Stochastic Parody: Language Specifications

Matt Laws and Leah Williams

December 17, 2023

1 Introduction

Have you ever wanted to make a parody of your favorite song but don't have the musical talent or time to do so? Our language will make it possible for individuals to create "new" music by taking existing song and generating new lyrics to fit the existing melody. Our language allows users to select which words of a song will get changed, and then outputs a new song with the requested changes. These words can be changed completely randomly or to a selected set of words / sentiment that the user specifies. Our language will allow people to more easily create new parodies of their favorite songs, and is a fun tool that will help people enjoy music in a new and cool way.

Being able to write parody lyrics by hand that match the rhythm of the original song is incredibly difficult / time consuming and only talented song writers can do a good job. But we believe that music is a way to convey emotions and experiences that go beyond what can simply be written on the page and it is important provide as many people as possible the opportunity to interact with it on all levels – including parodies. Our language provides these skill to the average person. Using our language people can create songs / parodies with completely new words without having to spend the tireless hours generating an intricate cadence or matching an existing one. Making music creation more accessible is important because it will help grow peoples creativity and parodies are a interesting way to experience music.

Using a programming language implementation improves on a standard application level implementation because of the combination of dexterity and simplicity that the language provides. Users can simple paste in their favorite song and using our syntax specify the details of the translation. A standard application could accomplish the same task but would require much more overhead from the user and the programmer. Stochastic Parody provides an easy to use and forgiving syntax that allows users simple and precise control over how a parody is generated.

2 Design Principles

Song writing is an emotional process, and generally expand on the ways a certain experience has affected the writer. But, a computer cannot convey these emotions in the same way people can, so our language allows users to add a sentiment and keywords as inputs so that human emotion can remain a part of the creation. This allows users to generate songs that fit an emotion or experience they are trying to convey rather than what a computer decides. Furthermore the language has forgiving syntax to enable users to simply paste in the lyrics of their favorite song as the base of their parody. This is an important feature because we want it to be easy to get started with our language. From there users can add our annotations to denote what parts of the program should be translated / rhymed. This allows users to keep elements of a song that they like and translate others. Since these annotations are made with simple ! or $ symbols, it is very easy for the user to implement changes. To further simplify things for the user, we provide a type of variable called a section. A section allows users to save a series of one or more lines so that they don't have to retyped each time if they repeat. A common occurrence of this is in songs with choruses. Using a section a user can save the chorus as section and then simply call it whenever the chorus occurs.

The language assumes several heuristics about how a parody should be created that help generate more clear and understandable text. The first heuristic is that we defaults all words to their main pronunciation. We use this to generate the most accurate matches of words. For example Hello has both of the following pronunciations listed in the Lenzo (2014)'s CMU pronouncing dictionary: HH EH0 L OW1 and HH AH0 L OW1. Most people would use the pronunciation HH EH0 L OW1, so that is used as the pronunciation for hello. Another heuristic we use is part of speech matching. We assume that in order for the output to make sense, when replacing a word we should replace it with a word that is in the same part of speech. These assumptions are implemented in pre-processing (steps are outlined in detail in the code/preproc/ folder) and in the dictionary read. The part of speech information was parsed from Fellbaum and Tengi (2010)'s data. Additionally, to enforce the use of known words we define a list of common words that we attempt to translate to before entering the whole dictionary, this helps constrain the output words to a standard vocabulary of 3000 words instead of all 130,000 stored in the CMU dictionary. Our list of common words is provided by AG (2023). Lastly, we posit a definition of rhyme and syllable. Syllable matching – translation – is defined by matching both the syllable count and which syllables the stresses fall on. This ensures a seamless insertion into the base song. We define rhyme as the sound and the stress of the last syllable. Overall these heuristics make for better and more interesting translations.

The last principle is our notion of randomness. All words that are translated are transformed into to a pseudo-random word that fits the specified translation (it still looks at the keywords, sentiments, and common words first). This randomness is important because it allows a user to generate a myriad of different parodies of a single format, which allows users to experiment with and fine tune their parody. Randomness is a cornerstone of our language: hence the name Stochastic Parody.

3 Examples

Below are several example implementations of our language. Code for our language is written according to the syntax outline in section 5. We recommend storing code in a .song file for organization, but it is not required. The program is run by calling dotnet run <file.song>. The evaluator will then print out a text block that represents the output of the program. This can be saved to a file by redirecting the standard out as such: dotnet run infile.song > outfile. Futhermore, the actually process of the translation can be shown by passing v or verbose as a second argument. As stated the whole point of the language is to generate songs that fit the same rhythm of an existing song. The examples below are located in the examples folder and can be run by calling dotnet run examples/example-#.song. If you want to test how well we achieve that goal try to sing some of the examples, we chose some of our favorite songs to demonstrate!

3.1 Example 1

```
example-1.song:

{SENTIMENT} happy
{KEYWORDS} williams, computer, program, language

Just a $small !town girl
Livin' in a !lonely world
She took the !midnight train going anywhere
Just a $city boy
Born and raised in South !Detroit
He took the !midnight train going anywhere


$ dotnet run examples/example-1.song

Output:
"just a tall beach girl
living in a super world
she took the income train going anywhere
just a party boy
born and raised in south success
he took the income train going anywhere
```

```
"
```

Note that all the words preceded by an exclamation mark or a dollar sign have been translated. Further, words such as beach, super and party are all from the "happy" sentiment. Lastly, our program file ends with a blank line, this is **essential** for the code to run correctly – this is because all lines end with a newline character by definition.

3.2 Example 2

```
example-2.song:

I spent !twenty years trying to get out of this $place
I was !looking for something I couldn't $replace
I was running away from the only !thing I've ever $known
Like a !blind dog without a $bone
I was a !gypsy lost in the twilight $zone
I hijacked a rainbow and crashed into a pot of $gold


$ dotnet run examples/example-2.song

Output:
Warning: no match was found for "replace"
"i spent study years trying to get out of this race
i was soccer for something i couldn't replace
i was running away from the only joint i've ever blown
like a fence dog without a zone
i was a dealer lost in the twilight tone
i hijacked a rainbow and crashed into a pot of field
"
```

Note that this output gives a warning that "replace" could not be matched with another word that rhymes, and is of the same part of speech, so it is not translated. In order to remove the warning the user could remove the dollar sign or change it to a exclamation point. As in example one the rest of the designated words were translated. Finally, we were able to omit the sentiment and keyword field without error if we didn't want to include them.

3.3 Example 3

```
example-3.song:

{KEYWORDS} milk toast eggs waffle

lineOne = [Ain't no sunshine !when !she's $gone
]

lineTwo=[
    Anytime she goes away
]

*lineOne
It's not !warm when she's away
*lineOne
And she's always gone too $long
*lineTwo
Wonder this time where she's $gone
Wonder if she's gone to $stay
*lineOne
And this !house just ain't no $home
*lineTwo


$ dotnet run examples/example-3.song

Output:
"ain't no sunshine eggs me praun
it's not far when she's away
ain't no sunshine eggs me praun
and she's always gone too strong
anytime she goes away
wonder this time where she's praun
wonder if she's gone to tay
ain't no sunshine eggs me praun
and this toast just ain't no nome
anytime she goes away
"
```

Note that we used sections (variables) in order to avoid retyping certain lines that appeared multiple times. Further note that each occurrence of that line used the same translation. In fact when any word is translated it will continue to translate to the same thing anytime we ask it to translate throughout the song. Further, the sections can be declared with various formats as seen in the difference between lineOne and lineTwo; however there must be a newline character before the close brace – this is because a section is just a list of lines and lines all end in a newline character by definition.

3.4 Example 4

```
example-verbose.song:

{KEYWORDS} ponder
$wonder


$ dotnet run examples/exampe-verbose.song verbose

Output:
Word List: ["ponder"]
Assembing Dictionary
Dictionary Complete
"wonder" --> "ponder" from "Priority"
Matched Features: Emphasis: "10"
Part of Speech: "verb"
Rhyme: "ER0"

"ponder
"
```

Note the output now contains information about how words were translated. Since the only word translated was wonder, you can see that it was translated to ponder because the emphasis, part of speech and rhyme matched. Further because ponder was a keyword it was a priority match.

4 Language Concepts

The one of the main goals of this language was to make parody making more accessible to people without much prior knowledge of music. However, a basic understanding of song structure could help a user get the most out of our programming language by better selecting which words they want to translate. However it is simple to learn: for example, translating filler words does not always have as interesting of an effect as translating like key nouns. Furthermore, an understanding of the format that parodies normally take could help the user make more precise annotations in their code to generate better parodies. Generally speaking, changing words that represent the meaning of the line make for better results. For example in the line: "A thousand miles seems pretty far" changing words such as "thousand", "miles" and "far" would likely generate more interesting output then changing words like "pretty", "a", and "seems."

While it is not essential to understand rhyme and rhythm to write a program in Stochastic Parody, a basic understanding would help the user better understand the evaluation of the language. The language uses the CMU dictionary (Lenzo, 2014) to complete translations, we further modified this dictionary to include parts of speech. A line in this dictionary look like the following:

```
PARODY noun P EH1 R AH0 D IY0
```

This entry tells us the word, part of speech and pronunciation. The pronunciation is given by the ARPAbet symbol set – a set of phonemes – along with the stress of the words. These features are key to translating words and matching rhyme as described in section 2. The inner-workings of a translation can be displayed by passing v or verbose as a second argument to the program call as seen in section 3.4.

5 Formal Syntax

Below is a formal BNF syntax for our grammar. Our language is very white space forgiving, it allows for leading / trailing spaces at the start and end of lines along with spaces and / or commas separating words. It allows for multiple newlines before and after each line; however, it requires at least 1 newline character after every feature of the grammar. This includes the last line – thus the last line of the program file should be empty to account for the last line of the code's newline character. Below we outline a method of white space forgiveness that we find optimal for coding in this language but note that other white space combination **may** be possible; however, we guarantee the below to be correct.

5.1 Backus–Naur Form

```
<Grammar> ::= <Sentiment> + <Keywords> + <Body>⁺
<Sentiment> ::= "{SENTIMENT}" + ␣ + <Word> + '\n'
             | ϵ
<Keywords> ::= "{KEYWORDS}" + (<Sep> + <Word>)⁺ + '\n'
            | ϵ
<Body> ::= <Section Instance>
        | <Section Declaration>
        | <Line>
<Section Instance> ::= '*' + <Section Name> + '\n'
<Section Declaration> ::= <word> + "=" + "[" + <Line>⁺ + "]" + '\n'
<Section Name> ::= <word> ∈ {Declared Sections}
<Line> ::= (<Translation Unit> + <Sep>)* + <Translation Unit> + '\n'
<Translation Unit> ::= <Translate Word>
                    | <Rhyme Word>
                    | <Contraction Word>
<Translate Word> ::= '!' + <Contraction Word>
<Rhyme Word> ::= '$' + <Contraction Word>
<Contraction Word> ::= (<Letter> | ''')⁺ ∈ {CMU Pronunciation Dictionary}
<Word> ::= <Letter>⁺ ∈ {CMU Pronunciation Dictionary}
```

```
    <Letter> ::= {a..z} ∪ {A..Z}
    <Sep> ::= ␣* + (',' | ϵ) + ␣+
```

5.2 Notes

The set CMU Pronunciation Dictionary is all the entries in the CMU Pronunciation Dictionary – a dictionary that contains over 130,000 words and their pronunciations (Lenzo, 2014). The pronunciations detail both the stress and the sound of each part of the word according to the ARPAbet symbol set (Lenzo, 2014). Words that are translated must be contained in the dictionary or the code does not know how to accurately translate it. The set Declared Sections refers to all of the names of sections that have been declared, where the name of a section is the word before the equal sign.

6 Semantics

6.1 Sentiment

Abstract Type: *Sentiment*
F# Type: string
Syntax Example:

```
<SENTIMENT> happy
```

Meaning: Sentiment is a string that represents the sentiment that the translation will try to mimic. For example if we provide the sentiment happy a list of words that correspond to happy will be added to the preferred list of words for translation.
Notes: Currently the only sentiments supported are happy, funny, angry, and sad. If included, this field should be the first line of the program, but can be left out.

6.2 Keywords

Abstract Type: *Keywords*
F# Type: string List
Syntax Example:

```
<KEYWORDS> Williams, cow, computer, science, programming
```

Meaning: Keywords is a list of strings that represents keywords that translation will try to include. These words, in addition to the sentiment words make up a list of preferred words that are the first choice to translate if the features match. For example if we provide the keywords Williams, cow, computer, science, and programming these words would be added to the preferred list of words for translation.
Notes: If included this field should be directly after the sentiment line, but can be left out.

6.3 TranslationUnit

Abstract Type: *TranslationUnit*
F# Type: Record of word: string; translate: bool; rhyme: bool
Syntax Example: N/A: this is a private type
Meaning: A TranslationUnit keeps track of the relevant data of a given word in the input. The TranslationUnit stores the word, along with flags for if the word should be translated and if the word should be rhymed. They are used to tell the evaluator what kind of processing it should do with a given word i.e. if / how it should be translated. If a word is translated / rhymed it is changed to a word that is the same part of speech for better flow of the new lyrics.
Notes: This is a private type only used within the language, and is not referenced by the user.

6.4 Line

Abstract Type: *Line*
F# Type: TranslationUnit List
Syntax Example:

```
Just a $small !town girl, Livin' in a !lonely $world
```

Meaning: A line is a list of words to translate along with instruction on how to complete that translation (TranslationUnit). The parser equates a preceding ! to denote translation without guaranteeing a rhyming match and a preceding $ denotes a translating to a rhyming word. This example represents a line of the initial song, *Don't Stop Believin'* and will be translated into a line of the new song.
Notes: Lines endings are delimited by newline character, there is no max size of a given line and words should be separated by a single comma or spaces or some combination of the two.

6.5 Sections

Sections are the language's notion of variables. There are two parts of a section, a declaration and an instance.

6.5.1 Section Declaration

Abstract Type: *Section*
F# Type: string ∗ (Line List)
Syntax Example:

```
chorus = [
    line
    ...
    line
]
```

Meaning: A section is a form of variable in the language. Sections hold one or more line so that they do not need to be retyped over and over. Sections are declared as detailed above and referenced as detailed in the section instance section, 6.5.2. When a section is referenced the lines that are stored within the section variable, what you declare using the above syntax, are inserted into the lyrics at that position.
Notes: Sections are mutable. That is if you define a section later with the same name as a previously defined section will overwrite the meaning of that section to be the new declaration. Additionally sections must have a newline before the close bracket.

6.5.2 Section Instance

Abstract Type: *Section_Instance*
F# Type: string
Syntax Example:

```
*chorus
```

Meaning: A section instance is how to use a declared section. A section instance will insert the lines specified by the section declaration into the lyrics with the specified translation instructions.
Notes: Section instances must be used after a corresponding section declaration as detailed in 6.5.1.

6.6 F# Algebraic Data Type

```
    type TranslationUnit = {word: string; translate : bool; rhyme: bool}

    type Grammar =
    | Sentiment of string
    | Keywords of string List
    | Line of TranslationUnit List
    | Section of string * (Grammar List)
    | Section_Instance of string
```

6.7 Notes

Our primitives are lines from an existing song that a user can paste in (or write) that will set the tune / rhyme scheme of output song. Any line of code in the program not preceded by an explicit label will be interpreted as a line. The other primitives are keywords – words that are we attempt to translate to first –, and sentiment, a word that describes the sentiment of the new song to generate. All of these rely on a word primitive. A word is just a F# alpha string with one caveat: it must be in the CMU pronunciation dictionary as described in section 5.2.

Some songs generate issues with when they have words like "waitin’" which are not contained in the CMU Dictionary. However, the language does translation of common abbreviations of words automatically. For example "waitin’" would be converted to "waiting" which does exist in the dictionary.

The actions of the language are sequential and all elements are delimited by a newline character, for instance: program line n precedes program line n+1 in defining the rhythm of the new song. Furthermore, within each line, we use spaces to delimit each word in the line of song and they are "combined" together to form 1 line by all being on the same file-line of the input. The language also provides sections (variables) that allows a user to save a series of lines and insert them again later. A section combines 1 or more lines into a compact structure for the user to call instead of retyping the lines every time.

7 Future Work

We believe that Stochastic Parody is sufficiently complete to be used by whoever is interested. However, there are several additional features that we would like to implement in the future. We want to make the parser more generous on any inputs that may be pasted in from song lyrics including numbers and other special characters. Furthermore we would like to develop a way to match unknown words to potential translations in order to translate words that are not contained in the CMU dictionary. Finally we would like to improve our sentiment system. Currently we only support several sentiments that are aggregated prior to running the code. We would like to expand our system to be able to ad hoc generate a list of words that fit any given sentiment. With these future changes Stochastic Parody will create even better rhymes than in its current state.

8 Conclusion

We had a great time developing this language, but just as importantly have had fun using this language. Many times we have read out lines generated by Stochastic Parody to our friends for everyone to admire its greatness. If you are a user of our language we hope you enjoy!

References

Signum International AG, 2023. URL https://www.ef.edu/english-resources/english-vocabulary/top-3000-words/.

Christiane Fellbaum and Randee Tengi. Wordnet, 2010. URL https://wordnet.princeton.edu/download/current-version.

Kevin Lenzo, 2014. URL http://www.speech.cs.cmu.edu/cgi-bin/cmudict.