Arthur Kim

Professor Perkins

LING 185A

16 March 2022

In 1982, David Marr wrote in *The Philosophy and the Approach* that to understand how a machine processes information, the three levels at which a machine carries out such a task must be understood: computation theory, representation and algorithm, and hardware implementation (Marr 25). Given that the human mind comprehends language by taking audiovisual input and converting it mentally to some meaningful internal representation, the human mind serves as an example of the aforementioned machine. Computational linguistics aims to understand how the mind comprehends language on all three levels described by Marr, and this Haskell project and paper aims to focus particularly on the representation and algorithm level. The guidelines for this project clearly address the computation theory and associated problem: given context-free grammars, there are three pertinent methods of parsing such a grammar. These high-level methods are bottom-up, top-down, and left-corner parsing. With the aim of describing context-free grammar parsing at the representation and algorithm level, this project utilizes a given Haskell framework to represent context-free grammars and implements each type of parsing via Haskell algorithms and code.

The implementation such that functions are implemented in order of which the project guidelines assigned the implementation of said functions. For the code of the `parser` function in particular, note that an input sequence can be speculatively parsed in multiple different paths, with only some of them leading to valid parses. The helper functions implemented before `parser` aim to break down this speculative parsing into manageable smaller tasks. For example, `stepBranch` explores potential ways to parse a particular input by following only one hypothetical parse sequence and attempting one transition step (SHIFT, REDUCE, etc.). `stepBranchAll` applies all the possible

transition steps on one hypothetical path taken to parse the input. `splitBranch` extends the behavior of `stepBranchAll` to apply to every single speculative parsing path currently being explored. `genBranches` recursively runs on itself, generating speculative paths to parse the input until no new paths are generated, suggesting that parsing is complete and ending the recursive loop. It implements the main behavior of `parser` and returns the pertinent info needed to be returned by `parser`. The `topDown` and `leftCorner` functions are implemented by following the same structure given in the provided `bottomUp` function, merely tweaking the starting and goal configurations to comply with the high-level logic of top-down and left-corner parsing. Finally, `removeDupe` cleans up some quirks I seemed to find while implementing left-corner parsing regarding duplicate parses being returned. I can't seem to pinpoint the issue, but the validity of parses seems to be fine. `removeDupe` merely prevents duplicate parses from being returned. Overall, the main bulk of functionality of the program comes from the top-level `topDown`, `bottomUp`, and `leftCorner` functions. The other functions certainly provide important functionality, but they don't help give an idea of the overall functionality and structure of the code. As such, here are relevant test cases that give a clue as to how the program should be utilized:

```
topDown cfg12 (words "watches spies with telescopes")
bottomUp cfg12 (words "watches spies with telescopes")
leftCorner cfg12 (words "watches spies with telescopes")
```

Yes, I realize these are derived from the project guidelines, but the usage cases give a clue as to the program should be utilized, and the output does enough to tell you that the displayed behavior is working properly since the VP phrase "watches spies with telescopes" can be understood as watching people who have telescopes or as committing the of watching spies by using telescopes.

Overall, as long as a proper CFG is combined that can generate a given input, my implementation should be able to parse it properly. It should return every proper parse, as far as I can

tell. The test cases that I borrowed from the project guidelines shows proof that the top-down, bottom-up, and left-corner implementations of parsing all catch each valid interpretation of the VP. Due to time constraints, I don't have any good test cases here that can lead to multiple valid parses, but I do think I proved the robustness of my implementations via trial by fire. Using the sentence "While John watched the baby that won cried" from the garden-path analysis section of Homework 7, I modified cfg4 to include the words "watched" and "cried" in the terminal words section and added accompanying terminal symbol rules. Given the computational difficulty of garden-path sentences, if each parsing implementation can output the proper parse, it suggests that my implementations are robust. Indeed, here are test cases:

```
topDown cfg_test (words "while John watched the baby that won cried")
bottomUp cfg_test (words "while John watched the baby that won cried")
leftCorner cfg_test (words "while John watched the baby that won cried")
```

Unfortunately, one linguistic question that my implementation can't seem to properly answer is the effect of left-branching structures, right-branching structures, and center-branching structures on computational difficulty for bottom-up, top-down and left-corner parsers. I haven't bothered to include test cases here because they don't hold validity. The bug for left-corner parsing I mentioned prior (that needed to be rectified by the `removeDupe` function) is causing left-corner parsing to take longer than it computationally should. As such, in the garden-path test case above, the `leftCorner` parser actually took noticeably longest, even though left-corner parsers are expected to have less difficulty parsing the right-branching structure (the NP "the baby that won cried") compared to bottom-up parsers. Also it is worth consideration that the development environment I used is extremely powerful; for the bottom-up parser to output far quicker than the left-corner parser suggests that it is most definitely the bug with left-corner parsing (which again, only affects the speed of computation, not the validity of output) causing these unexpected results. As such, my Haskell algorithmic model of bottom-

up, top-down, and left-corner parsers can't accurately help answer the question of branching structures' effects on each parser. Finally, note that due to my development environment, the difference in bottom-up and top-down parsing becomes imperceptible, despite bottom-up parsing being expected to take longer than top-down parsing for the garden-path test case.

On the topic of right-branching structures, my implementation is able to handle all three types of embedding structures:

```
bottomUp cfg_test (words "Mary 's boss 's baby met the actor")
topDown cfg_test (words "Mary 's boss 's baby met the actor")
leftCorner cfg_test (words "Mary 's boss 's baby met the actor")
-- right corner parsing already shown possible prior
bottomUp cfg_test (words "the actor the baby met watched the boy")
topDown cfg_test (words "the actor the baby met watched the boy")
leftCorner cfg_test (words "the actor the baby met watched the boy")
```

Note that the last test case takes particularly long to run because of the aforementioned bug with left-corner parsing that I can't pinpoint; I promise the results are valid! Again, I'd like to emphasize that my development environment is so fast that the difference in length of time it takes for a bottom-up versus top-down parser appears mostly negligible. That being said, the bugged left-corner parser took particularly long for center-embedded structures, even compared to just the right-branching garden-path test case. As such, these results might have some validity in serving as proof that left-corner parsers do still see an increase in difficulty parsing center-embedded structures.

It should be noted that my parsers make brute-force guesses as to the path taken in parsing a given input. As long as a transition step is possible at any certain point, my parsers will take them and explore the viability of the path given by taking that step until further steps simply become impossible and the goal configuration hasn't been reached. In other words, my parsers take a fully naive approach to parsing inputs. I can't say I know of any particular search techniques that could be implemented since I haven't learned beam search, A-star search, etc., but if there exists a function delta that assigns

values to transitions, the context-free grammars used in the project can become semiring-weighted context-free grammars. As such, we can weight transitions or assign probabilities to them such that the parsers only return the analysis of an input that maximizes weight or probability.

In the end, my Haskell program aimed to tackle the representation and algorithm level underlying the high level concepts of how bottom-up, top-down, and left-corner parsers can use the rules of a context-free grammar to determine whether said grammar can generate a given input and demonstrate how that input would be parsed into grammatical categories. The Haskell representation of context-free grammars was already given as a CFG data type taking grammatical categories and terminal symbols (words) within the context-free grammar. The process of parsing in and of itself is similar regardless of using top-down, bottom-up, or left-corner processing. Given a set of possible transition steps based on which type of parsing is being used, a parser uses the grammar rules given by a context-free grammar to determine whether an input can be parsed from the parser's starting configuration into the parser's goal configuration. The parser broke down the tasks of applying a potential transition step and exploring various potential parsing paths that could lead to a valid parse by using helper functions. The functions were organized into the hierarchy of applying a step to generate further speculative parsing paths to a current path, then applying all applicable steps to the current path, then applying all steps to all current paths being explored by the parser, and then finally using an overarching function to recursively generate paths and explore them until no more paths are generated, at which point any valid parses remaining are output. Despite my implementations being able to handle ambiguous parses (such as the VP "watches spies with telescopes"), garden-path sentences, and all three types of embedded structures, there are a number of hypothetical improvements that could have been made to extend my project in new directions and increase my implementation's linguistic validity. For one, left-corner parsing is bugged such that although the parses returned are correct, it takes far longer than it should. Fixing this issue and adding some way to time how long it takes each parser to parse a given input should help concretely demonstrate how different types of embedded structures lead

to increased computational difficulty for different parsers. Furthermore, having a semiring-weighted context-free grammar along with updating my implementations to handle the associated values for each rule in the aforementioned grammar would help my implementation better resemble human cognition. I assume that humans don't just brute-force word sequences in their head (the internal human parser doesn't take a naive approach), and as such, this is one area where my implementations don't reflect human cognition. A probabilistic or weighted context-free grammar and updated parser implementations would better reflect human cognition since the parsers would make "smarter" decisions based on transition values.