PROJECT Interim Report: Exploring Ambiguity Detection and Poetry Generation using Tree Adjoining Grammar

SECTION 1: Narrowing Down Our Task

Poetry is open-ended – many writings could count as poetry. Yet, for a text, we have some judgment of whether it qualifies as a poem – by its popularity among academics or critics, or its appeal to you as an individual reader. And, while random strings of letters may appear in (or as) poems, we often look for some familiar structures, like repeated sounds or grammatical sentences or emotionally connected words. On the other hand, we also look for variation within and across poems.

For example, above, lines 1, 2, and 4 repeat an end rhyme (childs, while, files) and a stress pattern (with stresses on capitalized words), but line 3 – maybe pleasantly – breaks this pattern.

In contrast, consider:

Arguably, here, the repeated lines are more monotonic in rhythm, and the last two lines are ungrammatical and seem meaningless; this poem(?) appears altogether less pleasing.
These examples illustrate two motivations to study poetry in computer science: capturing human taste (and thought), and systematizing language. Or, as Manurung cites Binsted: with Artificial Intelligence, ever since Deep Blue beat chess masters, “more artistic-based tasks - - are often proposed to be the defining benchmark” of intelligence; and with Natural Language Generation, there is a call for flexibility to generate “fluent and natural-sounding texts” or texts “where the communicative goal is vague.”

It is a sizable task to specify a poetry generation process and what parts a computational system should automate. Using linguistic distinctions for language, the parts might be, for example, sound (phonemes, stresses...), word meaning (synonyms, connotations...), or grammar (order of clauses, verb tenses...).

Existing systems show a range of methods and focal points. Some use fill-in-the-blank templates to generate variation in meaning, for example “a ___ laptop was ___ without a trace.” Some use neural networks to induce poems from frequency distributions of words or characters. Some relevant systems do not generate text, but use statistics to analyze and reframe a poem visually, or provide dictionary-like tools for more involved searches like rhymes or stress weights or descriptive words. Manurung describes a system that uses evolutionary algorithms over a rich grammar, to generate well-scoring phrases measured by some evaluation functions. More specifically, Manurung interprets poem writing as a state-space search, where the states are some features of language, like sounds, words, or phrases. Since a poem calls for both structure and variation, a computational generator needs to recognize both incorrect states and desirable future states.

Studying the examples above, we find three broad characteristics that we would like in a system: 1) the ability to generate large amounts of varying output, 2) the ability to detect and guarantee “good quality”, and 3) the possibility to improve quality in specified features, like “be stricter with rhymes” or “add more ambiguity.”

We quickly note a tradeoff between these criteria: for more variation in output, we will have difficulties to specify features or guarantee their quality, and vice versa. For example, for limited feature specifications like rhymes, we could simply apply template systems. However, this would severely limit possibilities for other feature specifications and the ability to generate large amounts of varying output. On the other hand, neural networks might be able

Rhymezone http://www.rhymezone.com/
could generate large amounts of output, even detect “good quality” based on a sizable dataset. However, it is difficult to tackle specific features in neural networks without large modifications in their architecture and datasets.

Like Manurung, we choose Tree Adjoining Grammar (TAG) as a backdrop for our studies. This is a rich grammar that can represent detailed features, such as relative clause attachments or subject and object attachments – that is, for the sentence “I threw a ball that was heavy.” TAG can express how “I” is the subject of “threw”, how “ball” is the object, and how “that was heavy” is a relative clause. We hope that these detailed features allow us to specify features (like “swap the relative clauses between these two sentences”), as well as to generate more varying output.

We choose TAG also for a practical reason: it is my adviser Bob Frank’s field of expertise, and I already spent a summer analyzing a new parser that our research group developed. Thus, I already have familiarity with the data formats of the parser, and further applications might help evaluate the accuracy and applicability of the parser.

For a literary or poetic feature, we pursue ambiguity – a sentence or phrase having multiple plausible meanings. For example, “I saw Jack come up with a bag.” could both mean that Jack designed a bag, or that Jack carried a bag up a hill. For another example, “Jack spoiled the lab with his cookies.” could mean that Jack ruined the lab space or that Jack treated his lab team well.

We choose ambiguity because, traditionally, it separates computer science and literature, yet I find it as an emerging connection point. Traditionally, programming languages have aimed for deterministic interpretations. However, so called higher level languages like Python and R, or commercial AI like SIRI or Alexa, try to guess user intentions from ambiguous expressions that humans find easier to use or learn. For example in Python, the ‘+’ operator could mean integer addition (1+1=2) or string concatenation (“hel”+“lo”=“hello”). Furthermore, probabilistic algorithms and statistics (and big data programs) give a numerical perspective on ambiguity. For a common example, Word2Vec interprets word senses as probabilistic distributions over their neighboring words (that is, a word is defined/represented by how often it appears with other words). On the one hand, statistical programs give literary studies new tools to engage with large amounts of text at

---

8 In the first case, our ambiguity lies with the phrases “come up“ vs. “come up with.” This could be conveniently specified in TAG using “coheads” and “adjunctions.” In the second case, our ambiguity lies with the word “spoiled” and its multiple word senses. These word senses would not be detectable with TAG, but would require lexical resources like Wordnet. See Princeton University "About WordNet." WordNet. Princeton University. 2010. <http://wordnet.princeton.edu>
once. On the other hand, with machine learning, programmers (and possibly computer scientists) may become more and more involved with “real-life” applications, which are prone to involve ambiguity.\(^9\)

We further categorize ambiguity into more manageable parts. For example, above we mentioned cohead ambiguity for “come up with”, and word sense ambiguity with “spoiled.” We look at example poems to identify some types of ambiguities. For our example poems, we hypothesize that religious poems would be rich in ambiguity, and choose eight poems from the web page onbeing.org. Appendix A has our notes on the full poems. Below, we gather some generalized categories.

1. Whether a (prepositional) phrase attaches to a previous phrase,
   e.g.,
   
   The one
   Who played
   By the rules
   I came up with
   The one\(^{10}\)

   Here, “By the rules” could be connected to the line above (“who played | By the rules”) or below (“By the rules | I came up with”) or even both. The line breaks strengthen the ambiguity. We note that a line break could stand for a comma even when no comma is written (having us read “By the rules, I came up with”).

2. Whether a phrase ends or is further qualified.
   e.g.,
   
   I lived
   no more.

   Compared to the case above, we have no attachment ambiguity for “no more.” However, a newline introduces an ambiguity at the end of the line “I lived.”

3. Whether we have nested relative clauses,
   e.g.,
   
   The mother’s night prayer for her son to make it home safe \(^{11}\)

---


Here, the mother might pray that her son makes it home safe, or that she herself makes it home safe for the sake of her son (more obvious in the form “to make it home safe for her son”).

4. Whether we have a cohead, e.g. “come up with”, as seen before.

5. Whether we have an idiom, e.g., “we take the same gold for scepter and crown” 12 Here, we may take gold in exchange for scepter and crown, or interpret gold as (“take it for”) scepter and crown.

6. Which word sense we have, e.g., “shoot a duck” vs. “duck a shot;” here, “duck” could be a noun (animal) or a verb (drop down).

7. Which word we have for the same sound (homonymy), e.g., “duck” and “dock.”

8. What pronouns refer to,
   e.g.
   
   we take the same gold --
   that has disguised you -- 13

   Here, “that” could refer to either gold or the action of taking gold.

Given our TAG, we focus on phrase and pronoun attachments (or “structural ambiguity”), as illustrated in cases 1-4 and 8 above. We also focus our initial explorations on the poem “Every riven thing,” 14 because of its explicit use of phrase ambiguity: its first sentence “God goes, belonging to every riven thing he’s made.” is repeated across the poem with different punctuations, for example as

   – a stillness where
   God goes belonging. To every riven thing he’s made
   there is given one shade --

More detailed examples follow in section 2 and appendix A. To allow for both setbacks and extensions, we formulate our goal as follows: detect, illustrate, and finally generate structural ambiguity.

**SECTION 2: Results on Ambiguity Detection**

This section sketches how we apply our TAG parser to detect ambiguity. The detailed results and code are in appendix B. We also include the preceding code for running our parser and its dependencies on Yale’s High Performance Clusters; these are in appendix C.

To explain our results, we first briefly overview our tools – the TAG formalism and our TAG parser.\(^{15}\)\(^{16}\)

At this stage, we may ignore the details of the TAG formalism. However, for a flavor, we mention that for any input phrase, our parser outputs a parse tree that includes phrase attachments. A sample image is below:

![Figure 1: A sample TAG parse tree for the phrase “every riven thing he's made.” The leftmost numbers are token sequence ids, indicating the position of the word in the original word sequence (e.g. “every” is the 0\(^{th}\) token). The](image)

---


rightmost numbers prefixed by t (e.g. t29 for “thing”) are so called “supertags,” which are like more detailed part of speech tags and may be ignored at this stage. The arrows indicate attachments, where the parent word points at its children. The small texts next to arrows, e.g. “adj.” for “adjunction”, are the relations between the two words.

More importantly, we note that our parser provides for each parse a (logarithmic) likelihood score. These scores reflect how well the parse would match our training set, the Penn TreeBank (since our parser relies on a neural network). Our hope is that the Penn TreeBank is large enough to reflect general grammatical structures in English, so that only well-formed English phrases would receive good scores.

Our hope is to use our parser scores to detect well-formed English phrases. Let us call these good_scored_phrases. We test this notion on our poem. We first cut each sentence into all possible contiguous sequences of words. For example, we would cut the sentence “I ran away.” into sequences “I”, “I ran”, “ran away”, and “I ran away.” We then parse each sequence and inspect their scores. We manually find a threshold that removes what we consider ill-formed English phrases. The rest we call good_scored_phrases. To aid our effort, we draw a distribution of the scores, and visualize parse trees for given chunks. (See Appendix B Section Thresholding).

We then consider ambiguity in terms of our parser. Recall the phrase(s) “The one Who played By the rules I came up with.” Let us call an unpunctuated sequence of words a “chunk.” We might split this chunk into well-formed (or good_scored) phrases as follows:

“... The one Who played By the rules | I came up with ...“

Notably, we would interpret that the one played by some rules, but I might have been fully unaware of the rules. As an alternative, we might split this chunk as

“... The one Who played | By the rules I came up with ...“

Here, the one might have been unaware of any rules, while I came up with some rules.

Thus, if a chunk can be split into two good_scored_phrases, we find two ways to detect ambiguity: First, we can have an ambiguous ending by introducing a line break between the good_scored_phrases. E.g.

The one Who played By the rules
I came up with.

or
I lived
no more.

Second, if the chunk can be split in several ways into good_scored_phrases, we have ambiguous phrase attachments:

The one Who played
By the rules
I came up with.

Thus, we have a way to detect ambiguity in a chunk of words, relying on our ability to detect well-formed English phrases. (See Appendix B Section ambiguous phrases)

However, we still want to find ambiguous readings of whole sentences, such that the whole sentence makes sense. This may not be true in the above example, “The one Who played By the rules I came up with” – if this chunk was preceded by a single verb (e.g. “I saw The one Who...”) and followed by nothing else, we could only read it in one way, instead of the two ways shown above. In contrast, if we also had a following noun (e.g., “an idea”), read we could have both readings of “I saw The one Who played | By the rules | I came up with an idea.”

To find ambiguous readings of whole sentences, we use simple dynamic programming: Starting from the last word of the sentence, we find all possible ways to concatenate non-overlapping good_scored_phrases from a given position the the end of the sentence. (See Appendix B Section Full-Sentence Readings/Paths)

We leave with a range of possible further directions:

1. We may apply our methods on various genres, such as repair manuals and textbooks that are not meant to be ambiguous. We would like to see if these indeed allow less ambiguity (in structure).

2. We may compare the line breaks that we discover to the line breaks that poets decide upon, and reason if our parser does capture structural ambiguities and if the poet had such ambiguities in mind.

3. We may further consider how to visualize or illustrate the large number of possible readings of a single sentence. The multiple paths from start to end along
good_scored_phrases do not yield a tree, and since the differently split phrases still overlap, it is difficult to visualize multiple splits at once.

4. We may explore perplexity, or the amount of “surprise” we face as we travel along a path. This should require a more extensive parser that serves as a language model.

5. We may explore language generation, inverting our parser process to generate language. We might use our ambiguity detection to infer rules to generate ambiguous language.

6. For “poeticness”, it would be interesting to add in semantics and phonetics. As time allows, we may consider phonetic constraints, relying on tools such as Prosody or Rhymezone, or semantics, relying on tools such as Wordnet and Rhymezone. 17

APPENDIX A: Notes on eight poems from Onbeing.org.
APPENDIX B: Python Code (in Jupyter Notebook) for Detecting Ambiguity with TAG parser.