Ling 185A – Final Project Report

Carson Kim

Winter 2021

1 Introduction

For my final project, I decided to try a Haskell implementation of the transition-based parsing schemas we saw in Week 7. The three parsers — bottom-up, top-down, and left-corner — are all declared as the functions `buParser`, `tdParser`, and `lcParser` in `FinalProject.hs`, respectively. They were all written to accept rules and configurations from context-free grammars in Chomsky-Normal form. This means that we can parse sentences whose rules have a maximum of two nonterminal symbols on the right-hand side.

2 Implementation

The first section of code, labled as `cfg setup`, includes declarations for a few small grammars used for both testing and demonstrating the parsers. The `RewriteRule` and `Cat` data types were incorporated from Assignment 6, as well as the cfg containing the NP/VP ambiguity in the sentence "watches spies with telescopes". I modified the code to omit generalized semirings, as they are not relevant parts of the data structure for the purposes of this assignment. As a result, the CFGs in my code are a 4-tuple containing a list of nonterminals, a list of terminals, a nonterminal start symbol, and a list of rules.`cfg2` is modeled after the example sentences provided in the Week 7 handout for each parser. Of course, valid starting configurations are provided for each combination of parser and CFG they can be derived from.

The parsing implementation begins with some basic helper functions; they mainly operate on rules and configurations. In addition, there are the key functions `maxLoad` and `maxLoadMany` that take in a list (or a list of list) of configurations and obtains the size of the max memory load.

Bottom-up parser, of course, contains a shift and reduce function that do most of the work. Similarly, top-down parser contains a predict and match function. Lastly, the left corner parser contains its own versions of shift, predict, and match since they operate on `StackSymbol`s instead of vanilla nonterminals.

All three parsers take in virtually the same arguments and all output a list of list of configurations (since there are multiple "paths" that a parser can take). Each function accepts a list of `RewriteRule`s and a configuration of the form `([a], [b])`. You can start by entering something like

```
*FinalProject> paths = buParser (getRules cfg2) bu2
```

which stores the list of list of configurations into `paths`[1]:

```
*FinalProject> paths
[[([],["the","baby","saw","the","boy"]),
([D],["baby","saw","the","boy"]),
([D,N],["saw","the","boy"]),
([NP],["saw","the","boy"]),
([NP,V],["the","boy"]),
([NP,V,D],["boy"]),
([NP,V,D,N],[]),
([NP,V,NP],[]),
([NP,VP],[]),
([S],[])]]
```

You can then call `maxLoadMany paths`, which should output the size of the maximum memory load across the list of list of configurations.

```
*FinalProject> maxLoadMany paths
4
```

Since `maxLoad` just obtains the largest memory load for a list of configurations, you can do something like this[2]:

[1]Output has been reformatted for readability, of course.

[2]The use of the list operator "head" in these examples was intentionally avoided to prevent any unnecessary professorial rage.

```
*FinalProject> paths = tdParser (getRules cfg1) td1
*FinalProject> maxLoad (paths!!0)
2
*FinalProject> maxLoad (paths!!2)
3
```

Of course, you can call the top-down and left-corner parsers in a similar fashion by entering something like:

```
*FinalProject> lcParser (getRules cfg2) lc2
```

or

```
*FinalProject> tdParser (getRules cfg2) td2
```

You can pass in any starting configuration and it will output the remaining configurations for the parsed sentence. Passing in ”invalid” starting configurations for these parsers will output the empty list, since there are no paths to retrieve. Attempting to pass in a configuration that’s <u>not</u> of the type `Configuration (StackSymbol nt) t` into `lcParser` will result in a type error.

```
*FinalProject> tdParser (getRules cfg2) bu2
[]

*FinalProject> lcParser (getRules cfg2) bu2
<interactive>:160:26: error:
    * Couldn't match type `Cat' with `StackSymbol Cat'
```

3 Motivations and Challenges

I chose this as my project topic because thought it would be interesting to see how sentences are parsed programmatically. In addition, I wanted to explore the differences between the logic of a Haskell implementation and the heuristic tools I utilized when working on Assignment 7 by hand. I found most of the programming to be quite intuitive, with just some pattern matching needing to be done for the “shift, predict, connect, etc.” steps. The main challenge of this assignment was figuring out how to navigate the search space in the main parsing functions. The best approach I found was to simply obtain a list of all possible steps one could take from a given configuration and the resulting configuration for each of those steps based on every rule available to us in the grammar. After we obtain that list, we must simply iterate through and determine which of those results in a possible goal configuration and build up the list accordingly.

Another thing I learned from this assignment was how to implement simple heuristic shortcuts for my functions to run more optimally. For the top-down and left-corner parsers especially I encountered a problem where taking repeated predict steps would result in an unbounded number of configurations thus causing the program to run infinitely. Since some of the rules in the grammar I included are recursive, i.e. `NP (NP, PP)`, taking a predict step would result in an infinite number of configurations being generated. To prevent this, I first prioritized match / shift steps in these parsers, which guarantees the elimination or addition of only one symbol at a time. Secondly, I introduced a `bound` constraint that limits the number of symbols allowed for a given configuration (the bound can be anything within reason, but for my relatively small-scale grammars I just made it twice the length of the string in the initial input configuration).

4 Conclusion

Though challenging at times, this assignment proved to be quite rewarding and it certainly put my recursion and pattern-matching skills in Haskell to the test. In addition, it was neat to see how the helper functions all clicked together rather neatly in the overall design of the program. Future iterations of my program would likely expand the number of nonterminals allowed in the right-hand side of rules. If I did this, I would need to modify the `RewriteRule` data type to have the form `nt (nt, nt, nt)`, `nt (nt, nt, nt, nt)`, etc. or simply have each nonterminal associated with a list of nonterminals. Despite a few mental roadblocks here and there, I had quite a bit of fun working on this project.

Acknowledgments

Much gratitude to Professor Hunter and Yang Wang for their patience and assistance during office hours and throughout the quarter.