Tina Fotouhi

Dr. Perkins

Ling 185A

10 December 2023

Final Project Option1

Introduction

Context-Free Grammar (CFG) parsing represents a fundamental aspect of computational linguistics, playing a crucial role in our broader understanding of human language. One of the core functions of CFG parsing is its ability to systematically model and analyze the syntax of natural languages. This is particularly important in the realm of natural language processing (NLP) applications, where understanding the structure and rules of language is key. By providing a structured framework, CFG parsing allows us to explore the intricate patterns and rules that govern language structures. Tools and systems in computational linguistics, such as machine translation and speech recognition also rely heavily on the syntactic analysis provided by CFG parsing. By understanding and implementing CFG parsing, these applications can accurately process and generate human language, making interactions with technology more natural and intuitive.

In addition, CFG parsing is instrumental in comparing the syntactic structures of different languages. Such comparative analysis is invaluable in identifying universal aspects of grammar and understanding the shared features across various linguistic systems. This not only contributes to linguistic theory but also has practical applications, such as in the development of more efficient and accurate language processing tools that can work across different languages.

Another significant aspect of CFG parsing is its capacity to handle and resolve ambiguities inherent in natural language, which can often lead to different interpretations. CFG parsing provides a systematic way to identify these ambiguities and analyze various syntactic structures in different ways, enabling the selection of the most appropriate structure for a given context. This analytical capability is especially beneficial in applications like speech recognition systems, language generation systems, and other NLP tools where discerning the correct interpretation of language is essential.

Body

A CFG is a specific type of formal grammar that is used in the analysis of natural languages, particularly in parsing and understanding the syntactic structure of languages.
A CFG is composed of a set of rules that define how symbols in a language can be combined to form valid strings or sentences. These rules are used to parse sentences, breaking them down into their constituent parts to analyze their syntactic structure. CFG is particularly powerful for its ability to handle recursive structures, which are common in natural languages. There are various approaches to CFG parsing, including bottom-up parsing, top-down parsing, and left-corner parsing, which are the focus of this project. Each of these parsing strategies employs a specific algorithm to analyze and construct the parse tree of a given input sentence according to the rules of a CFG.

The first parsing strategy that I implemented is the bottom-up parser, which operates with a shift-reduce algorithm to process an input string from left to right, one symbol at a time. It maintains two primary data structures: a stack and an input buffer. In the shift operation, the parser moves the next input (terminal) symbol from the input buffer to the top of the stack. In my code, this is implemented by taking the leftmost terminal symbol in the input list and looking for

terminal rules in the context free grammar that translate that terminal symbol to a non-terminal symbol. This non-terminal symbol assigns that input word with a grammatical categorization.

My reduce function uses a helper function findMatchingRules to create a list of all possible sequences (extractedLists) from the stack that might match the right-hand side of any production rule. It does this by progressively taking different amounts of elements from the top of the stack (from 1 element to the entire stack length). The function then removes these symbols from the stack and pushes the non-terminal from the left-hand side of the rule onto the stack. This step effectively replaces the sequence of symbols on the stack with the corresponding non-terminal, signifying that a syntactic structure represented by the rule has been recognized. A combination of these shift and reduce steps parses the string from the "bottom" (a full input string and an empty stack) up, reaching the goal configuration of a single non-terminal on the stack representing the full sentence (S), and an empty input list representing a fully parsed string.

The next parsing strategy that I implemented is the top-down parser. Top-down parsing begins with the start symbol, predicting what next non-terminals on the stack could be based on the rules of the grammar, then matching terminal symbols to those predictions until the whole sentence is parsed. It starts with the highest level of the parse tree and expands from the "top down." My predict function is implemented by looking for rules in the grammar whose left-hand side matches the top element of the stack. It then replaces that top element with the right-hand side symbols of the matching rule, allowing the parse to proceed with the anticipation that this grammar rule will be applied next based on the current state of the parsing process. The match function looks for terminal rules where the left-hand side matches the non-terminal at the top of the stack and the right-hand side matches the current input symbol. When a match is found, the corresponding symbols are removed from the stack and input. This step is important as it checks

if the parser's prediction about what it should see next in the input aligns with what is actually there. The top-down parsing algorithm utilizes a combination of these two steps to eventually produce both an empty stack list and an empty input list, signifying a fully matched non-terminal stack and a fully parsed input string.

The last parsing strategy I implemented is the left-corner parser. At a high level, a left-corner parser is a type of syntax parser that combines elements of both top-down and bottom-up parsing strategies. It starts parsing from the leftmost symbol of a production rule (the "left corner"), and then uses this information to guide the rest of the parsing process. This approach allows the parser to effectively handle recursive structures, particularly left-recursive elements, which can be challenging for pure top-down parsers. By beginning with the leftmost symbols and progressively building up the parse structure while also incorporating top-down predictions, the left-corner parser balances the immediate data-driven approach of bottom-up parsers with the predictive nature of top-down parsers. My implementation incorporates four functions in order to conduct a left-corner parse: shiftLC, matchLC, predictLC, and matchLC. It also interprets non-terminal symbols as stack elements such as NoBar and Bar, in order to handle the intricacies of approaching a parse from both top-down and bottom-up perspectives. NoBar signifies that the non-terminal is expected next in the input according to the top-down parsing strategy, while Bar indicates that the non-terminal has been partially processed in a bottom-up manner and that the parser is in the process of completing this recognition.

My shiftLC function is a specialized version of the regular shift function, placing a NoBar marker on a non-terminal symbol before pushing it on the stack. This indicates that the shifted element is not yet completely processed, and will need further completion by subsequent steps. The matchLC function not only checks if the current input symbol matches the expected

terminal in the rule like the regular match function, but also ensures this match aligns with the context set by the left-corner parsing strategy. It looks for terminal rules where the left-hand side corresponds to the Bar-marked non-terminal on the stack, indicating a partial match in progress, and only performs a match in such a case. Next, the predictLC function considers the current state of the stack to determine which non-terminals might come next in the input. It finds rules where the right-hand side begins with the non-terminal at the top of the stack, to ensure that the predictions are contextually aligned with the part of the input already parsed. Based on the identified rules, predictLC replaces that symbol with any additional symbols from the rule's right-hand side (adding Bar to indicate partial completion of these structures), followed by the left-hand symbol (NoBar). Finally, the connectLC function serves as the bridge between the parser's predictions and the actual input. It does this by identifying rules where the right-hand side matches the current non-terminal on top of the stack, and the left-hand side corresponds to the second non-terminal on the stack. For rules with a single symbol on the right-hand side, that symbol is simply removed from the stack, as the prediction has been fully validated. Otherwise, the remaining right-hand symbols are added to the rest of the stack, labeled as Bar to indicate that they are part of an ongoing structure that needs to be completed. Ultimately, these processing steps work in conjunction to produce an empty stack and empty input list, indicating that the string has been fully parsed.

Discussion

My parsers are able to produce all of the different parses for all of my test cases, which include sentences varying from 2 to 12 words long. I initially started out testing my functions on just cfg12, but then realized the importance of testing on different CFGs once some of the sentences that were working with cfg12 were not working with cfg4. I eventually realized that

this was because my reduce function was only equipped to process rules that had exactly two non-terminals on their right-hand side, such as NTRule NP [NP,VP], but not rules with one (e.g. NTRule VP [V]) or more (e.g. NTRule NP [D,N,ORC]) right-hand side non-terminals. This was preventing sentences that incorporated such rules from being parsed. In order to resolve this problem, I added a helper function to test different numbers of non-terminal symbols to find matching rules that had varying numbers of non-terminals on their right-hand side. I also realized that this was not a problem for the top-down parser, since the predict function only looks at the first right-hand symbol in order to predict possible matching rules. I also created an additional cfg from the section 7 handout, which can produce sentences like “the girl met the boy in Spain” and “Mary ate paella in Spain.” This helped me check whether my parsers were able to work with a cfg other than the ones I had been building them with.

My top-down parser struggles with left-branching (left-recursive) rules, such as NP → NP PP, because these rules cause the parser to repeatedly try to resolve the same non-terminal, leading to infinite recursion. This issue occurs as the parser continuously predicts the same non-terminal it's trying to expand, making no progress in consuming the input string and potentially causing an endless parsing process. This led to issues with stack overflow when testing sentences on CFGs that included these types of rules, which required me to remove them in order to successfully produce the parse sequence.

All three of my parsers are able to process right-branching, left-branching, and center-embedding sentences of varying lengths. It is important to note that I added a helper function to my main parser function to prevent the size of the stack from expanding beyond a certain length. This was needed in order to be able to conduct left-corner parses on some longer sentences, and larger context free grammars with a large number of rules. Without the limit, the

stack size would sometimes grow much too large for the machine to be able to handle, resulting in stack overflow or extremely long processing times. This limit provided the parser with a way to discern that it was going down a bad path that would not ultimately lead to a successful parse, allowing it to move on to attempt a different path before overwhelming the machine.

An issue that arose from this added implementation was finding a stack limit that would be compatible with different parsing strategies. This compatibility issue was mainly between the bottom-up and left-corner parsers, because bottom-up parsers need a large stack size in order to push every element on the stack when processing certain types of sentences. These include right-branching sentences like "John met the boy that saw the actor that won the award." Meanwhile, the left-corner parser requires a small stack limit in order to process this sentence without taking an extremely long time. In fact, I had to increase the stack limit to 12 for the bottom-up parse to work, but this stack size was too large for the left-corner parser to be able to output anything. I ended up setting the stack limit to 8, as that number seemed to accommodate a large variety of different types of sentences. However, I recognize that this number would have to increase in order to successfully produce a bottom-up parse for some longer sentences. An interesting observation that I made was that I was able to bring the stack limit down to as low as 2 and still produce successful left-corner parses on some sentences, even relatively long ones. This reflects the stack behavior we observed in the parses we did by hand in class, which showed how the left-corner parser continuously expands and shrinks without ever growing very large when it comes to left-branching and right-branching structures.

I also tested my parsers on some "unacceptable" sentences, such as the center-embedding sentence "the actor the boy the baby saw met won." These were all successful, since they were still grammatical according to the rules of the CFG. This type of sentence ran quickly with the

top-down parser but took longer with the left-corner parser. This was expected, as the left-corner parser experiences an increase in stack size with center-embedding sentences. A stack limit of 8 ended up working for them both.

I also tested some left-branching structures, such as “Mary ‘s boss ‘s baby won.” As expected, this type of sentence was processed very quickly by the bottom-up parser, since it did not lead to an increase in the stack size. However, I noticed a slowdown with the top-down parser, which does experience an increase in stack size with left-branching sentences. Initially, my stack limit was actually set too low for the top-down parser to be able to process this sentence, and had to be increased to 7 in order to work. I added print statements into my functions in order to examine the stack size on different parsers and different sentences, which proved to be very helpful in drawing these conclusions.

In computational linguistics, the left-corner parser notably resembles the way our brains are believed to process language. This parser uniquely merges the methods of top-down and bottom-up approaches, reflecting the dynamic and flexible nature of human linguistic comprehension. It begins its process with the leftmost components of language input, skillfully integrating them with broader grammatical structures. This technique closely mirrors human language processing, where the brain balances immediate linguistic input with established contextual understanding. The left-corner parser's ability to simultaneously predict and adapt to unfolding linguistic patterns mirrors the human brain's sophisticated approach to parsing language. The similar parsing abilities of left-corner parsers and human brains is also evident in our capability to comprehend increasingly lengthy right and left-embedded sentences and still find them coherent. However, when a center-embedded sentence extends beyond a certain point,

it becomes increasingly more challenging for us to grasp its meaning, leading us to consider it as incomprehensible or "unacceptable”.

Something that could help my parsers make improved and more human-like guesses would be to have them make more contextual decisions. Enhancing parsers with the ability to consider broader context, like semantic information, could make their guesses more accurate. In addition, it could be helpful to implement some sort of adaptive parsing strategy that could switch between top-down and bottom-up parsing based on the specific syntactic structures it encounters. This would more closely replicate the flexibility of the human brain as it processes language. Another way to improve things would be to incorporate probabilities in the CFG. In a probabilistic CFG, a parser would be able to make decisions based on how likely a certain rule or structure is in the language. This would also help reduce issues with ambiguity, as the parser would be able to choose the parse with the highest probability. Implementing a probabilistic CFG would probably involve training it on a large set of data in a language in order for it to be able to accurately estimate the probabilities of different rules. This would be an interesting possibility to explore further to expand the scope of this project.

Bibliography

Daniel Gildea, Giorgio Satta; Synchronous Context-Free Grammars and Optimal Parsing

Strategies. *Computational Linguistics* 2016; 42 (2): 207–243. Doi: https://doi.org/10.1162/COLI_a_00246

"Overview of NLP: Issues and Strategies." *NLP1*, https://tildesites.bowdoin.edu/~allen/nlp/nlp1.html#:~:text=A%20context%2Dfree%20grammar%20(CFG,reading%20from%20left%20to%20right.