Guidelines and ideas for final mini-projects

You have a fair amount of freedom in choosing what to do for a final mini-project. It should not be particularly huge: something roughly the "size" of two weekly homework assignment is not a bad guide to keep in mind. The standard expectations are that you will submit some Haskell code and a write-up of what you have done, as an additional file in either PDF or plain text format. The write-up should do at least the following:

- Describe what your submitted code does and how to run it on a few illustrative test cases. (Be specific here: for example, give me something to copy and paste into ghci to get started.)
- Explain what parts of the code (if any) are imported from what we did during the quarter (either homework answers, or code provided or written in class), and what you added to it.
- Say something about what led you to choose to do what you did, and what you learned from it.

The mini-project will be graded with a number of dimensions in mind. *Not all of the following will be applicable to every project* (i.e. there will be many different ways to earn a good grade), but among the criteria I will be looking for are:

- Competent use of the central parts of Haskell that we have used in the course. (Showing that you've taught yourself a whole lot of additional Haskell is not going to earn much credit in itself.)
- Understanding core material from the course. (If you do something completely unrelated to the course content, it will be hard to demonstrate this.)

- Building grammars that correctly capture linguistically interesting patterns.
- Building on or extending things that we talked about in the course, perhaps in ways that connect them to things you've learned in other linguistics courses.
- Making connections between ideas from different parts of the course.

1 Matrices and FSAs

If you like matrix multiplication, there's a fun way to write matrix-based versions of many of the functions we wrote for working with FSAs.

Given a particular FSA with n states (ordered in some arbitrary way) and a particular symbol x, define M_x to be an n-by-n matrix of booleans such that the value at row i, position j is true iff there's a transition from the i'th state to the j'th state emitting the symbol x. In other words, M_x is a sort of "reachability matrix" which only takes into account transitions that emit the symbol x. To multiply boolean matrices, we do conjunction where we'd usually do multiplication and we do disjunction where we'd usually do addition. Then, given matrices M_x and M_y for two symbols x and y, it turns out that the matrix M_xM_y tells you something interesting about the string xy. (If you're not sure what, try it out on an example — it's quite amazing the first time you see it!) Lastly, think about the 1-by-n row vector which contains each state's I value (i.e. true iff it's an initial state), and the n-by-1 column vector which contains each state's F value.

Of course, all of this can be done with other semirings too.

(Can you see how forward and backward values fit in to this way of doing things?)

2 FSA-to-RegExp conversion

We implemented the algorithm for converting regular expressions to FSAs, to establish that every stringset that is the denotation of a regular expression is also generated by some FSA. There's also a procedure for going in the reverse direction, and establishing that every stringset generated by an FSA is the denotation of some regular expression. See e.g. Sipser (1997, pp.69–76).

For this you will need to introduce a type for FSAs with regular expressions on the transitions, perhaps something like this:

```
type GeneralizedAutomaton st sy = ([st], [sy], [st], [st], [(st, RegExp sy, st)])
```

(These are sometimes called *generalized finite-state automata* — but they have nothing (much) to do with the *generic finite-state automata* that we introduced in week 5!) Then the key part is in writing a function that takes one of these automata and produces a new one that's equivalent but has one less state.

3 Transition-based CFG parsing

Try a Haskell implementation of the transition-based parsing schemas for CFGs that we talked about in week 7.

One place to start here would be to define a type for configurations like this:

```
type Configuration a b = ([a],[b])
```


Then, given a CFG with nonterminals of type `nt` and terminals of type `t`, the configurations you'd use for bottom-up and top-down parsing would have the type `Configuration nt t`.

But for left-corner parsing, you'll need something extra like this:

```
data StackSymbol nt = Plain nt | Barred nt deriving (Eq,Show)
```

and then the relevant configurations would have the type `Configuration (StackSymbol nt) t`.

Operations like SHIFT, PREDICT, LC-CONNECT, etc. can naturally be formulated as either (i) functions that take a configuration and a grammar as inputs, and produce a list (perhaps empty) of configurations; or (ii) functions that take a configuration and a rule as inputs, and produce either a configuration or nothing (recall `Maybe` types). (These two options are not mutually exclusive!)

You could focus on calculating the maximum-stack-depth values for certain inputs, as we did in sections 1 and 2 of Assignment #7.[1] Or, particularly if you've learned about search techniques (e.g. beam search, A-star search) in other classes, you could look at the question of how a parser makes "guesses" about where to search next, like in section 3 of Assignment #7; in this case, attaching weights or probabilities to the CFG's rules would be a natural move.

4 Generalized left-corner parsing

It turns out that the three kinds of parsing that we talked about in week 7 — bottom-up, top-down, and left-corner — are not the only options. To begin to think about this, notice that the relationship between these three methods can be described as follows:

- In bottom-up parsing, the rule 'A → B C' gets used after both the B and the C have been found.
- In top-down parsing, the rule 'A → B C' gets used before either the B or the C has been found.
- In left-corner parsing, the rule 'A → B C' gets used when the B has been found but the C has not.

[1] Is there a nice way of calculating these maximum values without chugging through the transitions themselves, just based on the structure of a tree? I suspect there probably is, but I honestly don't know.

For a rule with only two symbols on the right-hand side, these are the only three options. (And notice that for a rule with only one symbol on the right-hand side, left-corner looks pretty much like bottom-up.) But if you have a rule like 'A → B C D', there's an additional option of having the rule triggered when the B and C have been found but the D has not.

The point within the right-hand side of a rule where we switch from bottom-up to top-down processing is known as the rule's *trigger point* or *announce point*. Bottom-up parsing puts the announce point all the way at the end, top-down parsing puts it all the way at the beginning, and (vanilla) left-corner parsing puts it after one symbol.

You could write a parsing system that takes as an argument a specification of where the announce point should be. If this takes the form of an integer specifying how many symbols precede the announce point, then an argument of zero will produce top-down parsing, and an argument of one will produce left-corner parsing. For bottom-up parsing, you'd need to allow specifications of how many symbols follow the announce point. So you could use a type like this:

```
data AnnouncePoint = FromLeft Numb | FromRight Numb deriving (Show,Eq)
```

(It's even possible to put the announce point in different positions for different grammar rules!)

5 Phonological rules and FSTs

Try writing a program that can convert a phonological rule in the format 'A → B / C _ D' into a finite-state transducer (i.e. an FSA with output strings; see section 4 of Assignment #6). Play with a few examples that produce different surface forms depending on the order in which the two rules are applied. (You would have talked about such cases in Ling20. Formally, what you're doing in this situation is equivalent *composing* the two transducers.)

You might use a type like this for phonological rules:

```
type PhonologicalRule = (Char, Char, Maybe Char, Maybe Char)
```


where the third and fourth components represent the left and right parts of the context requirements (i.e. the things next to the underline), which may be `Nothing` because some rules only care about one side. Or, for rules that need to "look further", you might want a list of characters instead.

6 Earley parsing

Try implementing the Earley parsing algorithm, which is a tabular/chart-based way of doing parsing that is not entirely bottom-up. It's sort of a mixture of left-corner parsing (week 7), which is transition-based, and CKY/inside parsing (week 6), which is chart-based. See e.g. Jurafsky and Martin (2000, pp.377–385).

7 Violation semirings

If you know a bit about Optimality Theory (OT), enjoyed semirings, and are up for a bit of a challenge, take a look at Jason Riggle's unpublished paper "Generating contenders", `http://roa.rutgers.edu/files/1044-0809/1044-RIGGLE-0-0.PDF`.

8 Finite-state tree automata and syntax (added 2/27/2021)

Along the lines of Assignment #8, you could try constructing finite-state tree automata to enforce something like Binding Condition A or Subjacency — or some other syntactic locality condition you know about. Remember that you can mimic basic "movement" configurations by requiring a c-command relationship between a trace and a mover.

For Condition A, a simple way to approach the problem without introducing the complications of indices is just to require that a reflexive has some appropriate locally c-commanding antecedent, i.e. allow anything for which there's a possible assignment of indices that would satisfy Condition A. So 'She washed herself' is fine but 'They washed herself' is not; no indices involved. (If you do things the "obvious" way *with* indices, you'll find that a *finite-state* tree automaton probably won't suffice.)

In connection with this you could also consider writing a function to *intersect* two finite-state tree automata. The way this works is very similar to the intersection of two FSAs that we saw (and implemented) in week 4.

Another thing to think about: for CFGs, we have outside values which are the "reverse" of inside values . . . shouldn't there be something that's the reverse of under values in a FSTA? I guess we could call them *over* values. Remember that the kind of thing that has an outside value is not a string, but rather a string-with-a-hole-in-it (which we can represent as a pair of strings); that's the kind of thing that surrounds an infix to make a complete string. With trees, what you need is a tree-with-a-hole-in-it, i.e. the kind of thing that surrounds a subtree to make a complete tree. These are sometimes called *tree contexts*.

9 More powerful tree grammars (added 2/27/2021)

Try implementing a more powerful stack-based tree grammar of the sort that we saw at the end of Week 8, for generating non-context-free stringsets. There are various ways to set up the details here; you can come up with something yourself, or have a look at work on *linear indexed grammars*, e.g. Gazdar (1988), `https://en.wikipedia.org/wiki/Indexed_grammar#Linear_indexed_grammars`. You could construct one of these grammars to generate the famous Swiss German crossing dependencies pattern (see readings on the CCLE site).

This could potentially tie in nicely with an implementation of some of the stack-based CFG parsing techniques we saw — the difference is just whether you're processing strings or trees.

References

Gazdar, G. (1988). Applicability of indexed grammars to natural languages. In Reyle, U. and Rohrer, C., editors, *Natural Language Parsing and Linguistic Theories*, pages 69–94. Springer.

Jurafsky, D. and Martin, J. H. (2000). *Speech and Language Processing*. Prentice Hall, Upper Saddle River, NJ, first edition.

Sipser, M. (1997). *Introduction to the Theory of Computation*. PWS Publishing Company, Boston, MA.