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.

2.1 Language-Specific Feature Selection

Each coding language has a group of words that are treated as special keywords. Programmers of these languages are encouraged not to name variables after these built-in names. These keywords are the core of any programming language. Our first approach to account for multiple different programming languages was to provide a structure to tag each keyword in a specific language. A note: identifying all keywords across the programming languages is the goal; however, for this project, we were limited in scope to the Java programming language (reasons for choosing Java will be discussed later), hence, our project currently deals with Java-specific keywords only.

abstract	continue	**for**	**new**	switch
assert***	default	goto*	package	synchro-nized
boolean	do	if	**private**	this
break	**double**	imple-ments	**protect-ed**	throw
byte	else	import	**public**	throws
case	enum****	instance-of	return	transient
catch	extends	**int**	**short**	try
char	final	interface	**static**	void
class	finally	**long**	strictfp**	volatile
const*	**float**	native	super	while

Figure 1: Table of Java Keywords

Figure 1 displays the entire list of Java keywords. There are approximately fifty keywords in the language; however, our current project's scope is limited to the keywords we felt were the most essential in their influence on both Java and programming in general. Most of our selected keywords are bolded – we chose to focus primarily on three categories of key words. Data types and the 'for loop' were emphasized due to their widespread use in all programming languages. We chose to use properties of function headers for a more Java-specific focus. Thirdly, we chose to include classes and access modifiers as classes play a crucial role in all Java programming (every component of Java is a class).

2.2 Processing Resources

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).

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). Lastly is the coreferencer. The coreferencer finds identity relations between entities in the text (Cunningham 2002).

2.3 Building From Scratch

We chose to build Spew from scratch instead of relying on previously existing Process Resources to avoid wasting time parsing through and repurposing code for our own needs. In order to be as successful and as efficient as possible, we used them as a reference while focusing on modularizing our code. Thinking long term, when Spew is able to parse input and 'Spew' out code in an increasing number of languages, spending time designing a modular and scalable platform was of high importance. One of our first solutions was to create a dictionary of Spew functions to create a connection between our parse and a process. In Python, we mapped a function string to a function object, which Spew then executes with an argument vector.

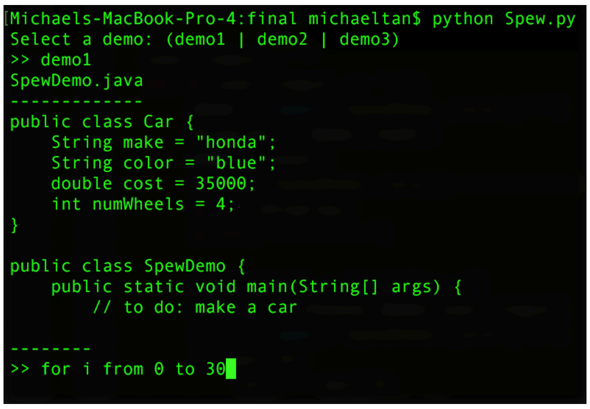
```
function_dict = {"for":spew_for_loop_pp,
                 "create":spew_new_object_pp,
                 "func-
tion":spew_new_function_pp,
                 }

    split_input = user_input.split()
    # find which spew function to execute
    for i in split_input:
        i = i.strip()
        if i in function_dict:
            spew_preprocess=function_dict[i]
            temp                              =
split_input[split_input.index(i)+1:]
            break
    o = ""
    try:
        o = spew_preprocess(split_input)
```

Figure 2. Mapping Function String to Spew Function

We define a variable "function_dict" which is a dictionary mapping a string key to a Spew function. This function_dict will be expanded in the future and will have a programming language as its key, followed by the specific function strings accordingly.

The current design for Spew is similar to that of a classic bash shell. Figure 3 shows Spew.py running as a foreground process, which features its own command line parser. This is only an initial setup for demo purposes. In the future, we hope to integrate Spew into a popular text editing software, Sublime Text, as a plug-in. This will allow you to quickly open a 'Spew' command line and enter in a prompt.



```
Michaels-MacBook-Pro-4:final michaeltan$ python Spew.py
Select a demo: (demo1 | demo2 | demo3)
>> demo1
SpewDemo.java
-------------
public class Car {
    String make = "honda";
    String color = "blue";
    double cost = 35000;
    int numWheels = 4;
}

public class SpewDemo {
    public static void main(String[] args) {
        // to do: make a car

--------
>> for i from 0 to 30
```

Figure 3. Spew's Shell-like Interface

We bundled together our versions of the sentence splitter, tagger, semantic tagger, and tokenizer 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 "tokenizer" 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 created a dictionary of keywords for each coding language. Afterwards, text is split into lines and then an array of strings. Spew then sets out to tag the words of that array. 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 also detect different sequences and patterns of annotations. For example in Java a typical function header would be written as follows:

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

Remembering that the end goal of Spew is to have built-in voice-to-text translation, we are faced with a slight dilemma. What is the best way to define a verbal structure for a relatively syntactically complex structure such as a Java function header? Our approach was to simply accept the text as someone would speak verbally. For example, if one spoke the following:

```
"public int get total int num int number"
```

Our system would tag each input token separately. Below is an example of its tagged output. In this case, 'name' is a tag given to any unidentified keyword.

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

3.2 Phrase Structure Rules for Identifying Code Sentence Structure

After established a mapped relationship between word and tag, Spew will rely on phrase structure rules to identify what type of code it will be generating. It can be easily observed that any arguments are declared with a datatype followed by a name, so we may write:

```
arg → datatype name
```

In Java, it is required to specify a return type when declaring a function. This is generally the first part of the any function declaration, but *can be proceeded* by a number of access modifiers (static, private, public, etc.). As such, we can create some rules to account for this:

```
RetGrp → access datatype
RetGrp → access access datatype
```

Finally, to deal with the name of the function itself, we can write the following rule:

```
FuncName → RetGrp NameGrp
NameGrp → name name (name…)
```

NameGrp is a rule that allows a combination of any arbitrary number of names. When this particular function is processing name group, conventions of the language are sure to be upheld. For example, the parse of "my very special name" would be camel-cased and would result in, "myVerySpecialName".

3.3 Sentence Building

Generating the actual sentence would take advantage of several of NLTK's built in parsers, including an implementation of a Recursive-Descent parser, a Shift-Reduce Parser, and a Left-Corner parser (NLTK, 2015).

3.4 Further Programming Difficulties

Another issue we faced was attempting to implement a way to access methods and properties of a class or object. One main method for printing output in Java is to use the following function:

```
System.out.println()
```

Several difficulties arise in the implementation of this function. Firstly, how can we define a way to concatenate tokens into object accesses? We asked several peers how they would generate the above function call verbally, and noted that one method for combining tokens was to use a verbal word as a split.

```
"System dot out dot print 'l' 'n'"
```

In this case, we can process the string in a way that whenever we encounter the string, "dot," we combine the tokens on the left and right with a period. Spew does exactly this.

3.5 Ever-Expanding Phrase Structures

Given that there are almost fifty separate keywords in Java alone, we must consider what will happen when we attempt to create phrase structure rules for not only Java but for all languages. What is the best way to solve this problem? Currently, our phrase structure rules are language specific. No Python user would specify a return type for a function declaration, and especially not an access modifier (they do not exist in Python).

One solution is to group a variety of phrase structures together for different languages. We can alter the implementation of a parsing function by having it examine one particular set of phrase structures at a time. This mapping of all language specific functions to a global programming concept (e.g. "function", "for loop", etc.)

allows Spew to 'understand' syntax in a generalized, non-language-bound form.

4 Current Stages

Currently, our program does not implement phrase structure rules for use in a CFG parse. We have set up a modularized codebase to continually build new syntax generation functions into Spew and have currently implemented the three functions mentioned earlier.

Since we couldn't rely on parsing algorithms to choose the proper syntax structure, we've taken inspiration yet again from a bash shell design. Spew will process input by tokenizing all words in the input string. The 0th token will execute the corresponding function and tokens[1:] will be used as an argument vector. We understand that this method contradicts the end result of Spew, and is closer to "hard coding" than it is CFG parsing; however, our goal of this version of Spew was to *show* what is possible. It is by all means a theoretical demo of what we hope to be able to achieve in the future.

5 Instructions for Running *Spew.py*

We invite anyone who is reading this paper to download our demo version of Spew.py from GitHub (link found under resources). Instructions for running Spew.py is as follows:

Run Spew.py from the command line using:

```
python Spew.py
```

Spew will prompt you to enter a demo number, try entering:

```
demo2
```

We have a file with an array of size 5 and an ArrayList of size 5 as well. Try entering the following command viewing its output:

```
for i from 0 to a dot length
```

Notice that Spew has generated a basic for loop in Java using the parameters you entered (i, 0, and a.length). Try experimenting with different iterators, start, and end indexes. For example:

```
for j from i to 64
```

Entering "demo1" or "demo3" will switch the demo accordingly. Here are some other input strings to try:

```
create new car named my car

function private static int add two
numbers params int a int b
```

Currently, the only implemented functions are *for*, *function*, and *create*.

6 Conclusion

Spew is an application designed to assist all kinds of programmers, regardless of language specialty and programming ability. Although Spew at this point is in its infancy, we hope that the prospect of having a hands-free, cross-language interface for programming is an exciting one to hear about. We plan on continuing to develop Spew in our free time and are excited to implement a working phrase structure set in the near future.

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.

Bird, Steven, et al. 2015. "Natural Language Processing with Python." *Natural Language Toolkit (NLTK)*. http://www.nltk.org/book/ch08.html

Download Spew:
https://github.com/michaeltangelo/spew