TJHSST Computer Systems Research Lab 2018-2019
Senior Research

A Lexically and Syntactically Constrained Random Latin Sentence Generator

Thomas Porter

Abstract
Beginning students of Latin practice their language skills by translating Latin sentences into English. As an alternative to the labor intensive process of manually creating sentences, while still accounting for the students' levels of skill, a constrained Latin sentence generator is developed. A dependency model is used, in which a sentence is modelled as a totally ordered rooted tree with words as vertices and dependency relationships as edges. The set of allowed words is the constrained vocabulary, and the set of allowed dependency relationships is the constrained grammar. The inflection of words is constrained by the grammar and the user, and is otherwise random. A fully customizable, grammatical Latin sentence generator is achieved, with the ability to expand vocabulary and grammar for classroom use.

Keywords
Latin — Dependency — Natural Langauge Generation

Contents

1. Introduction

Latin pedagogy requires many Latin sentences for students to practice translating. Currently, Latin teachers maintain large collections of sentence sets on paper, which is not as efficient or organized as an electronic method. These sentences must match the lingual proficiency of the students, which evolves rapidly in the first three years of study.

This project was to develop an electronic tool that can generate Latin sentences incorporating only a selected subset of lexemes (vocabulary items), syntactic dependency relations, and verb inflections. This tool is suitable for beginning and intermediate Latin students to generate Latin sentences based on the student's knowledge. The program randomly chooses which lexemes and dependencies to incorporate into each sentence, giving variety and unpredictability to the sentences.

Stochastic sentence generation is a central problem in Computational Linguistics, and is by no means a solved problem. Most modern approaches to this problem have two major features that conflict with the aims of this project. Firstly, prominent methods of sentence generation are sequential [1], meaning that the subtask of sentence generation is to choose the next word in the chronological sequence. This is unsuitable for Latin because Latin is largely non-canonical, meaning that the word order is relatively unconstrained [2]. This indicates that the relationships between words are better understood as dependencies in a tree than as successors in a sequence.

Secondly, modern approaches often use machine learning algorithms trained on corpora of language data. The aim of these models is for the frequency of linguistic occurrences in the output of the generator to approximately equal the frequency of similar occurrences in the corpora. This is unsuitable for a very tightly and variably constrained program like this, because such a machine learning algorithm is not easily amenable to such precise guidance administered after the training.

The solution employed here is to use a dependency model of syntax, generating each sentence as a tree with a ROOT element. Each node is an element of the selected vocabulary, and each edge is a dependency relation. Nodes and edges are chosen randomly among the appropriate options until each leaf of the tree declines to choose any more edges. The tree is compressed into a sequence based on a manually specified, sensible word order.

This program successfully generates a wide range of grammatical, Latin sentences, obeying the user's selections unless it is impossible or too difficult to do so. The range of sentences possibly generated by the program is significantly less than the range of all grammatical possibility, but the generated range is large enough to be useful for Latin students. When the user selects very few options, the program sometimes fails to create a sentence even when it is grammatically possible. However, this is due to the deeper inaccuracies due to the nature of a dependency model, and is likely not common enough to hinder usual use.

2. Methods

The program was implemented as a Python file. A graphical user interface (GUI) wriyten using the TKinter library handles all user input and all program output, including:

- Ending selection
- Vocabulary selection
- Grammar selection
- Approximate length selection
- Sentence number selection
- Results frame

The rest of the program consists of a sentence generator Python file, a vocabulary text file, a grammar text file, an inflection XML file, and a message text file. The message file contains the written instructions that appear at the top of the GUI.

The sentence generator builds a dependency tree from the top down using the selected words and grammatical relations. The tree building process has two main data structures: a sentence tree and a queue of newly added nodes. Each are populated by Word objects. The tree and the queue begin populated only by a ROOT node, and the following process is repeated:

1. Node is popped from the queue.
2. Dependencies on Node are found in the grammar selection.
3. A lexeme for each dependency is found in the vocabulary selection.
4. Each lexeme is inflected to become a Word.
5. Each such child is added to the tree and the queue.

This process continues until no dependencies are added. The tree is compressed into a sequence of Words, then is processed into a string, then returned. The vocabulary and grammar files are formatted to encode the information used by the sentence generator.

2.1 The Interface

A screenshot of the GUI can be found in Figure 1. It consists of seven frames:

1. Title
2. Instructions
3. Results (Lower left)
4. Initiator (Below title)
5. Vocabulary selection
6. Ending selection
7. Grammar selection

Results The results frame displays up to 8 sentences at a time. More sentences can be generated at a time, and these can be accessed using a discrete, manually programmed scrolling feature.

Initator The number of sentences to generate and the length limitation are able to accept any form of number, but will always interpret the input as between 0 and a certain maximum value. The maximum length limitation is 10, and the maximum number of results is 30. Demo mode can be toggled from the GUI, but Demo mode requires the user to look at and type into the terminal.

Vocabulary and Grammar The selector boxes for vocabulary and grammar allow the user to choose a subset of the available material with which the program will construct sentences. Selected items are gray, and all items are selected by default. The grammar selector employs hidden, or packaged, item. Some grammatical dependencies are distinct from the point of view of the program, and require distinct encodings, but should be combined into one choice for the user.

Endings The user may customize only verb endings. If no appropriate endings are selected, the sentence generator will choose an appropriate ending at random. Endings that do not exist are automatically deselected upon running the program and upon each round of sentence generation. All other endings are selected by default.

2.2 Sentence Generation

The sentence generation task employs a dependency grammar model, which models sentences as a rooted tree, with words as vertices (nodes) and objects called dependencies as edges. In this mode the rules of dependencies, dictating which ordered pair of parent and child may be linked in a dependency, characterize the languages syntax. Additionally, dependencies must contain information as to which side of the parent, and in what order, the children will go in the sequential sentence. The dependency model of syntax is the simplest and earliest of developed syntax models, and is most suitable to Latin due to Latin's loose word order and high morphological synthesis.

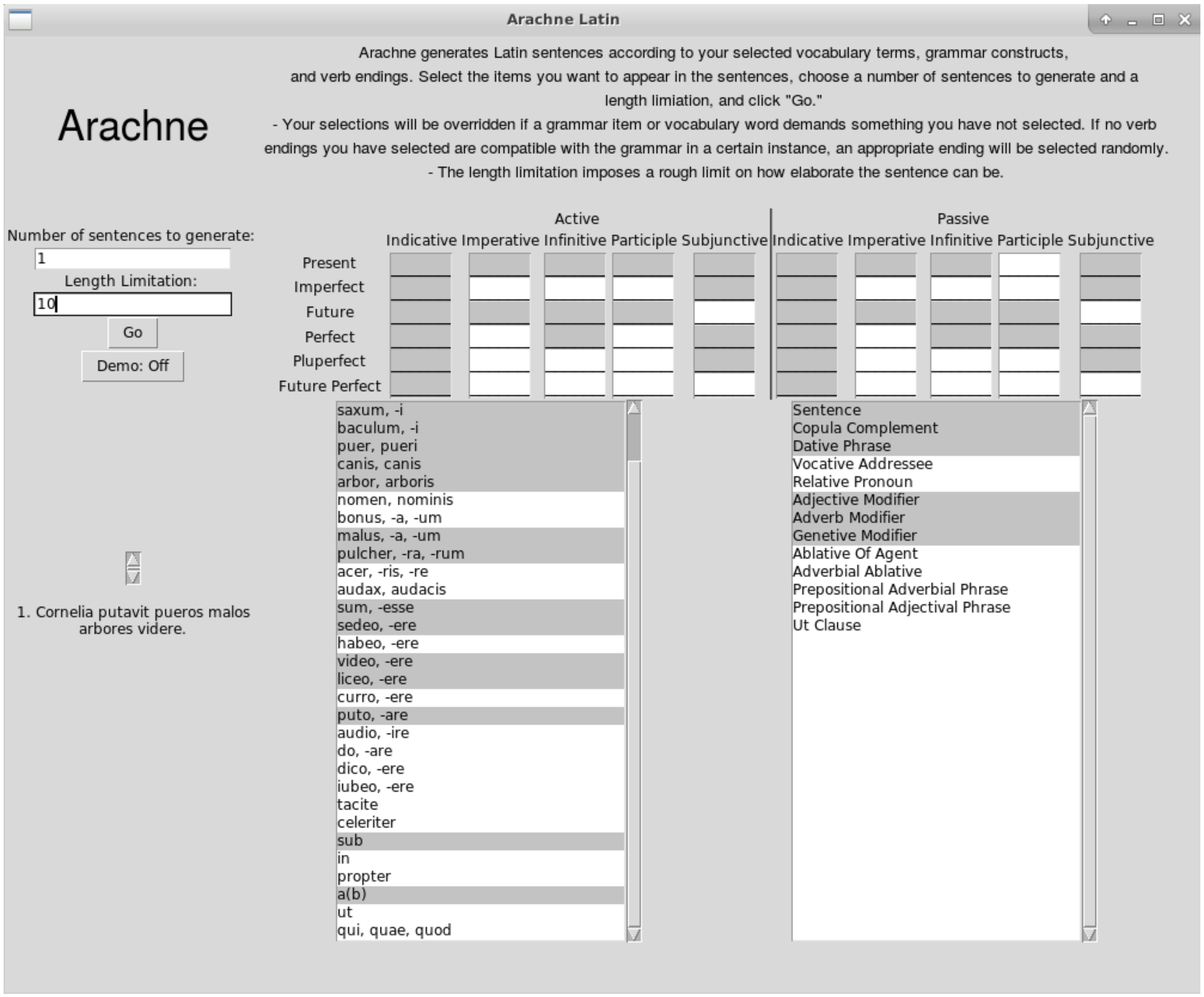
2.2.1 Syntax Tree

The sentence generation process repeatedly adds children to a tree of words according to dependency rules, generating a syntax tree. The algorithm keeps a queue of leaves on the tree that have not been processed. This is the algorithm:

P1. A ROOT Word with parent None is created.
P2. A Complexity integer is set.
P3. A queue is initiated with the ROOT as its only element.

The queue holds new words, along with their parents and depth in the syntax tree.

1. A Word object called Node is popped from the queue.
2. A list of dependencies is initialized as empty.
3. If Complexity > 0, selected dependencies are searched. For each:



Arachne generates Latin sentences according to your selected vocabulary terms, grammar constructs,
and verb endings. Select the items you want to appear in the sentences, choose a number of sentences to generate and a
length limiation, and click "Go."
- Your selections will be overridden if a grammar item or vocabulary word demands something you have not selected. If no verb
endings you have selected are compatible with the grammar in a certain instance, an appropriate ending will be selected randomly.
- The length limitation imposes a rough limit on how elaborate the sentence can be.

Number of sentences to generate: 1
Length Limitation: 10

1. Cornelia putavit pueros malos arbores videre.

	Active					Passive				
	Indicative	Imperative	Infinitive	Participle	Subjunctive	Indicative	Imperative	Infinitive	Participle	Subjunctive
Present	▨	▨	▨	▨	▨	▨	▨	▨	☐	▨
Imperfect	▨	☐	☐	☐	▨	▨	☐	☐	☐	▨
Future	▨	▨	▨	▨	☐	▨	▨	▨	▨	☐
Perfect	▨	☐	▨	☐	▨	▨	☐	▨	▨	▨
Pluperfect	▨	☐	☐	☐	▨	▨	☐	☐	☐	▨
Future Perfect	▨	☐	☐	☐	☐	▨	☐	☐	☐	☐

saxum, -i
baculum, -i
puer, pueri
canis, canis
arbor, arboris
nomen, nominis
bonus, -a, -um
malus, -a, -um
pulcher, -ra, -rum
acer, -ris, -re
audax, audacis
sum, -esse
sedeo, -ere
habeo, -ere
video, -ere
liceo, -ere
curro, -ere
puto, -are
audio, -ire
do, -are
dico, -ere
iubeo, -ere
tacite
celeriter
sub
in
propter
a(b)
ut
qui, quae, quod

Sentence
Copula Complement
Dative Phrase
Vocative Addressee
Relative Pronoun
Adjective Modifier
Adverb Modifier
Genetive Modifier
Ablative Of Agent
Adverbial Ablative
Prepositional Adverbial Phrase
Prepositional Adjectival Phrase
Ut Clause

Figure 1. The GUI

(a) If the dependency admits Node as a parent:
   i. If the dependency is mandatory, it is added to the list of dependencies.
   ii. If the dependency has probability p, it has a p chance of being added to the list of dependencies.

4. For each listed dependency:
   (a) A list Allowed_Child_Entries is initialized as empty.
   (b) For each Entry in the list of lexemes:
      i. If the dependency allows the Entry to be its child, the Entry is added to Allowed_Child_Entries.
   (c) Child_Entry is a randomly selected element of Allowed_Child_Entries.
   (d) The allowed inflections of Child_Entry are found.
   (e) Child is Child_Entry with a randomly selected, allowed inflection.
   (f) Child is added to queue as a child of Node.
   (g) Complexity decreases by 1.
5. If the depth exceeds MAX_DEPTH (here 13), the partially formed sentence is returned with an error.
6. If the queue is empty, the sentence is returned.
7. Else the process is repeated from step 1.

2.2.2 Inflection

The program only uses regular words and the copula (”sum,” ”esse”). The inflectional endings (or forms, in the case of the copula) are stored in a separate file. An ending is stored for each legal combination of part of speech, paradigm number (declension or conjugation), and inflectional parameters. I use the count noun ”inflection” to refer to sets of inflectional parameter values.

A when choosing how to inflect a word, the program finds the intersection between the complements of:

- The inflections that are never possible

- The inflections forbidden by the user
- The inflections forbidden by the Entry
- The inflections forbidden by the dependency

If this is set of available endings is empty, then the restriction by the user is ignored. In either case, an inflection is randomly selected from the resultant set. The stem of the Entry is concatenated with the inflectional ending and is initiated as a Word.

2.2.3 Post-processing

After the syntax tree creation, during which words are inflected, the program is storing a tree of Words. Now the only relevant information in each Word is the text, the canonical direction, and whether commas should be around the Word's phrase. The canonical direction informs the relative position of the children and parent in the sequence. A positive canonical direction for a dependency means that the child will go to the right of the parent, a negative to the left. A child with a greater canonical direction will go to the right of a child with lesser. The final constraint is that the resulting ordered tree has ”gap degree” 0, meaning that all subtrees are convex, or contiguous [3]. No two dependencies that can both exist with the same parent have the same canonical direction, so the two constraints above completely determine the sequence of words.

A comma character is placed before and after any subtree whose root's parental dependency is marked to use commas. Then contiguous sequences of multiple commas are replaced by a single comma, a space is place between each word (after a comma, if one is there), the first letter of the sentence is capitalized, and a period is appended to the end.

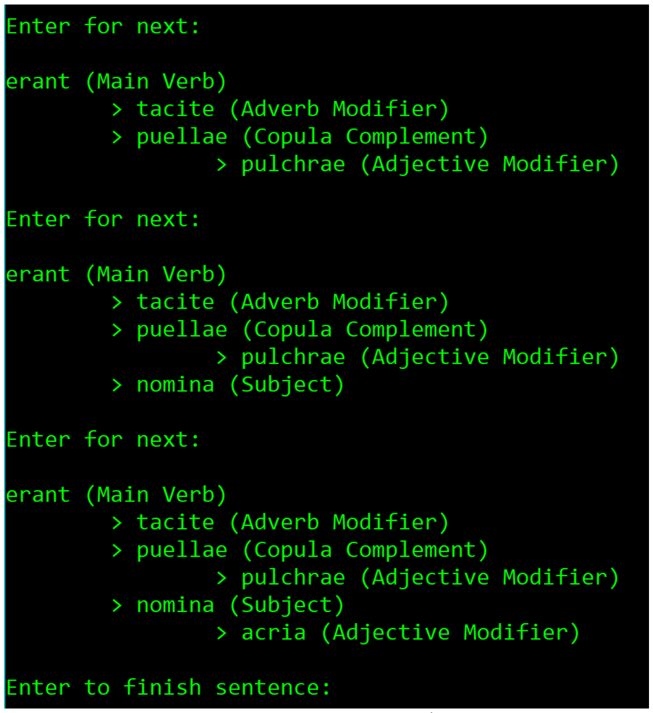
2.3 Data Files

Three main data files informed the sentence generation. The vocabulary file has a lexeme per line, storing the following data as strings:

- Main stems
- Part of speech
- Gender (for nouns)
- Inflectional paradigm number
- Other stems (for verbs)
- Tags in inherent to the lexeme/Entry

The grammar file has a dependency relationship per line, storing the following data as strings:

- Whether the dependency is hidden
- The name of the dependency
- Requirements for parent Word
- Mandatory or probability
- Requirements for the dependent
- Tags given to the dependent
- Inflectional constraints
- Canonical direction



```
Enter for next:

erant (Main Verb)
        > tacite (Adverb Modifier)
        > puellae (Copula Complement)
                > pulchrae (Adjective Modifier)

Enter for next:

erant (Main Verb)
        > tacite (Adverb Modifier)
        > puellae (Copula Complement)
                > pulchrae (Adjective Modifier)
        > nomina (Subject)

Enter for next:

erant (Main Verb)
        > tacite (Adverb Modifier)
        > puellae (Copula Complement)
                > pulchrae (Adjective Modifier)
        > nomina (Subject)
                > acria (Adjective Modifier)

Enter to finish sentence:
```

Figure 2. Demo mode

A hidden dependency is not shown to the user, but selected whenever the dependency above it is selected. The requirements for the parent word and dependency are Python code that is evaluated with their variables referring to the candidate parent or dependent. The truth value returned by the code is whether the candidate is acceptable. Inflectional constraints are implemented as a list of inflectional values, with an element ”None” indicating that an acceptable value may be selected randomly, i.e. is not constrained. For example, the requirement that a noun be Accusative but with unconstrained number would read ”[None, A].”

2.4 Demo Mode

There is an option, selectable in the interface, for ”demo mode.” In demo mode, when the sentence is created, the user can follow the creation of the syntax tree step by step. In the terminal, the program will print the syntax tree (except the ROOT) at each expansion, and requires the user to press ”enter” between each step. The tree is represented as an indented list of words, with children one tier to the right, below the parent (see figure 2). Each word is written with the name of its upper dependency (the dependency to which it is the child). This is approximately the grammatical role the word plays.

3. Results and Discussion

The program succeeds in producing a wide variety of Latin sentences. A single grammatical error may have been detected, but has not been repeated. The constraints are obeyed whenever possible. In short, it is fit for use as a study aid.

The program has the great advantage of being highly expandable. New words can be added very efficiently, with just one modest line of text. Grammar items are manageable to write, and once encoded can apply to the entire lexicon and coexist with other dependencies.

However, the program has a primary issue. It will sometimes, when heavily restricted, fail to produce any sentence despite the possibility of one within the constraints. This is because the creation of the syntax tree cannot rewind more than one step, and that some arbitrary choices are made during the inflection process. For example, a main verb is automatically given a gender, and the nouns are searched for one of that gender to be the subject. Thus it is necessary to choose at least one word of each gender. A similar phenomenon occurs with some inflections. This issue would require substantial paradigm expansion, and is inherent to the straightforward dependency tree creation model.

Other small issues are the labor intensive process of adding the inflections of irregular words, the difficulty in neatly implementing coordinating conjunctions, and the current lack of variation in word order.

4. Conclusions and Open Questions

In conclusion, the program very flexibly and accurately generates a wide variety of Latin sentences. It is functional enough to be of use in a pedagogical context. If it will be used, it may be expanded to include any vocabulary, and much of Latin grammar. Further improvements may be a connection to Wiktionary to automatically import information, and the addition of some user friendly features. For instance, the results could be saveable, the verb endings could be selected by column or row, the vocabulary could be bundled, and settings could be saved between runs.

5. Acknowledgments

I would like to thank to Mr. White, my lab director, for his advice and guidance. I also thank Dr. Gabor for helping me find a project and approving it. Finally, I thank the TJHSST Computer Systems Lab, and all of its students and directors.

References

[1] Rohit Mundra Milad Mohammadi and Richard Socher. Cs 224d: Deep learning for nlp. lecture notes: Part iv. 2015.

[2] J.P. Clackson and G. Horrocks. *The Blackwell History of the Latin Language*. Oxford: Blackwell, 2007.

[3] Manuel Bodirsky, Marco Kuhlmann, and Mathias Möhl. Well-nested drawings as models of syntactic structure. In *In Tenth Conference on Formal Grammar and Ninth Meeting on Mathematics of Language*, pages 88–1. University Press, 2005.