Fulati Aizihaer

Laurel Perkins

Ling 185A

12/13/2024

Final Project Report

In computational linguistics, parsing plays an important role as it connects the structure of language to its meaning and usage. Parsing involves analyzing sentences to understand the different components within the sentence and grammatical structure of the sentences based on a set of rules defined by the grammar. This process is crucial for computers to split sentences into smaller and meaningful parts to determine how these parts interact, which makes it easier for machines to understand and process human language. Parsing can be used for things like Natural Language Processing (NLP) tasks such as text analysis, language translation, and other computational linguistics applications.

To further explore parsing, we analyze strings derived from Context-Free Grammars (CFGs), which is a system used to represent the rules and structure of a language. A CFG is a four-tuple (N, Σ, I, R), where:

- N is a finite set of nonterminal symbols representing phrases
- Σ is a finite set of terminal symbols representing actual words in the language
- $I \subseteq N$ is the set of initial nonterminal symbols
- $R \subseteq (N \times (N \cup \Sigma)^*)$ is a finite set of rules within the grammar

Context-Free Grammars are used to build sentences using smaller units and then using that build and describe complex sentence structures as they can effectively capture the recursive

structures of the human language while making sure that it is simple and efficient enough to process. For example:

- S → NP VP: is a sentence (S) that is composed of a noun phrase (NP) followed by a Verb Phrase (VP).
- NP → D N: is a noun phrase (NP) that is composed of a determiner (D) followed by a noun (N).

CFGs can describe many syntactic patterns, such as nested structures and recursive structures, which are common in human language. However, when faced with longer and more complex sentences, CFG has limitations and difficulty handling those sentences.

In this project, we will mainly focus on three different parsing schemas for CFGs: bottom-up parsing, top-down parsing, and left-corner parsing. All three of these schemas show how machines can analyze sentence structure and approach parsing in a way that is similar to how humans think about and process sentences. Through implementing these three schemas, we will be able to further understand how parsing schemas work, where each of them excel, and their own limitations when it comes to parsing sentences with specific structure. As the names suggest, each of these schemas approach analyzing sentences differently, as they are being supported by different transition functions that implement the transition steps of the parsing for that specific schema.

A parsing schema is a structured method to break down and analyze a sequence of symbols, such as words and phrases using rules given in a grammar. For this project, I will focus on transition-based parsing, which is a method that constructs a given sentence through transitions, step by step. Transition-based parsing builds the structure of a sentence through

operations such as shifting words onto a stack, removing them, or combining them to form larger units. A transition-based parsing schemas consist of the following components:

- **Starting Configuration** specifies how the parser should begin, based on a given grammar and input sequence.
- **Goal Configuration** specifies the condition under which parsing is considered complete.
- **Transition Relation on Configuration** specifies the rules for moving from one configuration to another.

These are the information needed to implement a specific parsing schema and correctly identify all of the possible parses for a given sentence from a grammar through transition-based parsing.

I will first get started with the bottom-up parsing schema. The bottomUp function in my code implements the bottom-up parsing schema for a given CFG and an input sentence. Its starting configuration is an empty stack and contains the input, along with goal configuration where the stack contains only the start symbol of the grammar and an empty input, which means that all the words from the input sentence have been parsed. The bottom-up parsing uses two operations in order to correctly parse through a grammar, and these operations are shift and reduce. The shift operation moves a word from the input onto the stack if it correctly matches the right-hand side of a rule. The reduce operation combines elements on the stack into a larger structure using a nonterminal rule.

As for the top-down parsing, it is represented by the topDown function which also takes in a cfg and an input sentence. Its starting configuration is a stack that contains the start symbol of the grammar wrapped in the “NoBar” and also contains the input sentence. The goal configuration is when the stack is empty and the input is also empty. The top-down parsing also

uses two operations in order to correctly parse through a grammar, and these operations are predict and match. The predict operation adds the right-hand side of a rule to the stack if the left-hand side matches the current symbol on the stack. The match operation removes a word from the input if it matches the current symbol on the stack.

Finally, we have the Left-Corner parsing that combines features of both top-down and bottom-up parsing. This approach avoids some of the inefficiencies of top-down or bottom-up schemas. The starting configuration of the left-corner parser is similar to that of the top-down parser, where the stack that contains the start symbol of the grammar is now wrapped in the "Bar" and also contains the input sentence. The goal configuration is when the stack and input are both empty. The left-corner parser begins parsing with the leftmost part of the input and step by step builds the structure. It contains four operations which are shiftLC, matchLC, predictLC, and connectLC, where shift, match, and predict are similar methods from top-down and bottom-up parsers. In the left-corner parser, the shiftLC adds the matched symbol "NoBar (lhs rule)" to the front of the stack instead of the back which was how it is done in the regular shift method. The matchLC removes a word from the input if it matches the current symbol on the stack, where the top of the stack can be a "Bar" instead of that in the regular match function that only works with "NoBar nt" symbols. The predictLC predicts what should follow based on grammar rules, but it now also handles "Bar" along with "NoBar". The connectLC method is what's unique to the left-corner parser, where it combines a "NoBar" symbol and a "Bar" symbol from the stack elements to form larger structures when the left corner of a rule matches the current input.

In order to process all three of these parsing schemas, I also have a parser function that implements the parsing process by applying a series of transition functions. The parsing schema

provides the starting configuration, goal configuration and the transition steps. It applies a series of transition functions like shift, reduce, predict, match, and connectLC and implements specific transitions for different parsing schemas iteratively until a goal configuration is reached or no further transitions are possible. Then it returns a list of all of the successful parses that are possible by executing the given transition steps according to the given rules.

Additionally, I also have a helper function for the parser-base function which I named “initialRowParser”. This helper function initializes the parsing process by defining the starting configuration and makes sure to properly handle the transitions. Additionally, this helper function makes sure to include the initial NoTransition step of the parser through a boolean operation to check if the current step is the initial step of the parsing. These functions work together to manage the stack and input during parsing, making sure that the process follows the rules of the grammar.

After I finished implementing all of my parsing schemas, I then began to work on testing them. In order to effectively test my parsing schemas and make sure that they work on wide range of sentences and grammars, I tested them across four different grammars including the two provided grammars (cfg12 and cfg4) and two custom grammars (cfgCustom1 and cfgCustom2) which included different sentence structures. These tests made sure that the parsers could handle different linguistic patterns, sentence lengths and sentence complexities. The sentences I tested includes:

1. “watches spies with telescope” - cfg12: A simple sentence to test each individual schema's ability to handle straightforward syntactic structures.
2. “the baby saw the boy” - cfg4: Another simple and straightforward sentence structure with a different grammar.

3. "the actor the boy the baby saw met won" - cfg4: A deeply nested and complex sentence designed to challenge each schema's handling for recursive features.
4. "the girl ate paella in Spain" - cfgCustom1: A sentence that includes prepositional phrases.
5. "the aardvark the panda chased slept" - cfgCustom2: A sentence that includes embedded clauses.

Each of the parsing schemas were able to successfully identify possible parses for sentences above. However, parsing times differed based on the sentence complexity and which parsing schema was used. The bottom-up parsing was able to output all the parses efficiently and quickly even for the sentence #3, which was a complex sentence that had recursive structure. As for top-down parser, it was also able to output all parses quickly, however there was one catch, which was to remove all of the left recursion rules in the given grammar, where the grammar has rules of the form $A \rightarrow AB_1 \ldots B_n$, which made the top-down parser go into an infinite predict loop. To fix this specific challenge, I had to remove the individual left recursive rules that was in the form above in both cfg12 and cfg4, which was (NTRule NP [NP,PP]) and (NTRule VP [VP,PP]) from cfg12, and rule (NTRule NP [NP,POSS,N]) from cfg4. Once I removed these left recursive rules, top-down parsing worked fine without an infinite loop. As for the leftCorner parser, it was able to successfully identify all the possible parses for a given sentence, however one thing was the time it took to complete. For sentences 1, 2, 4, 5, left-corner parser was able to quickly identify the different parses for individual sentences, however for sentence 4, it took longer than the rest because of it being a deeply nested and complex sentence as well as the multiple center-embedding. From the results above, I noticed the following. The time taken to parse a sentence often correlates with the length and the complexity of the sentence, as well as

the depth of the stack. This suggests that stack depth is a critical factor in parsing efficiently. Other factors such as the number of possible parse paths also play a crucial role.

All three parsing schemas could handle left-branching, right-branching, and center-embedded structures, but they all differ in how well they manage each type. The bottom-up parsing schema and left-corner is efficient at handling left-branching. As for the top-down parsing schema, it struggles with handling left-branching because it sometimes encounters infinite predict loops because of the left recursion in the grammar. As for the left-corner parsing schema, it struggles with long complex multiple center-embedding sentences. Where with a complex and long sentence, it usually takes a longer period of time to get the results of the parsing.

After analyzing all three of the parsing schemas, in order to make parsers more efficient, we can take a look at how humans backtrack when reading sentences, as shown in Part 3 of Assignment #7. The eye-tracking experiments reveal that humans modify their understanding of a sentence when they read a new word that conflicts with their earlier assumptions of the sentence. For example, a bottom-up parser processes words step by step, by attaching the phrase like "the baby" to the verb "watched" as its object. Then it only realizes that this was wrong when it encounters a conflicting word like "cried," making it to backtrack later. On the other hand, a top-down parser predicts the sentence structure early on, such as assuming "the baby that won" is a part of a relative clause. And when this prediction fails, it backtracks at an earlier location in the sentence. From this, parsers can be made more human-like by recording their decision points and revisiting them when there is any error, which saves time by avoiding extra retries.

In addition to human-like features, we can also attach probabilities to the production rules of a Context-Free Grammar (CFG) and transform it into a Probabilistic Context-Free Grammar (PCFG). In a PCFG, each rule is assigned its own probability that reflects how likely it will be used during the generation of a sentence (Collins). Using Probabilistic Context-Free Grammar improves the parsing accuracy by evaluating the likelihood of different parses and then selecting the most probable one.

In conclusion, this project provided me with the opportunity to implement the three fundamental parsing schemas: bottom-up, top-down, and left-corner parsing, which allowed me to evaluate each individual parsing on different grammars and sentence structures. Each parser had its own strengths and limitations, where they approached parsing differently through different transitions. Bottom-up parsing was good at handling left-branching structures, top-down parsing struggled with left recursive issues, and left-corner parsing struggled with complex and longer sentence structures. These results highlight the trade-off of these different parsing schemas and the challenges of processing human language and grammar.

Citations:

Collins, Michael. "Probabilistic Context-Free Grammars." *Natural Language Processing* course notes, Columbia University, 2011, www.cs.columbia.edu/~mcollins/courses/nlp2011/notes/pcfgs.pdf.