Creating a library to aid in constructed language generation

Daniel James

April 2017

Chapter 1

Introduction

yod is a library, written in C#, designed to aid in the construction and generation of artificial languages. Artificial languages (also known as constructed languages or 'conlangs') are probably most publicly well known through the popularisation of 'fictional languages' - that is, languages designed for use in works of fiction, usually for adding depth to a fictional world. For example, J. R. R. Tolkien devised the languages of *Sindarin* and *Quenya* for use in his *Lord of the Rings* series. Other popular fictional languages that have reached mainstream awareness include *Star Trek*'s *Klingon* and *Game of Thrones*' *Dothraki*.

The creation of these languages has become an indispensable tool for writers to add depth and character to their worlds. However, constructed languages do not exist solely for creative uses. The language *Esperanto* was developed by its creator L. L. Zamenhof with the goal of being a global language that was easy for people to learn[1]. Now it is reported that as many as 63,000 people worldwide can speak Esperanto to some degree[2]. The existence of Esperanto and other so-called "international auxiliary languages" (languages designed to simplify complication between people of different countries and cultures) shows that the usefulness of constructed languages extends into real-world applications as well as artistic uses.

yod was developed with a focus on artistic languages, and so the goal was to generate a large variety of languages which could suit many different worlds and fictional civilisations. However, due to the hierarchy of rules *yod* uses to build languages, the languages are regular, in contrast to most natural languages, which means generated languages with certain features could be treated as auxiliary languages too.

As a library, *yod* provides interfaces to easily generate any part of a language, including phonology, orthography and grammar and sub-parts thereof. This works by having each class which represents a randomisable constituent exposing a `Generate()` method which creates a new instance of that class from scratch with procedurally generated properties. If necessary, this sometimes also means recursively generating that objects constituent members. Writing these generator methods in a way that takes into account certain restrictions or

statistics allows for much more realistic values for the object.

Merely generating a new language every time the library is used, however, would not allow for enough control over the language generation process for users who are more experienced with the concept of 'conlangs' and the construction thereof. It also does not allow for iterative language-building, wherein the user keeps features of a generated language that they like, but re-generates the rest of the features. This would allow a user to quickly and easily refine their language until it suited their use case.

To facilitate this, *yod* uses a data-driven approach which allows almost every class that can be randomly generated to also be serialised to and deserialised from external data files. These data files are written in JSON to allow for them to be altered or loaded by other tools, or if necessary, by hand. Provided is a basic command line interface to the library which can handle almost every step of the generation process. This is a simple example of a tool which can load and create the data files produced by the library, and utilise the library's API. Providing the language-generation tools as a library allow for more complex tools to grow which implement and interact with the code.

The library format also allows for software which might have a need for 'conlangs' to generate them directly and use them immediately, as opposed to generating one and then using the output files as input for the program. One such example could be in the case of a video game which requires a cohesive strategy for naming places, characters, flavour text etc. These entities could have their text generated on the fly, without any input from the game developer or writers. The *yod* library also allows the random number generator (which provides random values for all of the rest of the code) to be given a seed, meaning it will produce the same output every time. Thus, the objects in the aforementioned game could have different text every time the game was run, or the developer could choose a seed that gave output that suited them and keep that the same, without having to copy text from the output of the script into the game's script and data files.

talk about structure

Chapter 2

Phonology

The 'naïve' method of building words is by taking an alphabet (for example, the Latin alphabet) and concatenating random characters until the sequence reaches a random length between a minimum and maximum. For example, given $min = 3$ and $max = 8$ we can generate the following example text:

Figure 2.1: Random letters from the Latin alphabet

```
tqk qzsyjla msmnix jvxx wug sysrh cuepg snyow ptjo bcek
arjdubw pfwpt nabgzk jmq taphh zewll dmpr uvpmx sfpfk uuo
bdm vnjbq hahuj wstq kohvma irn fott axdut rlgg tawz wsol
wigom psqwd tnv vlzgt lbcikk bof msmyg zkqgubb veht ukaznqn
ixp rppfj eqllnko uyyp aot uowtn icv fgypx cenawnk hypq
rruh eosgrf wmakeg hhweua gnbfh mkpzi ebtwbv cjwrxw ucky
kqezcm ucme wmrk khsya llzbeqw uxwivpp pbao gkzu pda txdp
iwl gkmfqn uxeupe atjxy vyul
```

Several problems can be immediately seen with this generated text. First, many of the words are difficult to pronounce and unrealistic with regards to their consonant clusters - for example, words like `jvxx` and `rppfj` are unlikely to exist in any natural languages. Generating words in this manner will also not produce very much variation in languages, as each letter has an equal probability to be picked.

We also quickly run into the problem of representing language in text. Written languages are based on spoken languages, so generating a written language first without basing it on a spoken one will lead to an unrealistic language. Furthermore, it is hard to say how our generated words are pronounced - one can apply English pronunciations to some of the words (for example `tawz` becomes /tɔːz/ and `axdut` becomes /ˈæks.dʌt/), but this results in a very Anglo-centric phonology, as we are biased to only use phonemes that exist in our own language

cite

while pronouncing unknown words.

Because of these problems, it is obvious that merely stringing together random characters with no thought towards pronunciation is insufficient when it comes to creating realistic and varied languages. Therefore it is necessary to first create a phonology on which to base all of our language's words.

2.1 Phonetic inventory

In its simplest form, a phonology is a list of every sound that is included in a language. English phonology, for example, contains around 24 consonants (with more or less depending on dialect) and anywhere between 7 and 14 vowels, again depending heavily on dialect. The phonology for the 'Received Pronunciation' dialect of English can be seen in 2.1 and 2.2.

cite (WALS)

Table 2.1: Consonant inventory in English phonology

		Labial	**Dental, Alveolar**	**Post-alveolar**	**Palatal**	**Velar**	**Glottal**
Nasal		m	n			ŋ	
Plosive, Affricate		p / b	t / d	t͡ʃ / d͡ʒ		k / g	
Fricative	**Sibilant**		s / z	ʃ / ʒ			
	Non-sibilant	f / v	θ / ð			x	h
Approximant			l	r	j	w	

Table 2.2: Vowel inventory in English phonology (Received Pronunciation)

	Front	Central	Back
Close	i / ɪ		u / ʊ
Mid	e	ɜ / ə	ɔ
Open	æ	ʌ	ɑ / ɒ

Other languages have different phonemic inventories, with varying numbers of consonants and vowels. Some, for example, have as few as 3 vowels, or as many as 84 or more consonants, depending on the method of counting[3]. A very large factor in the variety of languages generated is the phonology, as it restricts the type of sounds which often has a profound impact on how it is perceived. Therefore the first step towards a completed language should be a randomly generated, unique phonology.

The International Phonetic Alphabet (IPA) is an alphabet created by the International Phonetic Association for phonetic representation of speech and language[4]. It contains (or aims to contain) distinct symbols for each unique sound possible to create that is part of a language. As such, it provides a perfect way to convey the "end result" of generation process, since it can describe precisely how every word in the language is pronounced. The IPA also includes

markers for syllable stress and other important factors which can be included in speech. In order to avoid the user from having to infer the pronunciation of a word from its orthography (how it is written), *yod* can produce an IPA transcription of its output, which shows exactly how to pronounce it without ambiguity.

However, the IPA can also be used as a basis for *creating* a phonology. It lists every sound possible in a language, so as a naïve approach would be to randomly choose sounds from this set to build words. This approach is similar to the approach we took before which involved choosing random letters from the Latin alphabet. However this time the words are being generated via sound directly rather than generating a written word, then trying to create a spoken representation of that.

The problems with taking the entire IPA as our set of possible sounds are twofold. First, it does not give us a unique phonology - instead the phonology is the same every single time, as its contents are not randomised. Second, if the phonology for the generated language contains *every* consonant and vowel from the IPA, then it will have many more phonemes than even the largest existing phonetic inventory. This is clearly unrealistic, and would be very difficult for anybody to learn. For artistic languages, this is a bit less of a concern but it is still preferable that the generated words be somewhat pronounceable and understandable, which gets less likely the more phonemes a speaker would have to discern from each other.

To create more realistic phonologies, the subset of sounds taken from the full phonetic alphabet can be restricted. If, for example, random phonemes from the set are formed into words of random length between 3 and 8, the following words could be produced:

Figure 2.2: Random subset of all phonemes

ɭθθɪ̄ ɕgd͡z ed͡zɻɭʊ ɭt͡ɪ̄gt͡ɪ̄χʊɻt͡ɪ̄ ɭʎ̝ɪ̄g̊ɪ̄ç d͡zħɖ͡zˌ ħħɻɪ̄eθ ed͡zçd͡z ʎ̝̊ʊd͡zeɭʊ ʊɭʊχħ

Although this method achieves variety of phonemes while keeping the total number of phonemes low - the phonology used to generate these words only contains 15 phonemes - a lot of the words are still unrealistic. Similar to figure 2.1, there are an abundance of large consonant clusters. Most words in the output lack a vowel and even those with vowels have extremely unrealistic consonant clusters.

Another problem with this generated output is the sequences /θθ/ and /ħħ/. The former contains adjacent voiceless dental fricatives, and the latter contains adjacent voiceless pharyngeal fricatives. Such sequences consisting of two of the same phonemes are unpronounceable and would most likely be realised as single instances of the phoneme (e.g. /θ/ and /ħ/). These problems show that this method of generation is still insufficiently realistic.

Is is important now to separate the phonology somewhat from the process

Figure 2.3: Shared properties of phonemes

- IPA symbol representation
- Sonority

of creating words. The problems just discussed are largely more to do with this word creation process, whereas there is still some work to do to improve the selection of individual phonemes for the phonology.

The first step is to create a distinction between the two different types of phonemes: consonants and vowels. Consonants and vowels have very different features, and play very different roles in the formation of words. Thus the phonology's set of consonants should be generated separately from its set of vowels.

Both consonants and vowels share the common features of phonemes shown in figure 2.3. Sonority is a numerical representation of how loud or sonorous the phoneme. Each phoneme also has a representative symbol in the International Phonetic Alphabet so they can be easily transcribed.

In addition to these shared properties, consonants also have the extra features shown in figure 2.4.

Figure 2.4: Unique properties of consonants

- Place of articulation
- Manner of articulation
- Phonation/Voicing
- Gemination

The place of articulation refers to the part of the vocal track that creates an obstruction of airflow. The manner of articulation describes *how* this obstruction affects the airflow - for example, "stops" or "plosives" completely block airflow, whereas "sibilants" and "fricatives" merely result in a turbulent airflow. The phonation of a consonant refers to whether, in producing the sound, the larynx affects the airflow through vibration of the vocal folds. Geminate consonants are consonants which are pronounced for a longer period of time, contrasting the usual shorter consonants. In 'stops' this is realised as a delayed release of the consonant, whereas other consonants with other types of articulation are merely prolonged.

Similarly to consonants, vowels can be distinguished by the features in figure 2.5.

Figure 2.5: Unique properties of vowels

1. Height
2. Backness
3. Roundedness
4. Length

A vowels characteristics are formed from the position of the tongue in the mouth and the rounding of the lips. The height of a vowel refers to the height of the tongue, and the backness is the forward/back position of the tongue. Roundedness is the state of the lips in pronouncing the vowel - either rounded or unrounded. This unrounded/rounded distinction can be seen in the difference between the two open mid-back vowels /ʌ/ and /ɔ/ (compare *shut* /ʃʌt/ and *short* /ʃɔːt/). The word *short* /ʃɔːt/ also shows a good example of a long vowel (although there is no long/short vowel distinction in most English dialects). cite?

In fact, both height and backness are defined by the formant frequency of the voice, which are produced by the tongue's position. A spoken vowel can be represented with two formants *F1 = height* and *F2 = backness*. Roundness also has an effect on these formants, usually being seen as a decrease in *F2* as well as a small decrease in *F1*.

It can be seen that both vowels and consonants have properties which relate to their length; these being consonants' gemination and vowels' length. Due to the similarities between these properties, they can be abstracted into the set of base phoneme properties as a 'Length' property. Thus we end up with our final set of properties being as follows:

Figure 2.6: Shared properties of phonemes, with length

- IPA symbol representation
- Sonority
- Length

This arrangement of properties lends itself extremely well to a simple data structure in C#. Consonants and vowels both translate trivially to classes, with their respective properties existing as enum members in those classes. For example, an enum `PlaceOfArticulation` can be created to represent the place of articulation of a consonants which would contain the set of values `Bilabial`, `Labiodental`, `Lingulabial`, `Dental` etc. Both the `Consonant` and `Vowel` classes are subclasses of a more general, abstract `Phoneme` class which contains only

Table 2.3: Vowel Quality Inventories[5]

Value	Representation
Small vowel inventory (2-4)	93
Average vowel inventory (5-6)	287
Large vowel inventory (7-14)	184
Total:	564

Table 2.4: Consonant Quality Inventories[6]

Value	Representation
Small (6-14)	89
Moderately small (15-18)	122
Average (19-25)	201
Moderately large (26-33)	94
Large (34 or more)	57
Total:	563

the properties that are shared between both of them. This allows the entire International Phonetic alphabet to be represented in code as a set of `Phoneme`s.

It is more useful, however, to distinguish between the set of vowels and the set of consonants. Thus the two classes `ConsonantCollection` and `VowelCollection` can act as lists of their respective type of phoneme, with a `PhonemeCollection` being formed out of a union of the two sets of phonemes. Each class's `Generate()` method can then be used to create a unique set of each type of phoneme.

Using statistics from the World Atlas of Language Structures, the realism of our phonology generation can be increased by roughly following the range distribution of numbers of vowels and consonants in natural languages. For vowels, the distribution is shown in figure 2.3. By using these numbers as weights for a weighted random algorithm, the vowel inventory in the generated phonology will be similar in size to natural languages' vowel inventories. The result of the weighted random algorithm is a range (minimum value and a maximum value). Using *yod*'s global random number generator, a value between these maximum and minimum can be chosen as the goal number of vowels. Then that amount of unique vowel sounds can be randomly chosen from the full set to complete the vowel inventory.

Similarly, figure 2.4 shows the range distribution of consonants in natural languages. The same weighted random algorithm can be used to get a range, then a random 'goal' value of consonants in that range.

For consonants, however, it does not need to be as simple as choosing random consonants from the set of all consonants in the IPA. There are approaches that can be used to pick consonants in a smarter way which will result in better consonant inventories.

Given a goal number of consonants that need to be added to the phonology, the aim is procedurally get as close to that number while taking into account

extra statistics for which consonants to pick. Adding and removing whole places or manners of articulation tends to be more natural than choosing consonants at random, so a way of modelling this would be to add places and manners of articulation until the intersection of them results in the right amount of consonants. The algorithm would therefore work something like this:

cite - only thing i know about this is that one Artifexian video

Algorithm 1 Consonant inventory generation algorithm

```
goal ← generated goal number of consonants
places ← ∅
manners ← ∅
consonants ← ∅
while |consonants| < goal do
    if random < 0.5 then
        places ← places ∪ {random place of articulation}
    else
        manners ← manners ∪ {random manner of articulation}
    end if
    consonants ← {x|x.place ∈ places ∧ x.manner ∈ manners}
end while
```

Generating a consonant inventory this way means that the resulting language will be more cohesive in its sounds, while still retaining a unique feel every time one is generated. This approach does not take into account the phonation of consonants, though, so more variety could still be added. Again, WALS provides statistics on voicing in natural languages. Figure 2.5 shows the distribution of languages which have a distinction between voiced and unvoiced fricatives and plosives.

Table 2.5: Voicing in Plosives and Fricatives[7]

Value	Representation
No voicing contrast	182
Voicing contrast in plosives alone	189
Voicing contrast in fricatives alone	38
Voicing contrast in both plosives and fricatives	158
Total:	567

The figure shows information about voicing in both fricatives and plosives. In order to get this information from the statistics into a form that can be used in code, the **Value** column can instead be seen as a tuple of two boolean values: "voicing contrast in plosives" and "voicing contrast in fricatives". With an object structure of `{fricative voicing contrast, plosive voicing contrast}` The values will then become `{false, false}`, `{false, true}`, `{true, false}` and `{true, true}`. By using a weighted random algorithm, as above, to choose a random tuple, the respective values can be taken from the tuple and set in our consonant-choosing code.

These values do not refer to whether voiced or unvoiced phonemes themselves are included in the phonology - rather they describe whether there is a distinction made between voiced and unvoiced phonemes. This can be simplified as "if the language makes a distinction between the two, include both voicings." Otherwise, the algorithm should only include only the consonant which has the phonology's "voicing" for that manner of articulation, which can be chosen randomly.

By using these refinements for generating a phonetic inventory, a unique and varied, yet still realistic, phonology can begin to be generated. This phonology provides the basis upon which the rest of the language is built.

2.2 Syllables

The next step towards a full phonology is building syllables out of the phonemes from the generated phonetic inventory.

Figure 2.7: Syllable structure

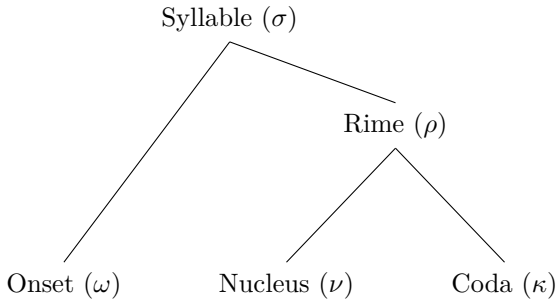

Figure 2.7 shows the structure of a syllable. A syllable is formed out of two parts - the *onset* and the *rime*. Furthermore, the rime consists of a usually required *nucleus* and a *coda*. Both the onset and coda can be consonants or consonant clusters. The nucleus consists of a vowel or vowels (monophthong or diphthong).

The role of *yod* is to construct syllables that fit this structure and then form them into words. The simplest method of generating syllables is by taking random phonemes from the pre-generated phoneme inventory and filling the onset, nucleus and coda with them. This method can be easily refined by not always filling all the parts of the syllable - the onset and coda can both be optional, so adding them in every instance is less realistic.

Another simple improvement that can be made is allowing for consonant clusters and diphthongs, which is done by picking multiple phonemes when filling a section. In order to vary the results, this should not be done every time, but should vary in frequency. The addition of clusters and diphthongs allows for more complex syllables to be created.

The maximum number of consonants in a consonant cluster, the maximum number of vowels in the nucleus, as well as whether any of the parts of the syllable are optional, are all properties of the language. Thus, syllable structures can be generated. The structure of a syllable is commonly shown as a string of either 'C's or 'V's (consonants or vowels)[8]. A language permitting only syllables with exactly one onset consonant and exactly one vowel, and no coda, would be described as having a CV syllable structure.

This representation also shows variable structures where some phonemes are optional - that is, the number of phonemes in a segment can differ. For this, bracket notation is used to denote an optional phoneme. For example, the syllable structure of the English language is cited as being (C)(C)(C)V(C)(C)(C)(C). This represents a syllable structure where the onset can have 0-3 consonants, the nucleus always contains a vowel, and the coda can have 0-4 consonants. While it is rare in English for a syllable to have the largest possible onset *and* coda clusters at the same time, examples can still be seen; the monosyllabic word *strengths* can be transcribed as /stɹɛŋkθs/. This word displays an onset cluster of length 3 (/stɹ/) and a coda cluster of length 4 (/ŋkθs/).

In order to generate a syllable structure in its most basic form, a minimum and maximum cluster value can be defined for onset, nucleus and coda. This yields a decently varied set of possible structures. To augment this, weights can be assigned to each possible value from the minimum to maximum, meaning that some lengths of phoneme clusters will be rarer than others. As in the case of the English language's 4-length coda consonant cluster, which is uncommonly seen, some clusters in the generated language will be less common. Not only does this increase variety in the languages that *yod* generates, but it also is more realistic, as it would be unusual to have all possible cluster lengths equally probable.

Now, with a syllable structure for the language decided, a syllable can be created based on it. First, a syllable structure is needed for this syllable - this can be randomly generated from the phonology's structure. For example, given English's (C)(C)(C)V(C)(C)(C)(C) structure, there are many possible subsets of structures that a syllable could have and still be considered valid. At its most complex, a syllable could match the above structure exactly, and at its simplest the syllable could just have the structure V. Most syllables will lie somewhere inbetween these two extremes. Treating the onset, nucleus and coda separately, a random value can be selected using a weighted random algorithm on our weighted set of possible values. Then a number of phonemes equal to the chosen number for each section can be chosen. By concatenating the onset phonemes, the nucleus and the nucleus phonemes in that order, a syllable is created.

Using this method, and the set of phonemes from the English phonology, the sample set of syllables in figure 2.8 was generated. The syllables in the figure are based on a generated syllable structure of (C)(C)V(C)(C).

Both syllables that match the most complex possible structure in this phonology (/d͡ʒtʊd͡ʒd/ and /pbɒhm/) and the least complex possible structure (/ɜ/ and /ɒ/ can be seen in this selection, showing the range of the syllable generation

Figure 2.8: Sample syllables with weighted structure (C)(C)V(C)(C)

/ɹlɒz ðidg əpn vɔsg ɹu ʃʒʊsθ ɪ ðjæ tæ sɒw vzʌf ɒkk ŋɔh eð æθ ɑʃʒ ɒsð fθɜð ɒʒ ʃɪ d͡ʒɒp əh həzw bəp ŋujn ɪl giʒw æʃh sjəl d͡ʒæŋð ʌd͡ʒ væ θʊlð iðd͡ʒ zɑl nɑ vʃɜst ʌfd͡ʒ lɔwm ɹɑld͡ʒ d͡ʒtʊd͡ʒd lɜvz ɜ ɒ idl ʌjn pbɒhm td͡ʒɑfð fɜʃj/

code. If each syllable in this sample is considered to be a monosyllabic word, the results from this code are already noticeably more realistic than the words generated in figures 2.1 and 2.2 (although in this sample only the phonemes from the English phonology are used, which can give a false sense of 'realism'). The structure of building clusters of consonants around a central vowel base creates realistic syllables.

However, there are still some instances where unrealistic phoneme sequences can be spotted, even in this short sample of results. For example, the initial cluster of /pb/ in the word /pbɒhm/ consists of two successive stops - both the voiced and unvoiced bilabial plosive. For a word to contain multiple plosives in a row already makes it harder to pronounce, and the fact that they are both bilabial plosives only increases that difficult. Either both are audibly released, which results in a long, difficult to pronounce cluster, or the first consonant will be deleted completely. The generated syllable /ɒkk/ also provides some trouble, as "double consonants" are rare in IPA transcriptions of natural languages. Usually when a double consonant appears in a transcription, it will either be across a syllable boundary (e.g., a sequence /...kk.../ is likely to represent the coda of one syllable and the onset of another syllable /...k.k.../) or the gemination of the consonant (e.g., /...kk.../ is another way of writing /...kː.../). However, these uses are rare, and most commonly a single consonant would be used in their place, or the IPA symbol for gemination inserted.

cite

In order to solve the problem of unrealistic consonant clusters, syllables can be built based on a 'sonority hierarchy'. The sonority hierarchy is a scale showing the relative strength of different phonemes based on their properties[9]. Different phonemes have different amplitudes, and patterns in the amplitudes can be seen depending on the place and manner of articulation, as well as the phonation. It can be seen that syllables in many natural languages follow a structure whereby the nucleus has the most sonorant phonemes, and then the rest of the phonemes in the syllable decrease in sonority as they get further away from the nucleus. This is known as the *sonority sequencing principle*. It can be seen in English by considering the initial consonant cluster /pl.../ compared to /lp.../ - the /l/ is a lateral fricative, which is more sonorous than the obstruent stop /p/. Because of this, the cluster /pl/ is much more common in onset consonant clusters. As the inverse of this, the cluster /lp/ is much more common in coda consonant clusters globally than /pl/. In English, an example of initial cluster /pl/ would be the word *plant* (/plænt/), and an example of coda cluster /lp/ would be the word *help* (/hɛlp/).

Figure 2.9: Sonority hierarchy[10]

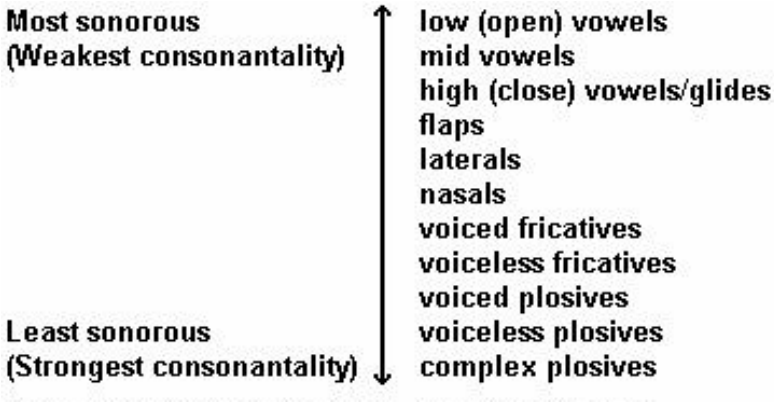

Table 2.6: Sonority hierarchy assigned values

Type	Value
Open vowels	0
Mid vowels	1
Close vowels (and semivowels)	2
Trills, flaps and taps	3
Laterals	4
Nasals	5
Voiced affricates and voiced fricatives	6
Unvoiced affricates and unvoiced fricatives	7
Voiced stops	8
Unvoiced stops	9

Applying the sonority hierarchy rules to the phoneme-picking algorithm will make a profound difference to the realism of the generated syllables. In order to implement the hierarchy in the code, every phoneme must be given a sonority value. Using the image in figure 2.9, a value from 0 to 10 can be given to every single phoneme based on its type. In fact, the IPA and therefore *yod* contains no complex plosives, so the values range is 0 to 9 inclusive.

Figure 2.6 shows how each category of phoneme is assigned a value representing its sonority. With this information encoded into each phoneme representation, the next step is to factor these into the strategy of picking phonemes for syllables.

To construct a syllable, generating the vowel first allows for the rest of the syllable to be more easily generated. Randomly choosing a vowel will give a value n between 0 and 2 inclusive. Then random consonants are chosen for the onset with the restrictions that all of their sonority values are greater than n, and for each one, the sonority value is greater than the one directly after it. Similarly, consonants are chosen for the coda with the same restrictions, except

that each consonant must have a greater value than the one *preceding* it.

Algorithm 2 Onset generation algorithm

```
1: phonemes ← empty list
2: length ← desired phoneme length
3: min ← 2 + (length − 1)                          ▷ see explanation
4: max ← 9                                  ▷ 9 is the highest value in fig. 2.6
5: i ← 0                                                    ▷ loop counter
6: while i < length do
7:     consonant ← random consonant with value between min and max
8:     append consonant to phonemes
9:     min ← min − 1
10:    max ← consonant.sonority − 1
11:    i ← i + 1
12: end while
```

Algorithm 3 Coda generation algorithm

```
1: phonemes ← empty list
2: length ← desired phoneme length
3: min ← 2
4: max ← 9 − (length − 1)
5: i ← 0                                                    ▷ loop counter
6: while i < length do
7:     consonant ← random consonant with value between min and max
8:     append consonant to phonemes
9:     min ← consonant.sonority + 1
10:    max ← max + 1
11:    i ← i + 1
12: end while
```

The pseudocode in algorithms 2 and 3 show how *yod* pieces together these segments. The similarities can be seen, with small changes due to the fact that the coda algorithm is essentially the same as the onset algorithm except it is building it in reverse order.

The reason for defining the minimum sonority value as 2 (or in the case of the onset algorithm, $2 + (length - 1)$) is that 2 is the lowest possible value that a consonant can gave, based on the data in figure 2.6. This means that the algorithm does not need to factor in the value of the nucleus vowels, as they can all be assumed to be between 0 and 2 (the range of values that a vowel can have).

The reason for the addition of $length - 1$ to our initial minimum value in the onset generation algorithm is to make sure that there are enough consonants to pick on every iteration. If min was set to 2 and $length > 1$, then the algorithm could pick a consonant of value 2 on the first iteration, and then not be able to

Figure 2.10: Sample syllables following sonority sequencing principle

sn3iʒh ʒi iɑh njʌʊʃ d͡ʒlɒθ sɪmt͡ʃ ŋjæ ɻjɜɪ mjɪt͡ʃd zɔg t͡ʃɹɪæd sɜf ʌð͡tʃ t͡ʃnɒθ kʒeð ʊt͡ʃ sʌvʃ figk ɪʃ ɻjɒd͡ʒb d͡ʒ3m kət͡ʃd uʊv t͡ʃniv d͡ʒŋʌæs ɪɹð ʒləp pʊʌ θuk pɒ lwɒwd hæt͡ʃ dət͡ʃ liɔ vjɔvf uzg uew ɑiθk zʊɑj pvuŋ tumθ feʌmk ʃʒi uf ðeɒn ʒɪɑɪj iɑz t͡ʃŋæ ulθ id

pick any consonants on subsequent iterations due to the fact that there are no consonants with *sonority* < 2. The same concept is true for the coda generation algorithm, except in this instance the sonority value must increase over more loops. Because of this, the initial maximum value must leave enough room so that larger maximum values can be chosen on subsequent iterations without causing an error in the code.

Figure 2.10 shows a set of sample syllables generated using the sonority sequencing principle. These syllables are based on a generated syllable structure of (C)(C)V(V)(C)(C). This structure displays not only the fact that onset and coda generation follow the sonority hierarchy, but also the inclusion of diphthongs due to the V(V) nucleus structure.

2.3 Stress and Long/Geminate Phonemes

In phonology, stress is when a syllable is given particular emphasis as part of a word. Stress in a syllable is denoted by an IPA symbol which is similar to an apostrophe, as seen in the second syllable of the word *hello* (/hɛˈləʊ/). Stress can be represented as a single boolean value in the `Syllable` object - if `true` then the syllable is stressed else it is unstressed.

Particularly, lexical stress (stressed placed on a given syllable of a word) is important to the word generation process. A word can have one syllable with lexical stress, and the position of this syllable differs in different languages. For some languages the stress can fall on any syllable, and must be learned per-word. For languages with fixed stress, however, the position of the stress is determined by language-wide rules, and so can be easily chosen by algorithmic rules.

Figure 2.7 shows the distribution of lexical stress positions in natural languages. Again, using a weighted random algorithm can assign the phonology a random lexical stress position from this list.

As discussed earlier, *yod*'s structure for phonemes contains a 'length' property, which has similar but distinct meanings depending on the type of the phoneme. In vowels, it refers to whether the vowel is long or not (e.g., the distinction between the words *ferry* /feˈɹi/ and *fairy* /feːˈɹi/ in Australian English). In consonants, it represents the gemination of the consonant.

In *yod*'s syllable and word generation code, these features are determined *after* a word is created. The phonology, when generated, randomly selects whether

Table 2.7: Fixed Stress Locations[11]

Value	Representation
No fixed stress	220
Initial	92
Second	16
Third	1
Antepenultimate	12
Penultimate	110
Ultimate	51
Total:	502

long vowels and geminate consonants are a part of the phonology. These features are separate, so a phonology can allow geminate consonants but not long vowels, or vice versa. Then when a word is constructed, it is scanned over to check for instances where two consecutive phonemes are identical. This can happen for two reasons: first, there is no unique restriction on vowel picking for the nucleus, so if a diphthong is generated there is a possibility for two identical vowels in sequence. For consonants, this approach does not apply, as generated onset clusters and coda clusters follow the sonority sequencing principle which forbids sequences of two consonants with the same sonority. However, since syllables are generated independently of each other, it is possible for the boundaries of two adjacent syllables to be the same consonant. That is, if a syllable ends with a consonant such as /k/ and the next syllable happened to start with /k/, the resulting word would contain the consonant sequence /...kk.../.

When a sequence of two identical phonemes is found, then those phonemes can either be reduced into one long vowel or geminate consonant, or one of the phonemes can be deleted. Figure 2.11 shows some examples of words before and after undergoing this process.

Figure 2.11: Process of converting identical phoneme sequences to 'long' phonemes

/ˈhæk.koʊ/ → /ˈhæ.kːoʊ/
/ˈpɑɑt/ → /ˈpɑːt/
/tɔɔˈpæn.naɪ → /tɔːˈpæ.nːaɪ/

2.4 Words

Chapter 3

Orthography

Chapter 4

Grammar

4.1 Lexicon

4.2 Phrase Structure Grammar

Bibliography

[1] L. L. Zamenhof. Dr. Esperanto's International Language. 1887.

[2] Svend Nielsen. Per-country rates of Esperanto speakers. 2016. URL: https://svendvnielsen.wordpress.com/2016/12/10/percountry-rates-of-esperanto-speakers/.

[3] Georges Dumézil and Tevfik Esenç. Le verbe oubykh: études descriptives et comparatives. 1975.

[4] International Phonetic Association. Handbook of the International Phonetic Association. 1999.

[5] Ian Maddieson. "Vowel Quality Inventories". In: The World Atlas of Language Structures Online. Ed. by Matthew S. Dryer and Martin Haspelmath. Leipzig: Max Planck Institute for Evolutionary Anthropology, 2013. URL: http://wals.info/chapter/2.

[6] Ian Maddieson. "Consonant Inventories". In: The World Atlas of Language Structures Online. Ed. by Matthew S. Dryer and Martin Haspelmath. Leipzig: Max Planck Institute for Evolutionary Anthropology, 2013. URL: http://wals.info/chapter/1.

[7] Ian Maddieson. "Voicing in Plosives and Fricatives". In: The World Atlas of Language Structures Online. Ed. by Matthew S. Dryer and Martin Haspelmath. Leipzig: Max Planck Institute for Evolutionary Anthropology, 2013. URL: http://wals.info/chapter/4.

[8] George Clements and Samuel Keyser. CV Phonology: A Generative Theory of the Syllable. MIT Press, 1985.

[9] Donald Burquest. Phonological analysis: a functional approach. SIL International, 2006.

[10] Eugene E. Loos et al. Glossary of linguistic terms. 2003. URL: http://www-01.sil.org/linguistics/glossaryoflinguisticterms/WhatIsTheSonorityScale.htm.

[11] Rob Goedemans and Harry van der Hulst. "Fixed Stress Locations". In: The World Atlas of Language Structures Online. Ed. by Matthew S. Dryer and Martin Haspelmath. Leipzig: Max Planck Institute for Evolutionary Anthropology, 2013. URL: http://wals.info/chapter/14.