21M.380 Sonic System Project Report: Generative Context-free Grammars

Introduction

For my final project, I made a system for defining context-free grammars (CFGs) and expanding a sequence of terminals into non-terminals using randomized productions. This contrasts with the more common activity of parsing a string of already-generated terminals into higher-level non-terminals, which is essentially the reverse process. A sequence of terminals can then be processed to produce some sort of output, such as a sound or a song. Using this system, complex definitions can be easily created as a series of productions, each of which maps a symbol to one or more probabilistically-weighted expansions, in order to create complex randomized, but structured, behavior.

Motivation

As a computer science student and programmer, I am interested in the languages used to represent digital information. This includes general-purpose programming languages such as Python or Java, but also domain-specific languages (DSLs) created for a particular use, such as AthenaCL. These examples are all at least somewhat imperative in nature; that is, they take the form of a sequence of commands or instructions to execute. For example, in AthenaCL, one can sequentially specify various parameters of a musical piece, issue a command to render the piece, then continue to apply further modifications. The opposite of imperative programming is declarative programming: instead of expressing what to do step-by-step, the programmer specifies what the finished product should be, and whatever is interpreting or compiling the program must figure out how to accomplish that. AthenaCL can be considered declarative as well, as one defines various types of parameterized musical objects, and then the interpreter automatically generates a sound represented by these objects; the user doesn't need to know how to actually implement a random walk or how to create midi events, for example.

Many prefer programming in a declarative style, myself included, because it's usually easier to symbolically define what you want than to create the specific commands to generate that definition. Unfortunately it's usually easier to create imperative languages, since the information is represented in less-complex components and is thus easier to render (basically, someone has to do the hard

part: either the programmer or the interpreter/compiler converting the program into something a computer can execute). DSLs, however, often lend themselves to a more declarative style than broader programming languages, as the information that can be represented is constrained and thus it is simpler to automatically render what the user models using the language, since there are fewer possibilities.

The generative CFG engine I created is essentially a tool for creating simple DSLs, with the added twist of randomized definitions. The random aspect makes the system suited for creative pursuits, such as music generation, that don't require that a program always produce the same "correct" output (in fact, the opposite is often preferable when creating computer music), although a more conventional DSL can be specified by only specifying one expansion for each symbol. I hope the system can simplify the creation of complex musical generation by allowing users to create a generative CFG (or several) and then use it to quickly define and generate different types of music.

Implementation

At the core of this system is the file gen.py. This script takes a string of space-delimited symbols and a grammar file, recursively expands each symbol according to the provided grammar until no more expansions are possible (i.e., none of the symbols have expansions in the grammar), and outputs the resulting sequence of terminals.

In general terms, a grammar in my system is a collection of productions, which map a symbol to one or more expansions. An expansion is a sequence of one or more symbols. Each expansion is annotated with a weight, reflecting how likely it is be chosen among the possible expansions. Grammars in my system are represented as YAML files (YAML is a data serialization format known for its readability, see www.yaml.com). The grammar is represented as a hash, with symbols as keys and a hash of expansions as values. Each expansion hash is keyed by single space-delimited string of symbols (the expansion itself) with an integer weight as a value. As an example, here is a grammar I made to represent a waveform, where the waveform is modeled as successive additions and multiplications of basic waveforms:

```
sound:
  init any+ : 1

init:
 '+sin([220-880])' : 2
 '+saw([220-880])' : 1
 '+square([220-880])' : 5

any:
 '+sin([220-880])' : 2
 '+saw([220-880])' : 1
 '+square([220-880])' : 1
```

```
 '*sin([220-880])' : 5
 '*saw([220-880])' : 3
 '*square([220-880])' : 3
```

If a symbols is followed by a *, it means it can be expanded into zero or more instances of that symbol. Using a + instead signifies that the symbol can be expanded into one or more instances. A range in brackets (like [220-880] above) is expanded into an integer randomly chosen from within that range (inclusive).

Grammars can contain recursive productions, but each production must contain at least non-recursive expansion since the script expands the sequence until it contains only terminals (symbols without expansions).

Here is an example of gen.py using this grammar, which is saved in sound.yaml:

```
$ python gen.py "sound" sound.yaml
['sound']
['init', 'any+']
['+square([220-880])', 'any', 'any*']
['+square(349)', '*sin([220-880])', 'any', 'any*']
['+square(349)', '*sin(426)', '+sin([220-880])']
['+square(349)', '*sin(426)', '+sin(645)']
```

As seen above, gen.py prints the sequence produced with successive expansions. The first line is the initial sequence given, and the last line is the final sequence of non-terminals produced.

Of course, this sequence alone is basically useless unless we have a way of converting it to the waveform it represents, in this case a square wave convolved with a sinewave and then added to another sinewave (the numbers are the frequencies). I wrote another script, wavgen.py, that takes sequences of this form and creates a wav file.

Grammars can also be composed, meaning that the output produced by applying one grammar to an initial sequence is used as input to another. When multiple grammar files are supplied as arguments to gen.py, it composes the grammar, calling the first one listed first with the initial sequence, passing its output to the second, etc.

For example, song.yaml defines a simple definition of a song:

```
song:
- note note note note+ : 1

note:
- C : 1
- D : 1
```

```
- E : 1
- F : 1
- G : 1
- A : 1
- B : 1
```

notes.yaml provides productions for converting the notes to waveforms:

```
# frequencies from http://ptolemy.eecs.berkeley.edu/eecs20/week8/scale.html
A:  {sin(440) : 1}
Bb: {sin(466) : 1}
B:  {sin(494) : 1}
C:  {sin(523) : 1}
Db: {sin(554) : 1}
D:  {sin(587) : 1}
Eb: {sin(622) : 1}
E:  {sin(659) : 1}
F:  {sin(698) : 1}
Gb: {sin(740) : 1}
G:  {sin(784) : 1}
Ab: {sin(831) : 1}
A2: {sin(880) : 1}
```

Composing these two grammars gives us the following:

```
$ python gen.py "song" song.yaml notes.yaml
['song']
['note', 'note', 'note', 'note+']
['F', 'D', 'D', 'note', 'note*']
['F', 'D', 'D', 'C', 'note', 'note*']
['F', 'D', 'D', 'C', 'D']

['F', 'D', 'D', 'C', 'D']
['sin(698)', 'sin(587)', 'sin(587)', 'sin(523)', 'sin(587)']
```

gen.py first expanded song using the productions in song.yaml, giving us a sequence of notes in the C major scale. This sequence was in turn expanded using notes.yaml, giving us the waveforms of the notes.

If we want to hear this song, we cannot use wavgen.py, as it combines waveforms into a single waveform through addition or multiplication instead of sequencing (we could play all the notes at the same time with wavgen.py though, as it automatically converts symbols of the form "wav(freq)" to "+wav(freq)"). Instead we would like to sequence the notes. I wrote yet another

script, wavsequencer.py, to perform this task. wavsequencer.py takes an initial sequence and an arbitrary number of grammar files, like gen.py and wavgen.py, expands the initial sequence by composing the grammars, and then uses wavgen.py to render each symbol in the new sequence.

Of course, these are just examples of what can be produced. New grammars can be written to produce more complex songs and sounds by using the productions defined in sounds.yaml and notes.yaml, and new scripts can be written to interpret new types of symbols (or reinterpret existing symbols, as in the case of wavsequencer.py).

Reflections

I really like the idea of using generative CFGs to produce music. Just playing with the few simple scripts and grammars I've produced, I've been able to create some interesting (if insubstantial) sequences, and it's fun to just run the same command over and over again and listen to the differing results. Using this system, it's very easy to build upon existing tools I've created, which I value in any kind of production framework (music, code, or otherwise).

I think I could have explored the system much more in depth though. I was extremely crunched for time this semester, so what I have so far is really more of a proof-of-concept than a tool that could actually be used. I would like to expand the CFG format, allowing for more expressive productions (for example, adding parameters to symbols that can be symbolically referenced in other symbols, adding more regular-expression constructions, and maybe allowing multiple symbols on the left-hand-side of an expression, which would make the grammars more contextual). I also could have done a lot more with the mini-DSL I designed for use with wavgen.py; the script has the ability to produce waveforms of varying duration and amplitude in addition to frequency, but there is currently no way to express that in a symbol. I may continue to expand the system over the summer to see if I can get more out of it, because I think this concept could lead to some interesting and useful composition techniques.

21M.380 Music and Technology: Algorithmic and Generative Music

Spring 2010