Final Project

15/2023, 11:59pm Pacific

The goal of your final project is to show that you have developed an understanding of the themes that have run throughout the quarter, and can extend these concepts creatively to tackle interesting computational problems in natural language. You can choose among three options:

1. Implementing and exploring the properties of transition-based parsers;
2. Comparing and exploring the properties of two "mildly context-sensitive" formalisms; or
3. A project on a relevant topic of your choice.

For the first option, you'll be asked to write and submit a program in Haskell, along with a short paper in PDF format that clearly explains what your program does, and how it connects to core ideas from the course.

The second option is an entirely pencil-and-paper project. You'll be asked to conduct a thoughtful review of selected primary-source literature in formal language theory, and write and submit a paper in PDF format demonstrating your understanding of the formalisms and how they connect with core ideas from the course.

The fourth option (topic of your choice) has the additional requirement that you schedule a meeting with me in Week 9 or 10 to discuss what you're planning to do. Below are some general guidelines that are applicable to all projects.

General expectations and grading

This project is not expected to be particularly large; it should be roughly the "size" of 2.5 homework assignments, give or take. The most important thing is demonstrating that you've deepened your understanding of some of the concepts that we've discussed. The best way to demonstrate this is through clear writing, so I am placing a large emphasis on the written component of this project. For projects with a coding component (e.g., Option 1), 50% of your grade will come from your paper, with the remaining 50% from the project itself. For projects that are entirely pencil-and-paper (e.g., Option 2), 100% of your grade will come from your paper.

There are many ways to earn a good grade, but some of the generally applicable criteria that I'll be looking for are:

- Understanding the core course material.
- Building on or extending things that we talked about in the course, perhaps in ways that connect them to things you've learned in your other linguistics courses.

- Making connections between ideas from different parts of the course.
- Competence with formal mathematical descriptions of grammatical systems and, where applicable, 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.)

Paper guidelin

Your paper should pages, double-spaced, submitted as a PDF. Think of this as a cohesive descri an outside audience of computational linguists. Organize your paper with an discussion. Be sure that all resources are cited properly. Your paper should owing:

Introduction

- Introduce the main topic and its motivation. Why is this topic important and interesting for the field of computational linguistics, and for understanding human language more broadly?

Body

- Describe the grammatical formalism(s) that you're working with, and, where applicable, the algorithms that are operating over these formalisms.
- For projects with a coding component: provide a clear, high-level description of your model implementation, walking your reader through how the code is structured. Think of this description as operating at Marr's computational level. Don't simply reproduce your code in the text; tell me at a high level what it is computing.

- For projects without a coding component: provide several clear, step-by-step derivations of relevant linguistic patterns, focusing on those that demonstrate the capacities of the formalisms you are discussing.

- Analyze what your model and/or formalisms can and cannot do. Show some successful examples, and some examples that illustrate particular challenges.

Discussion

- Discuss the broader linguistic assumptions of your model and/or formalisms, and how these do or do not relate to what we know about humans' linguistic faculties.
- Discuss the next steps that you or others could take to extend this work in interesting directions.
- Be sure to address any specific questions in the project prompts on the following pages.

Code guidelines (where applicable)

I should be able to understand your project by reading your paper, without digging through your code. But I'll also ask you to follow a few style guidelines to make your code easier to understand:

- Provide an explicit type signature for every top-level function.
- Above every function, include a short comment block that describes what the function does, and clarifies what your variable names mean.

- Use descriptive names. Think of your names as a type of documentation. For example, `helperFn` is not a great name for a helper function; `addFeats` is much clearer. Short names (a character or two) are great for local variables, while longer names are great for bigger functions.
- Use brief in-li ry to clarify the internal pieces of a function, but don't over-comment s for the top of a section of code, in order to explain the overall design s and data structures relate to each other.
- Make sure yo me you make a small change, to ensure that it compiles and still does . Include tester code that shows how you've tested your model's beha s.

Come talk to us if nd don't stress: if you're able to do something sensible that's within the spirit of the assignment, and reflect thoughtfully on what you've learned in the process, you'll do just fine!

1 Option 1: Transition-based parsing

In this final project, you'll have a chance to try your hand at coding bottom-up, top-down, and left-corner CFG parsers in Haskell using the transition-based parsing schemas that we learned in class. I have organi cture for you, outlined below.

Instructions: Dov .zip, unarchive its contents into the same directory on your computer, and t01_Stub.hs to FinalProject01.hs. (Please use this name exactly.) The of the stub file import definitions from CFGParsing.hs, so you can use ther if they were defined in the same file.

Please submit your PDF with your paper, and a modified version of FinalProject01_S

Notes:

Have a look at CFGParsing.hs. This sets up the basic parsing architecture that I've provided, which will become clearer as we go. But here are a few things to notice at the outset:

- The type RewriteRule nt t for CFGs is slightly modified from our previous version in that it no longer enforces Chomsky Normal Form. A NTRule now consists of a root nonterminal followed by a *list* of daughter nonterminals, so that we can now have rules of the form $A \rightarrow B$ and $A \rightarrow BC$.[1] Additionally, a new way to be a RewriteRule is to be NoRule, which won't be present in any grammar but will be used for a parser's starting configuration.

- To make things a little simpler, I'm suggesting that we start by working with the CFG type, rather than the GCFG type that we were working with in Week 6. The relevant difference is that the rules in the grammar are represented in a *list*, rather than as a function from rules to Booleans or other values. I'll also assume that there is only one initial nonterminal symbol for every CFG.

- I've given you a few helper functions that return the left-hand side or the right-hand side of a RewriteRule.

- There are several new types for working with our parsing architecture. A parse step (type ParseStep) is a triple whose members are the transition executed (type Transition), the rule invoked (type RewriteRule, and the resulting configuration (type Config). This represents a row in a parse table.
- All possible transitions that we may want to make use of are defined as type Transition. This includes the possibility of NoTransition, which will only be used for a parser's starting configuration, i.e. in the very first row of a parse table.
- The type Config is an ordered pair consisting of a stack of nonterminals and a list of terminal symbols. The stack in a configuration always allows for the bar/no-bar distinction (type Stack), which is necessary for left-corner parsing. For bottom-up and top-down parsing, only no-bar elements are used on the stack. This is a shortcut, but it suffices for our purposes.
- I've provided some simple helper functions (getTransition, getRule, and getRule) that will grab the transition, rule, or configuration of any ParseStep.

To get started, here are a few warm-up functions that you can try writing before attempting to build your bigger parsing model. We won't be grading these functions, but they're designed to prepare you for some of the trickier implementational things that might come up.

[1] Technically we could also have nonterminal rules like $A \rightarrow []$, which isn't so desirable. Think about what complications this kind of rule would introduce for parsing.

1.1 Warm-up 1: Rewrite rules (not graded)

Notice that type `RewriteRule` no longer enforces Chomsky Normal Form. Here are some ways to familiarize yourself with this change.

A. Write a functio `iteRule nt t -> Bool` that takes a `RewriteRule` and returns `True` if tha rmal Form, and `False` otherwise.

```
*FinalProject01> :  P,VP])
True
*FinalProject01> :  D,N,PP])
False
*FinalProject01> :  V])
False
*FinalProject01> isRuleCNF (TRule NP "Mary")
True
*FinalProject01> isRuleCNF NoRule
True
```


B. Write a function `isCNF :: CFG nt t -> Bool` that takes a CFG (of type `CFG`) and returns `True` if all of its rules are in Chomsky Normal Form, and `False` otherwise.


```
*FinalProject01> isCNF cfg12
True
*FinalProject01> isCNF cfg4
False
```


1.2 Warm-up 2: Parsing with FSAs (not graded)

On the Week 7 handout, we saw that transition-based parsing with an FSA would work like this:

Parsing with an FSA

- Starting configuration: $(A, x_1 \dots x_n)$
  where A is a start state and $x_1 \dots x_n$ is the input
- CONSUME step: $(A, x_i x_{i+1} \dots x_n) \Rightarrow (B, x_{i+1} \dots x_n)$
  where there is a transition from A to B labeled with x_i in the FSA
- Goal configuration: (A, ϵ)
  where A is a final state

Working with this simpler formalism will help build up the tools you need for parsing with CFGs. In particular, even though string generation with an FSA only involves a linear path through states, transition-based parsing for both FSAs and CFGs involves a *tree-shaped search* through possible parse paths.

In `CFGParsing.hs`, I've defined three helper functions that work with type `Automaton st sy`, our type for FSAs:

- `getDelta :: (Eq st, Eq sy) => Automaton st sy -> [(st,sy,st)]` takes an automaton and returns a list of transition rules in that automaton;

- `getGoalConfig :: (Eq st, Eq sy) => Automaton st sy -> [(st,[sy])]` takes an automaton and returns a list of goal configurations for that automaton as a
- `consumeFSA :: (Eq st, Eq sy) => [(st,sy,st)] -> (st,[sy]) -> [(st,[sy])]` executes the parser transition CONSUME. This function takes a list of transition rules for an FSA and a cu[illegible]ion (type `(st,[sy])`), and it returns the list of possible next configur[illegible]could be in after taking a single CONSUME step. If no CONSUME ste[illegible] current configuration, it returns the empty list.

```
*FinalProjec[illegible]elta fsa13) (2, [VW,C,VW])
[(1,[C,VW]),
*FinalProjec[illegible]elta fsa13) (1, [C])
[(2,[])]
*FinalProjec[illegible]elta fsa13) (2,[C])
[]
```

Trying writing a function that, when given a list of previous configurations, will use `consumeFSA` in order to find all full sequences of configurations that lead to a goal. We already know how to do this in various other ways, but here the aim is to write a function that will do this by searching over possible parse paths. Intuitively, at each step in parsing, the parser needs to consider all of the possible next transitions that it could take given the rules of the grammar. But only some of these next steps will eventually lead to a goal configuration; some will lead to dead ends. Your function should recurse over this tree-shaped search space, and only return the parse paths that successfully lead to a goal.

It's not so straightforward to do this with a "standard" recursive function of the sort that we have now seen hundreds of time. A function with this form

```
pathsToGoalFSA config rules goals =
    case elem config goals of
    True -> []
    False ->  \x -> ... x:(pathsToGoalFSA x rules goals) ...
```

will make it difficult to return only the successful parse paths, and not the dead ends. This is because only the current configuration is being passed to the recursive function call: the function only cares about the current configuration and where it's headed so far, but has no memory of things it's done in the past.

The way around this is to make things slightly more complicated. We can make the first argument of the function a pair consisting of the current parse configuration and a list of all of the parse steps that have been taken so far:

```
pathsToGoalFSA (current, history) rules goals =
    case elem current goals of
    True -> ...
    False ->  ...
```

The second element of this pair "keeps track of" the current parse history when it is passed to the recursive function call. Setting up this recursive case, along with an appropriate base case, will let you find only the successful parse paths.

What's the type signature of the function that I've sketched out above? It should be

```
pathsToGoalFSA :: (Eq st, Eq sy) =>
                  [(st,[sy])] -> [(st,[sy])] -> [[(st,[sy])]]
```


That is, this is a f pair consisting of a current configuration and a list of previous configurat a list of goal configurations. The function returns all of the full parses (sequ that lead to a goal. Since a single configuration sequence has type `[(st, [s` tion sequences has type `[[(st,[sy])]]`. If there are no sequences of config sult should be the empty list.

```
*FinalProject01> pa                    /W,C,VW]),[(1,[C,VW,C,VW])]) (getDelta fsa13)
   (getGoalConfigs
[[(1,[C,VW,C,VW])                      VW]),(2,[VW]),(1,[])],
 [(1,[C,VW,C,VW]),(2,[VW,C,VW]),(3,[C,VW]),(1,[VW]),(1,[])]]
*FinalProject01> pathsToGoalFSA ((1,[VW,VW]),[(1,[C,VW,VW,VW]),(2,[VW,VW,VW])])
   (getDelta fsa13) (getGoalConfigs fsa13)
[[(1,[C,VW,VW,VW]),(2,[VW,VW,VW]),(1,[VW,VW]),(1,[VW]),(1,[])]]
*FinalProject01> pathsToGoalFSA ((3,[VW,VW]),[(1,[C,VW,VW,VW]),(2,[VW,VW,VW])])
   (getDelta fsa13) (getGoalConfigs fsa13)
[]
```


1.3 CFG Parsing

Now onto the actual project: constructing your CFG parsers. Each parser that you'll implement will make use of a common computational core, a "parser base" function that abstracts away from the differences among bottom-up, top-down, and left-corner parsing. The parser-base function has parameters for (i) the starting configuration, (ii) the goal configuration, and (iii) the transition steps. So, you'll need to write separate functions that will supply these parameters to the parser-base so it can be used in bottom-up, top-down, or left-corner mode. We'll break this down into a few steps.

C. Write functions

```
shift :: (Eq nt, Eq t) => [RewriteRule nt t] -> Config nt t -> [ParseStep nt t]
reduce :: (Eq nt, Eq t) => [RewriteRule nt t] -> Config nt t -> [ParseStep nt t]
```

which execute the SHIFT and REDUCE transition steps of a bottom-up parser. Each transition-step function is of the same type: they take a list of rewrite rules and a configuration, and return a list of possible parse steps. If no parse steps are possible, the result should be the empty list.

Having the transition-step functions all be the same type allows them to be easily worked with as a group. For example, you can use `map` to apply a list of transition-step functions to the same argument. (This is a useful hint for later.)

Note: The formatting for the functions below is provided via a "pretty-print" module that makes the output of these CFG parsing functions a little more readable, like a parse table. You can ignore the details of this module.

```
*FinalProject01> shift [(TRule NP "Mary"), (TRule NP "John")] ([], ["Mary"])
===== BEGIN PARSE =====
Shift   NP -> "Mary"   ([NP],[])
===== END PARSE =====
```

```
*FinalProject01> reduce [(MTRule "S" ["NP","VP"])] (NoBar NP, NoBar VP) ["Mary"])
===== BEGIN PARSE =====
Reduce   S -> NP VP   ([S],["Mary"])
===== END PARSE =====
```


D. Now onto the fu[illegible]up parser will be split across two functions: `parser` and `bottomUp`. `bottomU[illegible]`up parser, but it is essentially a wrapper for `parser` (the parser-base functio[illegible]e starting configuration, the goal configuration, and the transition steps (i.e[illegible]I have provided `bottomUp` for you. Your task here is to write the function


```
parser :: (Eq nt, Eq t) =>
          [[RewriteRule nt t] -> Config nt t -> [ParseStep nt t]] ->
          [RewriteRule nt t] -> Config nt t -> Config nt t ->
          [[ParseStep nt t]]
```


that can be used together with `bottomUp`. This function should take the list of transition steps, the list of rules, the starting configuration, and the goal configuration given in `bottomUp`, and it should return a list of all of the successful parses that are possible by executing the given transition steps according to the given rules (a list of lists of `ParseSteps`).

When you put the two pieces together, `bottomUp` will take two arguments, a CFG and a list of terminal symbols, and it will return all of the possible bottom-up parses of the given list of terminal symbols under the given CFG. If the list of terminal symbols cannot be parsed by the given CFG, the result should be the empty list.

When you have it working, here is what you should be able to see:


```
*FinalProject01> bottomUp cfg12 (words "watches spies with telescopes")
[===== BEGIN PARSE =====
NoTransition   NoRule                ([],["watches","spies","with","telescopes"])
Shift          V -> "watches"        ([V],["spies","with","telescopes"])
Shift          NP -> "spies"         ([V,NP],["with","telescopes"])
Shift          P -> "with"           ([V,NP,P],["telescopes"])
Shift          NP -> "telescopes"    ([V,NP,P,NP],[])
Reduce         PP -> P NP            ([V,NP,PP],[])
Reduce         NP -> NP PP           ([V,NP],[])
Reduce         VP -> V NP            ([VP],[])
===== END PARSE =====
,===== BEGIN PARSE =====
NoTransition   NoRule                ([],["watches","spies","with","telescopes"])
Shift          V -> "watches"        ([V],["spies","with","telescopes"])
Shift          NP -> "spies"         ([V,NP],["with","telescopes"])
Reduce         VP -> V NP            ([VP],["with","telescopes"])
Shift          P -> "with"           ([VP,P],["telescopes"])
Shift          NP -> "telescopes"    ([VP,P,NP],[])
Reduce         PP -> P NP            ([VP,PP],[])
Reduce         VP -> VP PP           ([VP],[])
===== END PARSE =====
]
```

E. Build a top-down parser that makes use of your existing `parser` function. To do so, you will need to write the transition-step functions `match` and `predict`, which should have the same types as `shift` and `reduce`. You will also need to write a function

```
topDown :: (E          CFG nt t -> [t] -> [[ParseStep nt t]]
```

which makes use of d all possible top-down parses of a given list of terminal symbols under a gi

In some cases, you etimes encountering an infinite `predict` loop. This is a known issue with tc here is *left recursion* in the grammar— that is, when the grammar has rules $A \rightarrow A \ldots B_n$. In order to check that your parser is performing properly, you may need to modify the grammar to take out rules that have this form. Think about why these rules are problematic for top-down parsing, and not for bottom-up parsing. What could be done to get around this problem?

F. Build a left-corner parser that makes use of your existing `parser` function. To do so, you will need to write the transition-step functions `matchLC`, `shiftLC`, `predictLC`, and `connectLC`, which should have the same types as all of the previous transition-step functions. You will also need to write a function

```
leftCorner :: (Eq nt, Eq t) => CFG nt t -> [t] -> [[ParseStep nt t]]
```

which makes use of `parser` in order to find all possible (arc-eager) left-corner parses of a given list of terminal symbols under a given CFG.

G. Explore the properties of your parsing functions. As part of the discussion in your paper, consider and address the following questions:

- Are your parsers able to correctly identify all of the possible parses for a given sentence? Try testing them out on not just the two CFGs that were provided, but also additional CFGs that you construct yourself.
- Are your parsers able to parse all three types of embedding structures (left-branching, right-branching, and center-embedding) that we discussed in Week 7? Does the size of the stack grow in the ways that you'd expect with the length of the string, given what you know about these different parsing schemas? Provide some examples.
- Did you notice that any particular sentence type took longer for one or more of your parsers to parse, and was this straightforwardly related to the maximum stack depth needed to assign a parse to that sentence? If not, what else might be responsible?
- Examine how your parsers are making "guesses" about where to search next— how they seem to be navigating the search space, like the example that you considered in Part 3 of Assignment #7. What could be done to help your parsers make better or more human-like guesses? If you have learned about search techniques (e.g. beam search, A-star search) in other classes, think about how those techniques could be relevant here. More generally, attaching weights or probabilities to the CFG's rules could be a natural move. Consider how that might work.

2 Option 2: Mild context-sensitivity

In this project, you'll have a chance to take a closer look at the primary-source literature on so-called "mildly context-sensitive" grammatical formalisms. This is the class of formalisms that, by hypothesis, exhibit [illegible]l capacities as natural language syntax. Mildly context-sensitive grammars [illegible]an context-free (the ones at Level 2 of the Chomsky Hierarchy), but are [illegible]as the full class of context-sensitive grammars (the ones at Level 1). In part[illegible]d class of grammars are those that can generate certain types of *crossing d*[illegible]ested in natural language (such as $a^i b^j c^i d^j$), but cannot generate other cont[illegible]hat are outside the capacities of natural language syntax (such as $\{a^n \mid n$ is [illegible]

Here, you'll write a [illegible]ildly context-sensitive formalisms that look very different, but turn out to be equivalent in their string generative capacity: **Linear-Indexed Grammars (LIGs)** and **Tree-Adjoining Grammars (TAGs)**. In Week 9, we will be talking about reaching the zone of mild context-sensitivity by augmenting a context-free grammar with memory in the form of a stack. This is the idea behind Linear-Indexed Grammars. Tree-Adjoining Grammars work in a very different way, but end up in the same place in computational complexity. Thinking about how these two formalisms operate, and how they generate the same types of string patterns in different ways, will allow you to extend your knowledge about formal language theory to a question that is currently an active topic in the literature. If we take these mildly context-sensitive formalisms as hypotheses about the computational properties of natural language syntax, what are the differences between these hypotheses, and how would we test them empirically?

Instructions: Download `FinalProject2.zip` and unarchive its contents. Read the articles that are included in this folder, which should form the basis of your literature review. You are free to extend your literature review beyond the papers listed here, but if you do so, make sure to cite any additional articles appropriately.

- Shieber (1985) is the paper that is most commonly cited as "settling the question" about the non-context-freeness of natural language syntax. It discusses the famous example of crossing dependencies in Swiss German.
- Pullum (1986) is an amusing story of the discovery of non-context-free patterns, including others that were discovered around the same time as Shieber (1985).
- Gazdar (1988) provides an introduction to what he calls Indexed Grammars, but later became known as Linear-Indexed Grammars. Focus on the first 7 pages for a description of the formalism. Focus in particular on the description in example (2); this corresponds to the constrained stack-based tree automaton that we are discussing in class, where stack information is transmitted between a parent and only one daughter.
- Frank (2003) provides an introduction to Tree-Adjoining Grammars. Focus on the first 14 pages for a description of the formalism. He also provides an overview in Section 2 of all of the relevant background, from FSAs to CFGs to the Swiss German-type construction. The formalism he describes as an Embedded Pushdown Automaton operates with roughly the same idea as an LIG.

Please submit your paper as a single PDF. Within your paper, make sure you consider and address the following questions. But please do not submit your responses in list format. What you submit should be a cohesive and self-contained review paper, organized into an introduction, body, and discussion, following the general paper guidelines laid out at the beginning of this assignment.

- What are the formal components of a Linear-Indexed Grammar and a Tree-Adjoining Grammar? Discuss the objects that these grammars generate (strings or trees), and describe the system of rules that generate these objects.

- Provide **step-by-step derivations** showing how both an LIG and a TAG would generate[2] the context-fr... $\geq 0\}$. These strings exhibit nested dependencies between the a's and th... that these dependencies are generated by an LIG and a TAG, to the ...ncies are generated by a CFG (as we saw in Week 6). If you draw you... please make sure they are legible.

- Provide **step-**...howing how both an LIG and a TAG would generate the following thre...gsets. Discuss why these stringsets cannot be generated by a context-...

$$\{ww \mid w \in \{\text{a,b}\}^*\}$$
$$\{a^n b^n c^n d^n \mid n \geq 0\}$$
$$\{a^i b^j c^i d^j \mid i \geq 0, j \geq 0\}$$

- Below are two stringsets that are within the context-sensitive zone, but are stringsets that neither an LIG nor a TAG can recognize. Discuss why, intuitively, these are outside of the capacities of these two "mildly" context-sensitive formalisms. (Note: you don't need to provide a proof, just a thoughtful discussion.) If we take LIGs and TAGs to be hypotheses about the nature of natural language syntax, what does this tell us about the predictions of these hypotheses?

$$\{www \mid w \in \{\text{a,b}\}^*\}$$
$$\{a^n b^n c^n d^n e^n \mid n \geq 0\}$$

- What are the key similarities and differences in the ways that LIGs and TAGs generate the kinds of crossing dependencies that they're able to generate? Here's a useful idea to get your comparison going. In class, we saw that we could get to a pushdown automaton that would recognize the same stringsets as a CFG, by taking a finite-state automaton and adding stack-based memory. Compare this to how we get to an LIG by taking a CFG and adding stack-based memory.

  Similarly, in class, we saw that we could get to CFG by taking a finite-state grammar and adding the ability to grow strings "in the middle." Compare this to how we get to a TAG by taking a tree grammar that's like a CFG (e.g., a strictly-local tree grammar), and adding the ability to "grow in the middle."

  What does this comparison suggest about why LIGs and TAGs generate exactly the same class of string languages?

[2] Here I'm using the word "generate" in a somewhat looser sense, to mean not just the sets of structures that the grammar directly produces, but also the sets of strings that are yielded by those structures.

3 Option 3: Topic of your choice

You may also choose to pursue a project topic of your choice. However, this option has an additional requirement: you must come up with a specific plan of what you are going to accomplish as part of the project, and th**g with me in Week 9 or 10** to discuss your idea. My schedule tends to fil ance, so contact me by email as soon as possible to set up a meeting time. Th ng is to make sure that your project is a reasonable size and is something th nonstrate your understanding of the core course material.

If you choose this o our project in the following format: submit a PDF with your paper, and if y le, a Haskell file named `FinalProject03.hs`. If your file imports any code fi les that we created in the course, you must also submit those modules.

4 References

Frank, R. (2003). "Restricting grammatical complexity." *Cognitive Science* 28: 669-697.

Gazdar, G. (1988). "Applicability of indexed grammars to natural languages." U. Reyle and C. Rohrer (eds.), *Natural Language Parsing and Linguistic Theories*, 69-94. D. Reidel Publishing.

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

Pullum, G. (1986). "Footloose and context-free." *Natural Language & Linguistic Theory* 4(3): 409-414.

Shieber, S. (1985). "Evidence against the context-freeness of natural language." *Linguistics and Philosophy* 8: 333-345.

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