and sentence is grammatically correct. If not, the sentence is invalid.

Different Parallelization Strategies

Mark Johnson in his paper talks about 3 different ways of making the parsing algorithm (CKY parsing algorithm) run parallelly. They are:

1) Map-Reduce Algorithm

   A map reduce algorithm works best with data points that are independent of each other. Words in natural language sentence heavily depend on it's adjacent words for a meaning and a proper categorization. A map reduce algorithm also requires a key value pairing which isn't really useful while parsing natural language. This is why we did not move forward with this algorithm.

2) Symmetric Multi-Processor using OpenMP

   SMP makes use of the different cores of a CPU that are connected via a high speed bus to a shared memory. SMP is typically controlled by softwares like OpenMP. OpenMP are C++ based programs with pragmas that indicate which loops should be parallelized. (Mark Johnson)

3) Using a Graphics Processing Unit (GPU)

   Mark experimented different approaches to GPU parsing using CUDA. CUDA (Compute Unified Device Architecture) is a parallel computing platform and programming model that enables dramatic increases in computing performance by harnessing the power of the GPU (Nvidia.com). He found out that direct translation of algorithms used in standard CPU to use in GPU showed negative results. The algorithm needed to be changed in order to take full advantage of the GPU (Mark Johnson). Since we did not have access to a CUDA GPU, we did not move forward with this method.

Mark's experiment was based on Probabilistic Context Free Grammars (PCFG). Our focus is on Non-Probabilistic CCGs. Which is why using the same implementation and/or relying heavily on their research and result was not a good idea.

After careful consideration we decided to move forward with a shared state concurrency model with a custom ConcurrentSet data structure which builds implements ConcurrentHashMap.

Implementation

The parser is implemented primarily in Java using Maven build tool. The are 4 major packages in this implementation which are described below:

Categories

Categories package consis of **Category** class and it's subclasses. Category class is the representation of CCG Categories. It's extends into **AtomicCategory** and **SlashCategory** classes.

Slash Categories are composed of an Argument and a Result. Given the argument, the category results in the Result. **Right Slash** Types look for Argument in the Right of the category. For example: (S/NP) + NP results in S. **Left Slash** Types look for Argument in the Left of the Category. For example N + ((S/NP)\N) results in (S/NP)

AtomicCategory represents Categories that do not take any arguments. For example N, S, Det and so on. All the Atomic Categories are implemented using singleton pattern.

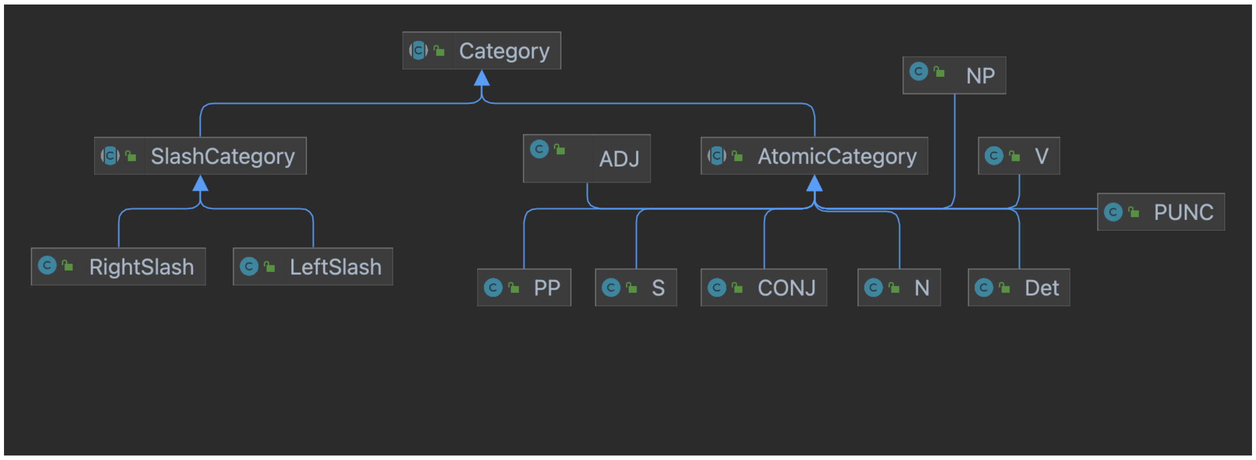

Figure 1: Class Diagram of Categories Package

Language

The Language package consists of a **Lexicon** class which is a data structure of the type HashMap<String, Set<Category>>. It provides methods to interact with the lexicon entries. The method initialize entries populates the data structure with the lexicons provided using Lexicon Populator package.

The **Grammar** class consists of a combine method that defines how two ParseTrees can be combined. It uses the rules defined in Rules enum.

The **Rules** enum provides the different rules that the Grammar class uses to combine two different ParseTrees. These are the rules currently implemented in the parser

- Forward Application
- Backward Application
- Forward Composition
- Backward Composition
- Forward Crossing Composition
- Backward Crossing Composition

In addition to this, the parser also type raises categories of type N to NP.

Lexicon Populator

The Lexicon Populator is used to populate the Lexicon data structure with the lexicon provided in the concrete implementations of the **LexiconPopulator** abstract class. It has adaptor design pattern implemented to convert Lexicon category types to the categories that are provided by Categories package.

Populator1 and **Populator2** uses lexicons from CCG Bank. Due to the size of the lexicons, the populator had to be divided into two.

CustomPopulator has lexicons hard coded into it. It was used during initial development phases and for testing purposes.

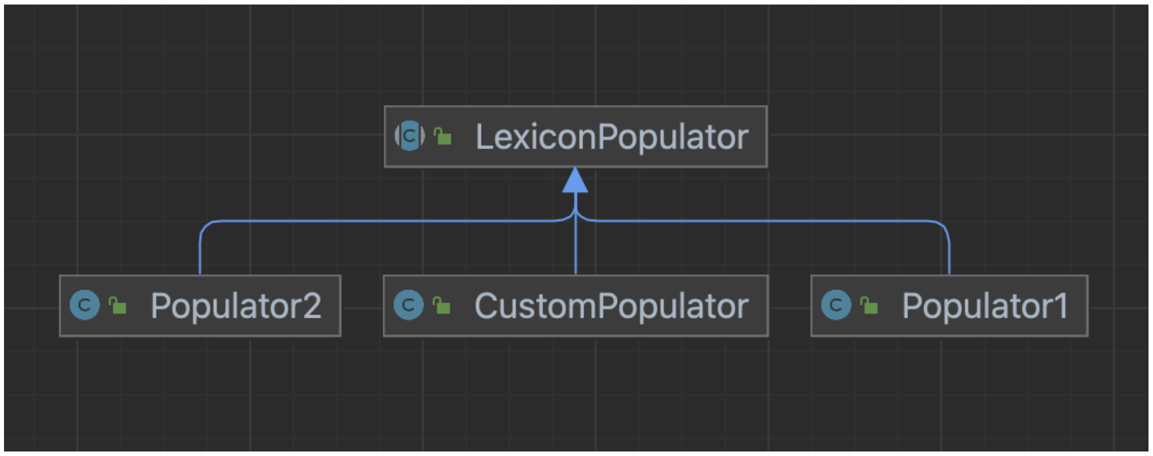

Figure 2: Class Diagram of Lexicon Populator Package

Parser

This package is the heart of the parser implementation. It consists of a Parser class that parses the sentence using CYK algorithm, builds a CYK chart and returns the chart to the client.

A cell refers to an element in the chart. This is represented by **ConcurrentSet<ParseTree>**

The chart refers to the data structure used in CYK chart parsing algorithm. It is represented by **List<List<ConcurrentSet<ParseTree>>>**. In other words it is a list of rows, and each row is a list of cell.

It uses some other classes for help.

A **Pair<K,V>** object is just a key value of the types K, V respectively. It allows method to access and update the key and value.

A **ParseTree** is the smallest unit in the algorithm. It is an entity that represents a Category c, a pair that combined to form this category c (represented by a left child and a right child), and a sentenceFragment representing what part of the input sentence it is.

Here is an example of what printing a ParseTree would look like (only showing a few Parses):

Sentence: “I like eating apples”

Parse 1:

```
1) S (I like eating apples)
  2) NP (I)
     3) null
     3) null
   2) S\NP (like eating apples)
     3) (S\NP)/NP (like)
       4) null
       4) null
     3) NP (eating apples)
       4) N/N (eating)
         5) null
         5) null
       4) N (apples)
         5) null
         5) null
```

Parse 2:

```
  1) N (I like eating apples)
   2) N/N (I like eating)
     3) N/N (I)
       4) null
```

```
        4) null
      3) N/N (like eating)
        4) N/N (like)
          5) null
          5) null
        4) N/N (eating)
          5) null
          5) null
    2) N (apples)
      3) null
      3) null
```

The **CombineThread** class inherits from the java.Lang.Thread class. This class has a run method that takes in two cells from the chart and combines each ParseTree from one cell with all the ParseTrees from another cell and returns the result of the combination.

A **ConcurrentSet<T>** is a custom set implementation that is built upon **ConcurrentHashMap<T, Boolean>**. This was needed to have a thread safe Set which is not provided by the Java Collections Library and also to remove the use of concurrency locks. Initially we had implemented concurrency locks on the Set data structure provided by the Collections Library. However, the locks were not shared among the threads which caused further issues. Hence, we needed to create something that we could use. The ConcurrentSet interface allows users to add a value whis is put into the parent class (ConcurrentHashMap) as a key with the value set to True always. The iterator method returns the iterator for the list of keys in the parent class.

The parse method in the **Parser** class builds the chart from top to the bottom. The topmost row (index: 0) of the chart is the set of possible categories of the words in the sentence. The last row (index: sentence.length() - 1) is the root of the chart. The root either contains a ParseTree with category S or other categories. A ParseTree with category S tells that the input sentence is grammatically correct, a null value means that the sentence is not grammatically correct, and anything else means that the sentence is either a fragment of a larger sentence or is not grammatically correct.

The parser first splits the input sentence into tokens and gets the lexical categories using getCategoriesFromLexicon() method. This methods also reports all the words in the sentence that aren't found in the lexicon. In this case, the parse fails and it exits the parser, displaying the words not found in the lexicon. The parser then initializes the chart using buildChartCells() method. This method creates n-1 rows in the chart. The 0th row has 1 cell (the root) and the (n-1)th row has n cells. n is the number of words in the sentence.

The algorithm then iterates through different lengths of spans, ranging from 2 to the number of sentence categories plus 1. For each span length, it further iterates through all possible starting positions in the sentence. At each start position and span length, a new ConcurrentSet is created to hold the parse trees.

Each CombineThread operates on shared data structures, such as the chart cells and the result ConcurrentSet. As the algorithm iterates over possible break positions within a span, it creates a new CombineThread object for each break position. These CombineThread objects share access to the chart cells and the result ConcurrentSet.

By running multiple CombineThread objects simultaneously, the algorithm takes advantage of shared state concurrency. Each CombineThread accesses and modifies shared data structures concurrently, allowing for parallel execution of the parse tree combination process. This approach enhanced the performance of the algorithm by utilizing multiple processor cores and speeding up the parsing process. To mitigate issues related to thread safety and data races, a .join() method is used to ensure that all CombineThread objects finish their execution before accessing and updating shared data structures. Afterward, the chart cell at position [span-1][start] is updated with the combined parse trees. The set of threads is then cleared, preparing for the next iteration.

Once the parsing algorithm finished, the parse() method returns the entire chart, which contains the parse trees for different span lengths and start positions.

Issues Encountered and Solutions

During the implementation there were multiple issues that we encountered. Initial concurrency implementation had some unresolved bugs, including thread freezing and infinite looping. This was also due to locks not being shared across threads. We solved this by replacing locks with thread safe Set (ConcurrentSet). With this implementation, parsing failed for sentences longer than 14 words, it was due to the cell capacity limit. This limitation also prevented the testing of concurrency with longer sentences. We solved this by implementing Eisner Normal Form Constraint while combining two ParseTrees.

Eisner Normal Form Constraint 1 states:
The output X|α of forward composition >Bn>0 can not be the primary input to forward application or composition >Bm≥0. The output of <Bn>0 cannot be the primary input to <Bm≥0. This can be implemented by a ternary feature HE∈{>Bn,<Bn,∅} and chart items of the form X, HE, i, j where HE => Bn (or<Bn) if X was produced by the corresponding composition rule (for any n>0) and ∅ otherwise. (Julia Hockenmaier, Yonatan Bisk)

We also encountered the issue of duplicate parse trees. In the root cell of the chart, multiple ParseTree objects had the exact same structure. This was because the lexicon had multiple entiries for the same categories for a word. This was solved partly by removing the duplicates manually. But given the amount of data we had, we implemented a workaround for it. Everytime a word and it's categories are accessed from the lexicon, the string representation of the Category is added to a hasmap. If the key already exists in the map, we don't add that Category Object again. Another way to to fix this could be overriding the hashmethod of the ParseTree so it uses the string representation of it. This will also prevent duplicate keys in a hash map data structure.

Results

We ran the following 18 sentences in 3 different versions of the parser. First was a basic parser with no concurrency implemented. Second, was the parser implemented with concurrency but unoptimized. Third was the parser with concurrency, ConcurrentSet, and Eisner Normal Form Implemented. Table 1 shows the results for each implementation:

Impl 1: Parser without Concurrency
Impl 2: Concurrent Parser without Optimization
Impl 3: Concurrent Parser with Optimization

	Sentences	Impl1 (ms)	Impl2 (ms)	Impl3 (ms)
1.	one is less than five	30	32	20
2.	one is even	3	3	2
3.	every number is even	9	2	2
4.	every number is a number	3	3	3
5.	one is equal to five	8	9	3
6.	one is less than or equal to five	678	530	15
7.	one is passing	1	1	1
8.	one is not passing	1	2	1
9.	any float that is greater than or equal to one and less than five is not passing	-	-	1384
10.	any float that is greater than or equal to one and less than five is passing	-	-	193
11.	any float is not passing	14	2	1
12.	one is an exception	2	1	1
13.	one throws an exception	0	1	1
14.	I booked a flight to PHILADELPHIA from CALIFORNIA	4	18	5
15.	I want five apples	0	0	1
16.	I want to drive a big yellow car on bumpy road today	2677	2203	33
17.	I want to drive a big yellow car on the narrow bumpy road today	-	-	87
18.	I want to drive a big yellow car on the narrow bumpy road to home today	-	-	1192
	Total Time	-	-	2951

Table 1: Parsing Time of Different Parsing Implementations

Here we can see how in sharter sentences there isn't much difference between the various implementations. But, as the sentences grow longer, the first two parsers (Impl1 and Impl2) either fail to parse, or takes too long to process. In some cases the IDE ran out of memory and crashed too, throwing a OutOfMemory error. Overall we can see how Impl 3 perform better. It also successfully handles longer sentences. The total runtime to process 18 sentences for the third parser implementation was 2951ms which is almost the same time the first 2 implementations take to parse sentence no.16. The third implementation takes only 33 ms for the same. This is 98.77% faster than Impl 1 and 98.51% faster than Impl 2.

This shows how using a shared state concurrency model, thread safe Set data structure (ConcurrentSet) and Eisner Normal Form Constraint really improves parser efficiency by a lot.

Future Work

Software testing is a crucial part of software development. It helps with identifying defects early on and improves overall software quality. Software developers often try to stay away from testing. They believe it is not essential and a waste of time and money. This results in software products that are not reliable and can lead to unsatisfactory customer experiences.

It is easier to write sentences in English than it is to write tests with wide coverage. Automating the test generation process will take the hassle away make it easier for developers to frequently test their code. Current parsing algorithms use a very exponential algorithm and that along with generation of test code will increase the over all runtime and space complexity.

We can leverage the concept of Lambda Calculus to generate semantics that can be used to generate enecutable software test cases. We hava a parser that is efficient and fast. The Parse Trees generated during parsing phase can be used to generate semantics.

Conclusion

In this research, we developed a Combinatory Categorial Grammar (CCG) parser to parse English sentences. Unlike existing machine learning implementations, our parser is modular, repairable, and efficient.

We applied classic linguistic ideas to parse natural language and used a shared concurrency model to improve efficiency. We explored different parallelization strategies such as Map-Reduce, Symmetric Multi-Processor (SMP), and GPU parsing but found the shared state concurrency model to be the most effective for our non-probabilistic CCG implementation. Testing our parser on a set of 18 sentences, we observed significant improvements in performance with the optimized concurrent parser compared to the basic and unoptimized versions.

Overall, our research contributes a modular and efficient CCG parser for natural language parsing, with potential applications in generating executable software tests. Further enhancements and optimizations can be explored to expand its capabilities and improve parsing accuracy.

References

1. Mark Steedman, 1996. A Very Short Introduction to CCG. In Combinatory Grammars, page 1.

2. Mark Johnson. Parsing in Parallel on Multiple Cores and GPUs.

3. Julia Hockenmaier, Yonatan Bisk. Normal-form parsing for Combinatory Categorial Grammars with generalized composition and type-raising.

Github Links:

4. Mike Lewis. EasyCCG. https://github.com/mikelewis0/easyccg

5. Marc Herschel. Java Implementation of CKY Algorithm. https://github.com/mynttt/CYK-algorithm

6. OpenNLP. OpenCCG. https://github.com/OpenCCG/openccg