Synthtactic Sugar

Final Report

Hadey Black, Evan Hughes, Carl Eadler

March 2015

1 Introduction

In this project we implemented a simple language for encoding music called IML (Intermediate Music Language), an interpreter from English to IML using the Cambridge dictionary API, and an easy to use sound synthesis platform to play the results in Pure Data (a visual programming language for sound synthesis). Our idea was to generate simple melodies from bodies of text which capture the the rhythm, mood and phrasing of that text.

2 Python

2.1 Why Python?

Synthtactic Sugar was written largely in Python. We made use of Python 2.7 and a number of libraries, the most significant being the Python Requests library which makes dealing with HTTP much more manageable.

Python was chosen because we were interested in using a language we weren't all familiar with, and because it has a low learning curve, allowing us to jump right in and start programming with little knowledge of the language. We largely credit the website learnxinyminutes.com as a major resource for learning Python. Ultimately we wrote approximately 430 lines of code for this project.

2.2 Functionality

The functionality is broken down into a number of Python files. First and foremost is `parser.py` which contains the main function. This file handles the input parsing of English text, as well as the API calls. There is also `XMLParse.py`, linked to `parser.py`, which does the parsing of the XML received as a response from the dictionary API calls. Finally, also linked to `parser.py`, is `toIML.py`, which takes

information provided by `parser.py` and `XMLParse.py`, and creates a text file containing IML. These IML files are handled then by `IMLtoPD.py`, which takes the IML files and creates an input file which can be read by Pure Data.
We will go into more detail in the following sections.

3 API Calls

3.1 Learning about Web APIs

Learning how to use web API calls was a new experience for the entire team, moreso than learning Python. HTTP is an unfamiliar realm filled with all sorts of crazy headers, parameters, request methods, and status codes. Thus, we had to familiarize ourselves with request methods `GET, POST, HEAD, PUT`, and others, although in the end we only needed `GET` after filling out a header. Thankfully, the Python Requests library made all of this quite manageable for us.

3.2 Choosing a Dictionary API

Initially, we used the Merriam Webster Dictionary API, which was extremely simple. We created a URL that had a specific substring for the word we wanted to look up, which we would replace as necessary, and made a simple `GET` call. We would check the response status code, and if it was 200, we would have a good response which we would save and look into as necessary. However, the Merriam Webster API was extremely bare, with nothing to offer on their website other than a 6 line PHP tutorial and a barely readable explanation of their XML contents. The responses did not have the information on the words that we wanted and needed to progress.

Thus, we went back to researching the best suited dictionary API for our purposes, and found the Cambridge Dictionary. While the Cambridge Dictionary API had no descriptions for their XML tags, they had thorough specifications on using their API.

3.3 Using the API

We used the Cambridge API `searchFirst` method, which returns the first match on whatever word we passed to it. We pass our API key as the `accessKey` value of the HTTP header, the `dictCode` in the URL, and the format and word as parameters. The API key is how the Cambridge Dictionary knows who is accessing the information and can track how many requests we make. We had to apply for various API keys, and wait a few days before we were approved and given a key to use. The `dictCode` specifies which dictionary we want to do our lookups in. We used the American-English dictionary. The format parameter specifies how we want our information returned to us, whether it be `JSON` or `XML` (we used `XML`). Lastly the word parameter is simply the word we want to look up.

Though it initially took us a long while to figure out, the Requests library streamlines the entire process for you making it much easier. You fill out your `params` and `header` in Python dictionary structures, and pass them to the requests `GET` call, as such:

```
cambridge_header = {'accessKey' : cb_api_key}
payload = {'format' : 'xml' , 'q' : word}
response = requests.get(url,params=payload, headers=cambridge_header)
```

Requests takes care of all the background work, preventing you from having to use the supposedly unnecessarily complicated and broken standard `urllib2` Python module. If the response came back with a status code of 200, we could check the response text and pass it on to the XML parser.

4 XML, Regular Expressions, and English

4.1 What is XML?

XML is a popular markup language for transferring large amounts of data in an easily-decoded way. Data in an XML document is enclosed by a set of angle-bracketed text called tags that describe what kind of data it should represent. For example, in our project, the XML files that we were working with used the tags `<title>` and `<ipa>`. Sample pieces of data enclosed by tags might look like the following:

```
<title>phenomenal<\/title> and <ipa>fnmn<sup><\/sup>l<\/ipa>
```

One of our major issues in development, which seasoned experts of using XML might have recognized, is that theres a slight difference from proper XML here; the closing tags `<\/title>` and `<\/ipa>` dont end like proper XML tags. Indeed, the backslashes should not be there. XML parsers from the python library refused to deal with parsing these corrupted files. and therefore, we ended up having to write our own set of regular expressions to extract data from these files.

4.2 Regular Expressions

Regular expressions (RegEx) are a domain-specific language that is used to search strings of data, usually alphanumeric characters, for particular patterns. For example, in our program, we wrote a general function `getTag` which takes a tag as a string and a definition (string of XML that describes a dictionary entry for a word) and returns data enclosed by the given tag, if it exists. The following is a sample of a regular expression command from our program;

```
splits = re.split('<\%s>'\%tag, definition)
```

This function specifically makes an array of disjoint substrings of the string `dictionary`, divided wherever it finds the angle bracketed text `'<%s>' %tag`.(This strange notation is string formatting in python. This code specifically replaces the `%s` in the string with whatever data is in the tag variable). Regular expressions have many powerful

features, such as using sets like `[a-zA-Z0-9]` which matches any lower-case, upper-case, or numeric character. Using features like this and others, users can easily make short expressions that create powerful parsers.

4.3 English and IPA Spelling

After running our data through our parser, we were left with data from the `<ipa>` tag like fɪ'nam·ɘ·nəl (The international phonetic alphabet spelling of phenomenal). After retrieving this data, we needed to find out how many syllables were contained in each word. In English, most words contain one syllable per vowel sound. Phenomenal, for example, contains 4 vowels, `[ɪ, a, ɘ, ə]`, each of which contribute one syllable to the word. Using this information, we created a function called `syllableCount` which takes the IPA spelling of a word as a unicode string and a list of tuples of vowel patterns matched with how many syllables they produce, and parses the word, counting how many syllables are contained in it.

5 IML

5.1 What is IML?

IML stands for Intermediate Music Language. It is a simple, difficult to read, yet easy to parse language for encoding basic musical information. The language is structured as a list of notes separated by commas where each note is of the form $([p_1 : p_2 : ...p_n]/[d_1 : d_2 : ...d_m])$. Each p_i is a pitch (in MIDI) which is played for d_1 milliseconds and then d_2 milliseconds and so on. To clarify, a simple example of a note sequence in IML might be $([47]/[800]), ([50]/[400 : 400])$. This example plays the note 47 for 800 milliseconds and then the note 50 for 400 milliseconds twice.

5.2 Generating and Parsing IML

Recall that when running `synthtactic.sh` the user specifies the key and scale of the music to be generated. Based on this input we generate a dictionary of the form $\{n : midi_n\}$ where n is the scale value from 1 to 8 and $midi_n$ is the midi value of n.

Via XML parsing we generate a list of the form $[(s_w, pos_w, c)]$ where s_w is the number of syllables in word w, pos_w is w's part of speech (noun, verb etc...) and c is the character directly following w (usually a blank space, but sometimes punctuation). The list has a tuple of this form for each word in the text file and preserves the order said file. Now, for each of these tuples (s_w, pos_w, c) we generate a tuple in IML $([p]/[d_1 : d_2 : ...d_m])$ (our current implementation does not generate multiple pitches at a time, though this an obvious next step in improving our project).

To generate the pitch value p we choose based on the value of pos_w. For nouns we pick p randomly from $n = 1, 5, 8$. Other parts of speech have similar mapping schemes. As

of now these mappings feel somewhat random and an obvious improvement would be to extend the mappings to generate chords. This would allow for the mood of a word to be expressed immediately, instead of relying on the surrounding context notes to create a mood.

To generate note durations for a word w we mainly look at the values of s_w and c. Initially, we assume every word will take up the space of 1 beat. We subdivide this beat into s_w parts (note plays s_w times over the course of 1 beat). Now, if the word is a conjunction or preposition we divide that beat in half. Finally, if $c =$ "," we multiply the beat by 2 and if $c =$ "." we multiply the beat by 4.

6 Pure Data

`Pure Data` is an open source visual programming language for sound synthesis. We used it to build our instruments which play the music our program produces. We also used it to sequence the music using `Pure Data's qlist` object.

6.1 Instruments

The instruments are basic additive synthesizers consisting of 8 *sin* waves generated by the `osc` object. The first *sin* wave is the fundamental frequency f while the others are ascending multiples of f which are played at half the volume of the frequency directly below. Additionally, we built a simple drum kit using `osc` and `noise` which is a white noise generator. Both of these instruments could be improved upon and we plan to do this in the future.

6.2 Sequencing

`Pure Data` handles sequencing using it's `qlist` object which reads a file in which each line is of the form

```
time1 obj1 value1;
time2 obj2 value2;
...
```

where `value2` is sent to `obj2` `time2` after event 1.

We utilize this feature by sending pitchs (values) to instruments (objs) at time relative to the prior event in milliseconds.

7 Conclusion

Our project uses a variety of languages including two primary programming languages (`Python` and `Pure Data`), various domain specific languages (`XML`, `IML`, and `RegEx`), a data protocol (the `Cambridge Dictionary API`), and a natural language (`English`). Each language has its own strengths, weaknesses, and nuances and our program fuses all of them together in its execution. It was up to us to learn how each language functions and how it is formatted. Overall, we learned a great breadth of languages for this project, some of them greatly in depth.

As it is, we are pleased with what we were able to accomplish within the time of the class. To improve upon what we've built, we could potentially implement more interesting methods of choosing pitches using a more dynamic approach - possibly by choosing chords. Additionally, we could implement a web based user interface using `JavaScript`. As a group we plan to implement these ideas in the near future.