LING 185A (W19) Final Project

Haskell Implementation of Earley Parser

Seth Lasam (404644057)

The provided code implements a rudimentary Earley Parser in Haskell. Given a grammar and a string, the parser generates a set of Earley states to determine whether or not the grammar can generate that string. If it can, it will output either these states or the parse tree that they represent.

To run the parser, load EarleyParser.hs into ghci. A command to invoke the parser takes the form “earleyParse grammar string output hygiene”, where “grammar” is some recognizable grammar, “string” is your input string in quotes, output is either ParseTree or States, and hygiene is either Clean or Dirty. A grammar must be of the form [Rule (Symbol a b) [Symbol a b]]; the Rule and Symbol definitions are basically identical to the ones in previous homework assignments. There are 6 sample grammars at the bottom of the code, named grammar1, grammar2, …, grammar6. A string should be a list of Chars that can be generated by the grammar in that order. The output flag determines whether you want the parser to output all the states it went through to parse the string, or whether you just want to see the resulting parse tree. Put States for the latter, ParseTree for the former. Finally, the hygiene flag is only important if you chose States for the output flag. If you want to see *all* the generated states, even the ones that eventually lead to dead ends or otherwise aren’t productive, then put Dirty as the flag. If you want to see just the states that actually lead to a proper parse, put Clean as the flag. If you chose ParseTree. just put one at random because I don’t know how to implement optional args in Haskell. For instance, after loading EarleyParser.hs, an example of a proper command would be:

earleyParser grammar6 “a+a+a+a” States Clean

which would have it output a list of only the useful states that were generated in parsing that string with that grammar. If you want to use your own grammar, you can define it directly in the source code (at the bottom) or define a variable for it in ghci.

This implementation comes with some caveats. It assumes that the start rule of any grammar is the left-hand symbol of the first rule in its rule list. It can't accurately recognize grammars with any compound rules (i.e. a rule like **S -> A t B | B t A** which has 2 possible paths). Since any compound rules in a grammar can easily be separated into non-compound rules, this shouldn't be an issue, but just keep that in mind if you want to add in your own grammar for testing. It will hang on grammars that have no way of terminating a string, so don't try to parse with those unless you don't believe me. When it encounters a string generated by an ambiguous grammar, it will always favor left-grouping: for example, if you ran the command on the previous page with ParseTree instead of States, it would have outputted a tree that corresponds to (((a+a)+a)+a). This output won't change no matter how many times you run the command. Finally, it only either outputs a successful parse or nothing at all, so it can't show you parses that failed partway, or incorrect parses, or anything like that.

The parser is based on the algorithm described in the Wiki page for "Earley parser", to which I am eternally grateful. The definition of how it works, what an Earley state is, what the three Earley operations are, starting and goal condition, etc., are very clearly explained there, better than I ever could. Note that this algorithm is more suited to an imperative language which can actually do nested for-loops, so I had to change the strategy for a Haskell implementation. I ended up defining a new data type called a Chart, which consists of two lists, the former holding calculated results and the latter holding results that are expected to be processed at a later time. The parser first initializes a Chart by giving its latter list the initial Earley state, which serves as the seed for the rest of the parse tree. Then it uses the three operations to look at this list and expand it if necessary. At every step, it saves state information in the former list, and parent / child information for a parse tree in the state itself, since I defined an Earley state data type to be able to hold that information. Upon a "completion" operation, it moves the calculated result to the former list and looks at the remaining items in the latter list for further processing. If there aren't any and the proper goal conditions have been met, then it outputs the resulting parse. Otherwise it'll say that no parse is available. It'll also say so if it reaches a dead end from which no further parsing is possible, but the goal conditions haven't been met.

Some parts of the code come from stuff throughout the quarter. The data type definition for a Symbol is one, with the added member of 'D' to serve as a "dummy" nonterminal which makes seeding the parser easier. There's also the definition for a Rule, which was inspired by the RewriteRule definition from Assignment 6: instead of using the form "NontermRule nt (nt,nt) | TermRule nt t", it uses "Rule (Symbol a b) [Symbol a b]", where the first lone Symbol represents a nonterminal and the next list of symbols represents the symbols it can be rewritten with. And I also took inspiration from the same assignment's Tree definition to define MyTree, with the only change being that a MyTree Nonleaf need not be binary. Of note as well is the "displayTree" function for printing an instance of MyTree in a readable format, which is mostly an adaptation of the "drawTree" function from Haskell's Data.Tree module. And the majority of the Earley operations' implementations (scanning, predicting, completing), as well as the algorithms for fillChart and earleyParse, come from adapting the algorithms given for them on the Wiki page.

I decided on this project sometime after Assignment 8, where we had to implement bottom-up, top-down, and left-corner parsers manually. The tedious task (no offense) of writing out all those states by hand and figuring out the resulting parses really made me appreciate the ability of computers to automate that kind of thing, so thought it’d be cool to write a program to that end. I like to think I learned a lot about Haskell and functional programming from this project. This is the biggest and most involved program I’ve ever written using a purely functional language, and it was a major challenge considering my mostly imperative background. I also had no idea what Earley parsing even was before hearing about it from the suggested projects page, and I think learning about it was extremely interesting and insightful. If I ever get into writing compilers or parsers in the future, I at least have that under my belt. And I believe this project really drove home how complex the world of parsing, grammars, and language is. To think that this huge personal project was just one of a myriad of different ways to parse a string given a grammar is pretty fascinating. Also, I used to dread both functional languages and having to write compilers and parsers in previous classes, but I think this project definitely made me more open to both.