Spew: A Practical Text-to-Code Translator

Michael Tan
New York, NY
michaeltan@nyu.edu

Arvind Ramgopal
New York, NY
Ar3514@nyu.edu

Abstract

Spew is a natural language processing application designed to generate formalized, language-specific code derived from spoken English. This paper will explore the methods and techniques used to achieve this goal. Furthermore, we will discuss several common issues faced by programmers globally and will explain how Spew can address and solve some of those problems. The design we used to create Spew is derived from existing areas of NLP research and will be discussed in depth.

1 Introduction

Programmers are familiar with the frustration that arises from bugs in code. One particularly annoying and consistently recurring bug is the syntax error. Regardless of one's expertise, it is inevitable that there will be a typo that can lead to a loss of productivity as the programmer scrambles to find and fix it. We were inspired to come up with a solution to avoid these syntax errors and allow programmers to stop having to worry so much about punctuation and language-specific details. In short, we are trying to create a generalized language platform to allow the generation of code in any language. In theory, the application Spew will be able to take in a voice input and output text that contains syntactically correct lines of code. However, the first step in this project is to tackle the parsing of the text itself.

Before discussing our approach, we would like to share some other applications we believe Spew would be well suited for in our modern world. We have found that Spew would be a solution for those with disabilities (e.g. ALS, blindness, amputation, etc.) and help them code without having to use their hands or keyboards at all. Another issue that was seen is that coding contains many repetitive bits of code and Spew would alleviate the pain of having to type many of the same lines over and over again. We believe with the right specifications Spew could improve productivity significantly whether it be small programs or large collaborative sections of code. Another prevalent issue is programmers' tendency to engage in prolonged typing, which can lead to carpel tunnel syndrome. Once full-fledged Spew would be able to alleviate any tension that may impact the hand and wrists.

Once more, the eventual goal of Spew is to provide a framework/language for users to talk in, and through our application that language will translate into any coding language specified by the user. A formal definition of Spew is that it is a natural language processing application designed to generate formalized, language specific code derived from spoken English.

2 Approach

Our approach is inspired by the concepts of POS tagging and finite state machines. When first tackling this project we realized the need to partially abstract programing languages as a whole. For example, a user of Spew who is not familiar with the Java programming language should not be rendered incapable of programming in Java – likewise, an experienced Java user who is trying to write a script in Python should be able to do so effectively. We expect users of Spew to be familiar with at least one programming language and have a general understanding of programming conventions in general. As such, we set out to discover an intermediary 'language,' between English and a specific programming language, which could provide access to all programming languages.

To begin, we categorized spoken English words in a custom, formal tagging structure. Based on the methodology of POS tagging, we

wanted Spew to be able to tokenize sentences and tag individual keywords appropriately. Appropriately, in this context, means that Spew will be self-sufficient in its capability to understand the relationship between the Spew Language and a programming language. The use of custom tags allows us to attempt to generate some sort of syntactic understanding and teach Spew to handle the rest.

Each coding language has a group of words that are treated as special keywords that are identified by the language to have a special function, so we took it upon ourselves to tag each word in each language. However, for our project we only worked with converting text to Java so we only tagged the set of keywords in Java as special keywords, but theoretically we would have done this in every single coding language.

Many people and industries use NLP and specifically POS tagging to perform tasks such as text generation and text summarization, but most of the time they use libraries, corpuses, and software that have already been created because it has been generalized for the English language. However as stated above, our words are not synonymous with those of the English language. I think it is fair to say that the words we are dealing with are homonyms, they are spelt the same but have totally different meanings. Anyways, the tools that take a piece of text and format its contents are called Processing Resources. Some Processing Resources are tokenisers, sentence splitters, POS taggers, gazetteers, finite state transducers, orthomatchers, and coreference resolvers (Cunningham 2002). There are many programs out there that already have all of these functions, for example, GATE and ANNIE. GATE is a framework and graphical development environment which enables users to develop and deploy language engineering components and resources in a robust fashion (Cunningham 2002). Combined with GATE is a set of processing resources which packaged together form ANNIE, A Nearly New IE system (Cunningham 2002).

2.1 Process Resources

To provide some context we will give short definitions of each of the Process Resources. A tokeniser, one of the Process Resources, splits the text into simple tokens such as numbers, punctuation, sybmols, and words. A sentence splitter is a waterfall of finite state machines which partitions the text into sentences. Without out this part there usually cannot be a tagger (our project has a variation of the sentence splitter and is something we will go into more detail on later). The tagger is a part-of-speech tagger (POS tagger) that produces an annotation on each word or symbol. A popular tagger that is used is the Brill tagger, developed and invented by Eric Brill in his 1995 PhD thesis, and can be summarized as an "error-driven transformation-based tagger" (Brill 1992). We have modeled our own tagger that relies heavily upon the sentence splitter and a weighted parser. Again these facets of our project will be more deeply explored later in the paper. The gazetteer is a list that contains items such as cities, organizations, days of the week, etc. Our project does not have any need from this Process Resource. The semantic tagger is an algorithm that follows hand written rules about phrase structure and pattern rules. There are many different semantic taggers each with their own set of rules, and we have our own version of this that works in parallel with our parser to detect certain patterns to improve the efficiency of our overall program. The orthomatcher, an optional module, performs co-reference or entity tracking, by recognizing relations between entities (Cunningham 2002). We did not have any need for an orthomatcher in Spew. Lastly is the coreferencer. The coreferencer finds identity relations between entities in the text (Cunningham 2002). Spew does not have any use for this Process Resource either.

2.2 Building From Scratch

To make Spew successful and as efficient as possible it was easier and much wiser to create all of our own Process Resources from scratch rather than modify existing ones, because if we were to take the route of modification there would be too many edge cases and it would be like fitting a square peg into a round hole. We bundled together our versions of the sentence splitter, tagger, semantic tagger, and tokeniser into one giant parser for our program. We wanted our parser to be able to handle all of the tagging in one go instead of splitting it up into separate executions. Because our input is coming from a Voice to Text software/API there would be no punctuation, so our "tokeniser" would meld together with the sentence splitter and break up the text initially into lines and then into words. Each line was treated like a separate job, because each line would translate to a real line of code. In our "language" we made the word "end" identify as the end of a line of code in an input.

So our sentence splitter would essentially split the text on the word "end". After this step the lines were then split into an array of strings which would then have to be fed through the parser to be tagged and made ready for our algorithm.

3 Process

3.1 Language-Specific Keyword Mapping

We have created a dictionary of keywords for each coding language and once the text has been split into lines and then an array of strings, our next job is to tag the words of that array. Our tagger is not specifically defined but it is one of the integral pieces of our parser. It performs just like a tagger except it performs along with other functions to give a final result that is more than just the tagged words. So we loop through the array in our queue of arrays and we tag all the keywords that appear in our coding language keyword dictionaries. We gave every keyword in the dictionary a different annotation, because our parser can theoretically also detect different sequences and patterns of annotations. For example in Java a function header would be written as follows:

public int getTotal(int num, int number){ }.

However because Spew gets it through the Voice recognition software/API it will look like the following:

public int get total int num int number

And after it has been tagged by our system, these will be the tags assigned to the words in a different array:

public → access
int → datatype
get → name
total → name
int → argument
num → argument
int → argument
number → argument

3.2 Further Programming Details

As you can see from above those are the tags that we generate. If you look closely you can see that the first "int" is tagged as a datatype but the second and third are tagged as arguments, and this is because the system we designed is smart and can pick up on the pattern that if a code specific keyword follows on word that is a non-keyword then most likely it is used as an argument, whereas initially it is used as a data type. These pattern recognitions and other types of special phrase detection falls under our "semantic tagger" portion of our parser. If you recall, our tokeniser only had words to split into tokens but it had no symbols or punctuation so one of the biggest challenges was detecting words that were actually symbols or numbers (e.g. "equals" should be translated to "=", "two" should be translated to "2", or "dot" should be translated to "."). Because these were not keywords they would not get tagged as such but they appear in specific cases, and they make up a large part of coding, so we made it in such a way that most of the time words like "equals" and the other cases would be immediately converted to there symbolic counterparts unless a specific pattern was matched where the translation was not necessary. To develop an algorithm that has to detect and handle these cases on such a large and general scale is a hard task to deal with.

This part of the parser where we had to detect patterns and phrase structures is where we deployed our version of finite state machines. The way Spew is programmed is that it has its own language and syntax that the user has to follow but that language and syntax is a gateway to the many coding languages and coding syntax out there. We designed it in such a way such that each word in an array triggers a different state. For instance if the user wants to create an object they would speak the word "create" and that would trigger to program to realize the state was in an object creation state. We have various trigger words that trigger various states and efficiently completes the line's syntax. You can think of the entire program as a network or graph of states, where each state (which is technically the word in the array (the words act as triggers)) connects to another state. And each line can be recreated by following a path of states and at the same time each path has its own sequence of syntax and if each line matches to a path, then each line matches to an output with the appropriate syntax. By having finite states it is fairly simple to create shortcuts in the program because you can group together similar phrase structures. For example if you have two lines that have different syntaxes but start out with the same phrase structure then you can still have them start at the

same states and just branching out to different states when necessary. This will lower the runtime because there will be less nodes to process.

4 Conclusion

There is still a lot of work to be done and many more techniques to be used and developed, but he first steps have been taken and hopefully they are in the right direction. The biggest challenge to tackle will be to account for all cases in the most general and abstracted way possible so that this can really be used for all coding languages. It would be quite wasteful to hardcode all of the edge cases, and it would be quite unwise to do so if Spew were ever to scale up. Ultimately Spew could potentially be a very powerful tool, and we believe its backbone could be used in future innovations, which is why research and implementation now would be very fruitful.

Reference

Cunningham, Hamish, et al. "A framework and graphical development environment for robust NLP tools and applications." *ACL*. 2002.

Eric Brill. 1992. A simple rule-based part of speech tagger. In Proceedings of the third conference on Applied natural language processing (ANLC '92). Association for Computational Linguistics, Stroudsburg, PA, USA, 152-155.