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 $max = 8$ the following example text can be generated.

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 phonetic inventory 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 which was taken 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

lθθɪ̄ çgd͡z ed͡zɻlʊ lt̪͡ɪ̄gt̪͡ɪ̄χʊɻt̪͡ɪ̄ lʎ̝ɪg̟ɪ̄ç d͡zħɖ͡zˌ ħħɻɪ̟̄eθ ed͡zçd͡z ʎ̝̊ʊd͡zelʊ ʊlʊχħ

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 the final set of properties ends up 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

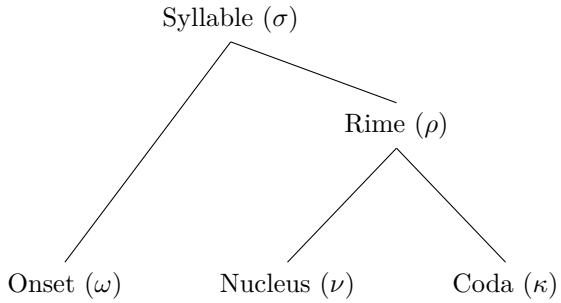


Figure 2.7 shows the structure of a syllable. A syllable is formed out of two parts - the *onset* and the *rime*. Furthermore, the rime consists of a usually required *nucleus* and a *coda*. Both the onset and coda can be consonants or consonant clusters. The nucleus consists of a vowel or vowels (monophthong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

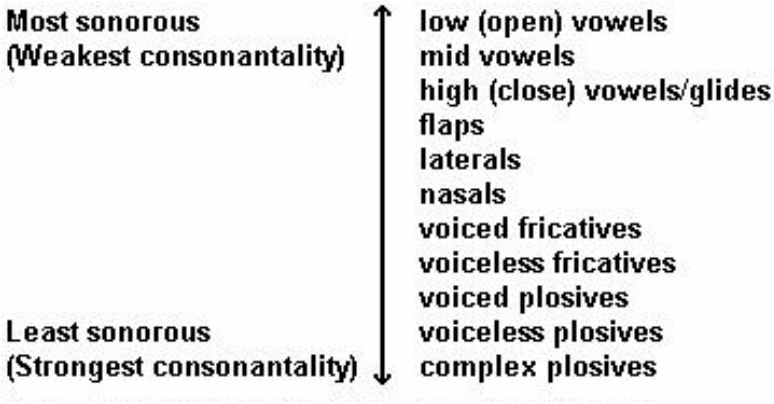


Table 2.6: Sonority hierarchy assigned values

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

snɜiʒh ʒi iɑh njʌʊʃ d͡ʒlɒθ sɪmt͡ʃ ŋjæ ɻjɜɪ mjɪt͡ʃd zɔg t͡ʃɹɪæd sɜf ʌð͡tʃ t͡ʃnɒθ kʒeð ʊt͡ʃ sʌvʃ figk ɪʃ ɻjɒd͡ʒb d͡ʒɜm 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/

A final refinement to syllable generation is to employ another general phonotactic rule. It can be seen that the phonotactics of many languages have restrictions on what consonant clusters are permitted as syllable-initial and syllable-final. For example, in Serbo-Croatian, the phonology contains both the phoneme /d/ and the phoneme /ʃ/, represented by the grapheme *š*. However, despite

Figure 2.12: Example words using subsets of English phonology

1. /ʌɜn mæɑ ɑɪj nuən ʃʌv bə lub keʌ uɜ ʌj nəj eil tab eəv jei ɹʌɑ kib lɪl d͡ʒæv buɜn/
2. /ˈhnɒɪgŋəw ʒnɒtˈɹwɒt͡ʃgðəŋ ˈnɹɒgŋɒt bt͡ʃənˈbʃəvɹɒb ˈʒɒʃɒ hvəˈtɒŋnɒd͡ʒ ˈdnəvt͡ʃɹə/
3. /eʒfˈge dɪtˈhɜ hʌˈist ŋɜ eiθbt fɪˈθɪj ɜ ziʌ ɪjˈɜt͡ʃgk tʌ ŋeˈfɪ zi/

following the sonority sequencing principle, the sequence 'šd' (/ʃd/) is not permitted as an initial cluster in a syllable[12]. English also has many phonotactic rules, some of which permit or allow certain initial and final consonant clusters[13].

While the rules governing clusters in these cases are quite complex, *yod* approximates these systems by merely restricting coda and onset consonants to randomly selected subsets of the phonology's consonant set. On generation, both onset consonants and coda consonants independently have a chance to be restricted. The restriction occurs by selecting a subset of the consonants of a random length, between a minimum of 2 and a maximum of the length of the total set.

Even though this concept has a very naïve implementation, it introduces a large amount of variety and lends further realism to the generated languages.

2.4 Words

Most of the complexity involved in generating words is involved in the actual construction of syllables. As such, word generation in *yod* is a relatively simple process. The range of possible word lengths in the language is decided when the phonology is generated - this range is in syllable length rather than phoneme length. A random value between the minimum and maximum of this range is then chosen.

Given this random value as the length of the word, that many syllables are then generated in order. Following this, both the stress-placing process and the long vowel/gemination are applied to the word, as described in the previous section.

Figure 2.12 shows three different sets of generated words, all using a subset of English phonemes. Figure 2.13 also shows three sets of word, this time generated using different subsets of the full set of phonemes. Using the English phonology makes for words which are more easily pronounceable without much practice by English speakers - however, using the full set of phonemes produces much more varied and unique results.

Figure 2.13: Example words using all phonemes

1. /ʐo˧ˈpɒ oʒɟˈʈu b̪əʑb̪˧ˈzoɢʔ ʔɯɟkˈʐəb̪ zəzb̪˧ˈdoʒɢ gəˈʐəʒk dʒuɢkˈʔɒ əgkˈʒuzk/
2. /hæ sa gu ɕɒ ɕɯ θ̠u ɕæ sɒ d̪̈a θ̠ɒ ɢɯ hɯ ʃu fɯ çɒ θ̠a ɕæ çɒ θ̠u sa hɯ gu sa θ̠a/
3. /ˈkɑɜtɑɔfgp̪ ˈθɑʈɔɜ ˈp̪ɒxɑb̪˧ˈ ˈθɜɔɒ ˈp̪ɛɔb̪˧ʈp̪ɔɜ ˈθɒɑθ̠˧tɔθ̠gk ˈp̪ɛfb̪˧ʈgɛɔb̪/

Chapter 3

Orthography

So far, the focus has been on creating a *spoken* language, due to the fact that all written languages are based on spoken languages to begin with. For constructed languages, however, not having a way of writing the language is a large negative. People would not be able to communicate non-verbally in the language, and for fictional languages, it is likely that the creator would want to have a written form of the language to use in their work.

Therefore the generation of an orthography is desired. An orthography concerns the way a language is represented by different symbols, or graphemes, in writing. There are three main types of orthography: *syllabic*, with a grapheme representing each syllable, *logographic*, with a grapheme representing each morpheme/word, and *alphabetic*, where each grapheme roughly represents a phoneme. *yod* is restricted to creating alphabetic orthographies - however the nature of the organisation of the library means that it would be a relatively easy task to take the output of the phonology generation and use that to create a logographic or syllabic orthography.

An alphabetic orthography can be seen as a map of each phoneme in the phonetic inventory of a generated language to a randomly chose grapheme, or rarely multiple graphemes. The orthography of Croatian, seen in figure 3.1, is a good example of this mapping. It was created in 1835 by Croatian linguist Ljudevit Gaj to give Croatia a unified way of writing the language. The right column shows the phonemes included in the language and the left column shows to which grapheme (or in some cases, digraph) the phoneme maps.

In English, such a table would be much more complex, as in English a phoneme can have many different written representations, and many graphemes or grapheme clusters represent multiple phonemes depending on context. Languages where there is a more complex mapping such as English are said to have a *deep orthography*, in contrast to the *shallow orthography* of Croatian, as well as the languages generated by *yod*.

To generate an orthography for a generated language, then, a system must exist to assign each phoneme a grapheme or grapheme cluster. *yod* handles this by having an object which maps each possible phoneme in the International

Table 3.1: Orthography of the Croatian language

Phoneme	Grapheme
A a	/a/
B b	/b/
V v	/v/
G g	/ɡ/
D d	/d/
Đ đ	/d͡ʑ/
E e	/e/
Ž ž	/ʒ/
Z z	/z/
I i	/i/
J j	/j/
K k	/k/
Lj lj	/ʎ/
M m	/m/
N n	/n/

Phoneme	Grapheme
Nj nj	/ɲ/
O o	/ɔ/
P p	/p/
R r	/ɾ/
S s	/s/
T t	/t/
Ć ć	/t͡ɕ/
U u	/u/
F f	/f/
H h	/x/
C c	/ts/
Č č	/t͡ʃ/
Dž dž	/d͡ʒ/
Š š	/ʃ/

Phonetic Alphabet to a list of possible symbols. This list was compiled as a default set of options - however, a user interacting with the library directly would be free to change this dictionary or add to it, in order to add more options. Of course, the user is also free to set the orthography directly to whatever level of specificity they desire.

Many of the possible symbols in these lists are based off common grapheme representations for their respective phonemes in different languages. To increase the variety, these symbols are also included with different diacritics for many of the phonemes. For each list, the first value is the 'preferred' symbol, and will be chosen on average 85% of the time, using default values. In the case of the preferred symbol not being picked, a random symbol from the rest of the list is picked. When generating a random orthography, the percentage chance of using the default symbol is able to be specified, which allows users to

Since many of the lists contain the same symbols, as the lists are primarily differentiated by having different preferred symbols, there is a small chance that the same symbol can be picked for different phonemes. However, as seen with the phonology of English, the same graphemes can sometimes represent many different phonemes, so if this occurs, it merely acts as another unique factor of the language. As long as phonemes get unique graphemes in a majority of instances, the language will not be overly ambiguous.

This process of picking which grapheme each phoneme is mapped to happens on the instantiation of a new `Orthography` object. This means that once defined, the orthography will give the same grapheme result for each phoneme, no matter how many times it is checked. This provides a consistent orthography for the language.

Furthermore, since this process converts one phoneme to a string represent-

Figure 3.1: Orthographised words, using orthography from fig. 3.2

âdp bĕrdh ĕrdh tâð'q̃ hâdhq uzd ârd âht turṭ ĕz ō ru ōgq̃ tudhq̃ ōɽdh duðq rādq â râð't hurd duɽht uhd ĕ tōgṭ durp dâ u bōð rōdp buɽhd hu ōbt bōɽd

ing its related grapheme or grapheme cluster, and syllables can be treated as ordered lists of phonemes, an orthography can be applied to syllables by mapping the conversion function onto the phoneme list of a syllable. Since `yod` rarely uses individual graphemes past this point, it is more useful for the function to return a string representation of the orthographised syllable. Thus the list is folded with a concatenation function, concatenating the graphemes for each syllable.

This concept also applies to words, which can be seen as lists of syllables. Because there now exists a way to apply an orthography to individual syllables, mapping this function over all the syllables in a word and folding the result with a concatenation function produces an orthographised, written string representation of the word.

The `LanguageOrthography` class exposes an `Orthographise()` method which takes an input and returns a string representation of it with the given orthography. This method is overloaded to accept either a `Phoneme`, a `Syllable` or a `Word` as input, returning the respective converted string.

The relatively simple act of creating an orthography lends a lot of 'flavour' to a language, as two languages with identical phonologies and phonetic inventories can be distinguished through the use of radically different orthographies. Figure 3.1 shows a sample of words that have been generated and then had the orthography from figure 3.2 applied to them.

Table 3.2: Generated orthography example

Phoneme	Grapheme
/p/	p
/b/	p
/t̰/	ṭ
/d̰/	dh
/t/	t
/d/	d
/q/	q
/ɣ/	g
/ʔɬ/	q̃
/ʔ/	-
/z/	z
/β/	ṿ
/ð̰/	dh’
/ð/	t̪h
/ð̲/	ð
/ɹ̲/	r
/ʁ/	ɽ
/ʕɬ/	r
/h/	h
/y/	u
/œ/	ĕ
/ɒ/	ō
/a/	â
/ʌ/	u

Chapter 4

Grammar

Given the tools described so far, a large variety of realistic words and syllables can be generated and spoken - and, if desired, converted to a written form. However, such speakers would not be communicating any useful information between each other. This is because the words that have been generated do not have any intrinsic meaning, and this meaning must be assigned to them.

The goal with *yod* was to be able to translate sentences from English to a generated language. Initially this was planned to be a straight translation from plain English. However, multiple obstacles complicated the pursuit of this goal. It became clear that doing this translation would require parsing the English sentence in full, in order to translate each word from English to the new generated language. This parsing process is outside of the scope of the project, and so a different approach was required.

Additionally, many languages, including English, encode information into words in the form of *inflection*, where the word is modified somehow to express information about different grammatical categories. These categories include tense, case, voice, aspect (which refers to how a verb continues over time), person, number, gender and mood. In order to accurately translate a sentence, this information would need to be included in the translation process. Also, another goal was to encode some of this information in the final translated sentence that *wasn't* in the original sentence. An example of this would be the addition of applicative grammatical cases - for example, in a language with 'instrumental' case, the word for 'crowbar' in the sentence 'I opened the hatch with a crowbar' would be *declined* to match the fact that the word is being used to represent a tool being used. This change could be represented by a suffix, morphologically changing the word to be 'crowbar-[instrumental case suffix]'. There are many more grammatical cases than are contained in just the English language, and being able to add these to a generated language would create much more unique results that would differ from just straight translations of English.

Another possible downside of translating from English is the problem of word order in sentences. Many languages use different orderings of subject,

object and verb in their sentences[14]. Merely copying the word order from the supplied English sentence would result in sentence structures which were too familiar. Instead, a range of different structures was desired.

These were the goals and obstacles related to being able to construct meaningful sentences with *yod*. Much of these restrictions heavily shaped the structure of the program, and required some elements to be cut back in complexity to be able to make the program easily usable.

4.1 Lexicon

The first step towards generating and translating full sentences is to build a lexicon. A lexicon can be seen as a list of *lexemes*, which in turn are the basic meaning-carrying elements of a language. In essence, a lexicon can be seen as a dictionary for the language which was generated, and each lexeme is a headword in that dictionary. The lexemes can also be referred to as words, which is closer to what one may imagine when thinking of the concept of a 'word', but it is important to keep in mind the distinction between these units of meaning and the meaningless 'words' generated by the phonology code previously.

In fact, in this case it is more apt to consider a lexicon as a bilingual dictionary, which people use to translate from one language to another. Each entry in such a dictionary has several properties which very easily map to properties in the code definition of lexemes.

The first property of a lexeme is its 'headword', or lemma. This is the base form of the word, without any inflections or other morphological processes applied to it. For example, the words 'sing', 'sung', 'sang', 'singing' and 'sings' are all forms of the lemma 'sing'. In dictionaries, this is often necessarily an orthographised form of the word - however, in code it is simpler to keep the phonetic information, and only orthographise when really necessary. Thus, the lemma of each of the lexemes contained in the language's lexicon will be 'phonology words', which were described in section 2.4 as ordered lists of syllables, each of which was an ordered list of phonemes.

The next property is the 'part of speech'. This shows which grammatical function a word plays in a sentence. This is essential information with regards to constructing meaningful sentences. It is also necessary to have this information when applying morphological processes to words, as the rules often differ depending on the part of speech. For example, in English, a past tense verb will usually have an '-ed' suffix, but there is no equivalent rule for nouns. Figure 4.1 shows a list of every part of speech which is a part of *yod*'s word tagging system, and its corresponding tag when serialised.

Finally, each of the lemmas has a 'translation'. In *yod*, this is idiomatically in the form of a single English word, but it can actually be any string which does not contain a space. This information is used to translate from English words into words of the generated language.

By necessity, when translating a sentence, all words in the sentence must have a corresponding entry in the lexicon, or they can not be translated. The

Table 4.1: List of parts of speech recognised by *yod* and their tags

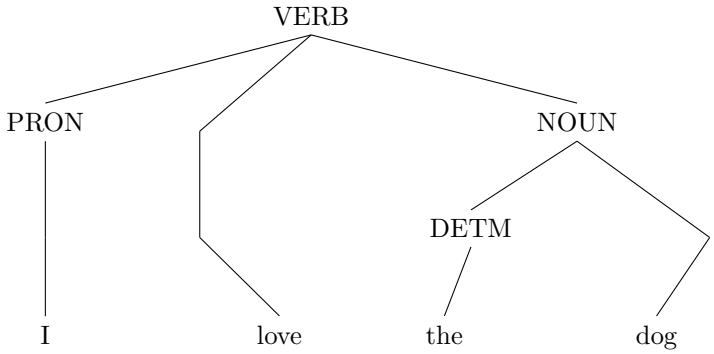
Part of speech	Tag
Pronoun	`PRON`
Noun	`NOUN`
Verb	`VERB`
Adverb	`ADVB`
Conjunction	`CONJ`
Preposition	`PREP`
Adjective	`ADJC`
Interjection	`INTJ`
Determiner	`DETM`
Relativiser	`RLTV`

lexicon containing all needed must be defined and passed to the library; for more general use, a larger lexicon containing many words could be used to prevent the user from having to define their own lexicon. This lexicon is given in the form of a JSON object, which can be deserialised from a file. This file contains a list for every part of speech, and in each of those lists is every needed word which is of that part of speech. When loaded, this gives us every word and its corresponding part of speech, and so a lemma can be generated for that word by the phonology.

While testing the assignment of lemmas to words using this method, it became clear that this approach was insufficient for certain, commonly used words. It can be seen that common words in language frequently are shorter on average than less common words[15][16]. When this trend is not taken into account, it can lead to simple words like 'be', 'I' and 'you' having an unwieldy amount of syllables. This makes the generated text significantly less realistic, especially if spoken - however, with no predetermined corpus to draw frequency data from, a different approach is needed. The method *yod* uses is to allow the user to designate certain words as 'common words.' When the lemma for these words is generated, their lengths are restricted to the minimum word length of the language. This ensures that common words are on the whole easier to speak, while still making sure that their lemmas fit within the phonological rules of the language.

Another improvement to lexicon generation is to try to group similar words. Since language usually evolves over a long period of time, certain separate related words can be introduced to the language with similar roots. In the absence of a naturally developed language, these similarities must be introduced artificially. *yod* uses a rather naïve approach to this, which nonetheless has a reasonable impact on the generated text. Similar to the way that common words are handled, the library allows the user to declare sets of related words in the lexicon file. The first time the lemma for a word in one of these sets is generated, it is generated normally. However, a list of all the groups is kept during the generation process, and this first lemma is set as the 'base lemma'

Figure 4.1: Parse tree in dependency-based grammar



for that group of words. On subsequent words of that group, the base lemma is recalled and then slightly modified. This means that words in groups of related words will have related lemmas. While this is not much of a realistic procedure, it nonetheless results in a small amount of extra realism to the language, as it evokes the idea of the language evolving more naturally.

4.2 Phrase Structure/Constituency Grammar

The phrase translation process is perhaps the most important section of *yod*'s whole translation section. The goal for this code is to take some representation of an English sentence and convert it into a sentence of a generated language, preserving its meaning. If the input is merely an English sentence (as a string) then in order to get information about the sentence structure and also the grammatical categories of each of the words, some amount of natural language processing (NLP) is required. As NLP is out of the scope of the project, an object structure which encodes all of this information is necessary.

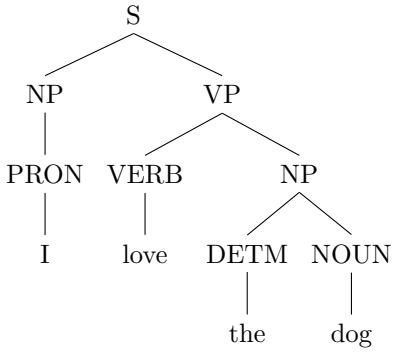
There are two different types of parse tree used in natural language processing for representing sentence structures: dependency-based parse trees and constituency-based parse trees. Both tree structures represent the same data, but in different ways.

Constituency-based grammars contain non-terminal and terminal nodes, with each terminal node representing a lexical token in the sentence, and each non-terminal node representing a 'phrase'. Dependency-based grammars treat every node as terminal, which creates simpler trees overall.

fix graphs

Figure 4.1 shows a sentence ('I love the dog') parsed into a dependency tree. Each node in the tree directly maps to a word in the sentence. Figure 4.2 shows the same sentence in constituency tree format. Dependency grammars can be more useful in some circumstances because of their simpler nature - however, constituency grammars (or 'phrase structure grammars') contain a lot more

Figure 4.2: Parse tree in constituency-based grammar



information about the phrase structure of the sentence, and so are better for the translation process. As such, *yod*'s sentence input is given in constituency tree format.

The tree input is represented as a JSON file, with properties of the object representing roots of subtrees of the current node. An input must be a single constituent of the tree, meaning the root must be a single node; the translation process can act on any node of the tree. The terminal nodes are represented by an object containing a `word` property which represents the English lemma of the word to be translated. This value is necessarily not inflected due to the fact that the lemma is needed to get a translation from the lexicon, and getting the lemma from an inflected English word is not necessarily a trivial task. Thus, preparing a sentence into the structure required by *yod* requires a bit of manual editing, or a tool designed to use natural language processing to convert English sentences into this format. The JSON format was chosen over other forms of serialisation largely to allow for easy hand editing as well as compatibility with other software.

The object which represents a terminal node also has a property which is a list of 'tags' which represent different grammatical categories. This allows grammatical information to be carried through to the translation process, and allows for the possibility of morphological changes to be applied to generated words based on their grammatical categories.

The part of speech of the terminal node is represented by the key of its parent. When loading in a phrase from a JSON object, this information as well as all the information contained in the node is used to construct a logical representation of the sentence.

This logical representation depends on the exact structure of the grammar required. *yod* allows users to specify their own phrase structure grammar, as a full grammar of English would be too complicated for a user to be able to parse their sentences into, hindering the ease of use of the library. Allowing for custom grammars also allows users to devise more complex and unique languages - for example, if a writer wanted to create an alien language which lacked verbs,

an English constituent grammar would not be sufficient. More complicated and unusual sentence structures can be permitted through a data-driven approach to creating a grammar. The grammar object should be thought of as representing the generated language's grammar, rather than an English grammar.

Grammars are also serialised and deserialised as JSON objects or files containing JSON objects. These objects have a very simple structure: the object contains a key called `rules` which then contains a key for each non-terminal constituent in the grammar. For each of these constituents, there is a list of phrases that it can be expanded to. Each of these phrases consists of a list of constituents, terminal or otherwise, which is ordered such that when a sentence is translated, the elements of the phrase will be appended to the final sentence in that order.

For phrases which contain multiple of the same constituent - for example, a compound noun phrase could be represented as `NOUN NOUN`, it is necessary to indicate the order in which these constituents are represented in order to distinguish the two nouns from each other when later translating the phrase. To accomplish this, duplicate constituents in phrases are marked with numbers; a compound noun in this form is represented as `NOUN-1 NOUN-2`.

The last addition to the phrase structure grammar file is not necessarily a part of the grammar itself but instead an instruction on how to construct certain phrases in the language. This property is a list of rules which should be 'compounded'. When the final translated sentence is constructed, a space is added between each constituent. If a rule is set to be compounded, however, the constituents of the rule will be concatenated without a space between them. A good example of this type of rule is the English language, where noun-noun compounds are frequently created by taking two nouns and concatenating them without a space: $fire + place \rightarrow fireplace$, $wrist + watch \rightarrow wristwatch$.

An example phrase structure grammar which can be used to translate moderately complex English sentences is shown in figure 4.3, and a conversion of the sentence 'I love my big dog' to *yod*'s dependency tree format is shown in figure 4.4.

When parsed, this example phrase will build into a tree of `Phrase` objects. Each of these objects holds information about its current phrase grammar rule, as well as a boolean value which says whether the phrase node is terminal or non-terminal. If a phrase is designated as non-terminal, it also contains a list of its sub-phrases - otherwise, it contains a reference to a `Word` object containing information about the word it represents.

Once this tree structure is constructed, it can begin to be translated. The first step of this is to translate the lemma of each terminal node in the tree using the generated lexicon. The corresponding lexeme with matching lemma and part of speech for each word is looked up in the lexicon, and the translated word from the lexicon is then added to the node. The reason for indexing by lemma *and* part of speech is that some words could have the same lemma but different parts of speech. An example of this is the distinction between the word 'that' as a determiner (e.g., 'I would like *that* one') and the word 'that' as a relativizer, which is a conjunction which introduces a relative clause (e.g., 'I

Figure 4.3: Example phrase structure grammar

```
{
  "rules": {
    "S": ["NP VP", "S-1 CONJ S-2", "INTJ S"],
    "NP": ["PRON", "NN", "DETM NN", "NP PP", "NP RC
        ", "PS NN"],
    "VP": ["VERB", "VERB NP", "VERB ADJC", "VERB PP
        ", "VERB ADVB", "VERB PRON"],
    "PP": ["PREP NP"],
      "RC": ["RLTV VP"],
    "NN": ["NOUN", "ADJC NN", "CN"],
    "CN": ["NOUN CN", "NOUN-1 NOUN-2"],
    "PS": ["NOUN", "PRON"]
  },
  "compound": {
    "NN": ["ADJC NN"],
    "CN": ["NOUN-1 NOUN-2"]
  }
}
```

live in the house *that* is on the corner'). These words are homonyms - that is, they are spelled and pronounced the same, but have different meanings - and uniquely identifying each word in the lexicon by its lemma *and* part of speech avoids the ambiguity which comes with them.

When each terminal node contains a translated word, the tree is then flattened to produce a list of terminal nodes. A recursive tree-flattening algorithm is run on the tree, which checks if a node is terminal or non-terminal. Terminal nodes merely return their word values (including English lemma, tags and parts of speech). Non-terminal node first flatten all of their children, and then concatenate these lists in the order specified by the node's grammar rule. For example, consider a `VP` (verb phrase) phrase which satisfies the rule `VP` $\rightarrow$ `VERB NP`. It is non-terminal, and contains a child of the type `VERB` and a child of the type `NP`. Both of these children are flattened, with `VERB` merely returning a list with one element. The rule then specifies that the lists should be concatenated in the order `VERB` followed by `NP`.

Additionally, during this process the rule for each phrase is checked to see if it needs to be compounded. If so, a list fold will be applied to the final list which merges all of the words, producing a single-item list which contains a word representing a compounded version of the phrase's children. This folding process ensures that the compounding process is more robust than simply not adding spaces between the words when printing the sentence - the compounded nature

Figure 4.4: Example phrase for grammar shown in fig. 4.3

```
{
  "S": {
    "NP": {
      "PRON": { "word": "i", "tags": "1,SG,SBJ" }
    },
    "VP": {
      "VERB": { "word": "love", "tags": "1,SG,PRS"
         },
      "NP": {
        "PS": {
          "PRON": { "word": "i", "tags": "1,SG,OBJ,
             GEN" }
        },
        "NN": {
          "ADJC": { "word": "big", "tags": "SBJ" },
          "NN": {
            "NOUN": { "word": "dog", "tags": "3,SG,
               OBJ" }
          }
        }
      }
    }
  }
}
```

of the phrase is reflected in the sentence data structure itself. Thus, a compound of the words 'fire' and 'truck' would result in a word element representing the whole noun-noun compound word 'firetruck', with its own compounded translation. The part of speech and tags for the resulting compound are carried over from the final word in the compound, allowing the noun 'truck' in this example to keep its part of speech as well as any possible applicable tags.

The tree flatten function returns a list, which represents a sentence in the generated language. Each element in this list holds the following information:

1. English lemma
2. Translated lemma
3. Part of speech
4. List of grammatical categories (tags)

In order to access the IPA representation of the full sentence, the list is folded which concatenated the translated lemmas of each element. If a written form is desired, then a `LanguageOrthography` must be specified - in this case, the same fold is applied but the `LanguageOrthography.Orthographise()` method is mapped onto the list first. Due to the overloaded nature of this method, as developed in chapter 3, the method can be applied to phonological `Word` objects directly, returning their orthographised string representations.

When flattened, it is feasible to imagine that the library could just return string representations of the phrase directly - for example, the full IPA transcription of the sentence. The reason for returning a list of `Word` structures instead is because this extra information is useful for showing that the sentence is a valid translation of the input. Linguists use a tool called an interlinear gloss to show the relationship between a source text and its translation[17]. For `yod`'s output, the generated sentence is treated as the 'source text' and the corresponding English is treated as the translation. Figure 4.5 shows an interlinear gloss of the sentence 'I love my big dog' in a language generated by `yod`, along with the relevant entries in the language's lexicon.

This figure shows the result of the translation process being run on the phrase described in figure 4.4, using the grammar in figure 4.3, as well as a generated orthography and a phonology based on that of English. It shows the translation of words using a lexicon, as well as the compounding feature shown using the compound adjective-noun 'big-dog'.

As described before, most classes in `yod` expose a `Generate()` method which creates an instance of that class with procedurally generated values in order to allow for varied and unique language generation. The grammar and word order of a language plays a large part in creating an identity for the language distinct from other languages. Because of this, it is desirable to create a unique grammar.

However, generating a random grammar poses an interesting problem. This problem is that the input sentences to `yod` necessarily follow the grammar of the output language. Because of this, having different constituents in the grammar on separate runs of the creation process will mean that an input sentence could

Figure 4.5: Sentence shown in fig. 4.4 using grammar from fig. 4.3

(1) *đâh đum đâh mebgwu*
/d͡ʒɑh d͡ʒum d͡ʒɑh mɜbgwʌ/
I love I big-dog
'I love my big dog'

Relevant lexicon entries:

- `đāh` /d͡ʒɑh/ `(pron) : I`
- `gwu` /gwʌ/ `(noun) : dog`
- `đum` /d͡ʒum/ `(verb) : love`
- `meb` /mɜb/ `(pron) : big`

potentially not match a generated grammar. This is unwanted, since if a sentence doesn't match the grammar, it cannot be translated; if the user doesn't know if the sentence cannot be translated before running it through the library, this means the library cannot be relied upon for uses where a translated sentence in required without the chance of the library not returning a value (for example, in game development usages). Furthermore, the more varied the results of the grammar generation, the lower the chance of the sentence structure matching the grammar. For example, if the generated grammar does not allow verbs, most phrases would be untranslatable.

To work around this, `yod` can generate unique grammars, but these grammars are all based on the default grammar shown in 4.3. This grammar allows for complex English sentences to be represented and translated. The randomisation aspect is implemented by randomising the order of many of the phrase rules. This means that, while the default structure defines the phrase rule `VP` → `VERB NP`, the generation algorithm can instead define the rule `VP` → `NP VERB`. This verb-final rule can be seen as similar to languages such as Japanese, where the object precedes the verb.

Additionally, the compounding rules can be generated. In the example grammar, both rules `NN` → `ADJC NN` (for adjective-noun compounds) and `CN` → `NOUN-1 NOUN-2` are compounded. When generating the grammar, either of these rules can be set to compound or non-compound rules at random. This lends a small amount of extra variety to the produced grammars.

One limitation of this generation strategy, and of constituency-based gram-

mars in general, is that it does not lend itself well to following some statistical data about natural languages. Linguists have gathered much data about the order of different parts of sentences in languages from all cultures[14]. Some of these patterns can be represented in the generation code - for example, statistics about the order of demonstrative and noun[18] and the order of adjective and noun[19] can be represented by changing the order of the `NP` $\rightarrow$ `DETM NN` and `NN` $\rightarrow$ `ADJC NN` phrase rules respectively. These statistics are used to inform the probability of the different orders these rules can take. However, for some rules about sentence structure, such as the ordering of subject, object, verb and oblique[20], the use of constituency-based grammars adds a lot more complexity. Specifically, creating a grammar where the object of a sentence and the verb were not adjacent would mean that the previously correct input structure would now not be able to fit into the grammar. This would happen because the grammar defines a `S` $\rightarrow$ `NP VP`, where S is a sentence. In this sentence structure, the noun phrase is necessarily the subject of the sentence, and the verb phrase contains both verb and potentially also another noun phrase which represents the object of the sentence. As the verb *and* the object are children of the verb phrase in a tree constructed based on this grammar, they must necessarily be adjacent when the tree is flattened.

An improvement to the grammar generation section of `yod` would therefore be to construct a more complicated English-based grammar which could potentially have the property of a verb-subject-object or object-subject verb structure, while still allowing input sentences to be structured in a relatively human-friendly format.

By randomising the order of phrase rules, as well as compounding rules, languages with different sentence structure can be generated. This helps the translated sentences to differ from the structure of the input sentences, instead of merely being word-by-word translations. On top of this, the `LanguageGrammar.LoadFromJSON()` method is available for users who want to use more unique grammars and construct sentences which conform to those grammars. This means that a user who needs the complexity of custom grammars can have it, but a user who does not can still have a unique grammar which allows for complex sentence translation.

4.3 Inflection

Given the translated sentence in figure 4.4, it can be seen that the pronouns 'I' and 'my' are both represented by the same translated word *đ*, despite being different in the English translation. The interlinear gloss indeed shows that both of these are reflecting the lemma form of the pronoun 'I'. In order to fully express the same meaning as the sentence 'I love my big dog', it needs to be somehow indicated that the second instance of the pronoun should be marked with genitive or possessive case.

`yod` contains a basic inflection engine to accomplish this task. Inflection is the morphological modification of a word to reflect its grammatical categories[21],

often through the use of an affix. An example in English would be the addition of the suffix ‘ś’ to a noun to indicate the possessive/genitive case.

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.

[12] Steven Uzelac. “Phonological constraints in Serbo-Croatian syllable margins: and Markedness in generative phonology.–”. PhD thesis. Simon Fraser University. Theses (Dept. of Modern Languages), 1971.

[13] Noam Chomsky and Morris Halle. “The sound pattern of English.” In: (1968).

[14] Matthew S. Dryer. “Order of Subject, Object and Verb”. 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/81.

[15] G.A. Miller, E.B. Newman, and E.A. Friedman. “Length-frequency statistics for written English”. In: Information and Control 1.4 (1958), pp. 370–389.

[16] Udo Strauss, Peter Grzybek, and Gabriel Altmann. “Word length and word frequency”. In: Contributions to the science of text and language. Springer, 2007, pp. 277–294.

[17] Balthasar Bickel, Bernard Comrie, and Martin Haspelmath. “The Leipzig Glossing Rules. Conventions for interlinear morpheme by morpheme glosses”. In: Revised version of February (2008).

[18] Matthew S. Dryer. “Order of Demonstrative and Noun”. 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/88.

[19] Matthew S. Dryer. “Order of Adjective and Noun”. 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/87.

[20] Matthew S. Dryer and Orin D. Gensler. “Order of Object, Oblique, and Verb”. 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/84.

[21] David Crystal. Dictionary of linguistics and phonetics. Vol. 30. John Wiley & Sons, 2011.