ELFG Language

Antonelli Samuele, Briola Kassandra, Tchigladze Otari

LINK TO THE GITHUB: *https://github.com/Oto0398/ELFG-LANGUAGE*

1. Abstract

This code provides the possibility to create a new language using four different base languages (English, French, Italian, Georgian) for the alphabet.
The peculiarity lies in having different alphabets (Georgian and Latin) that are mixed using the English syntax.

2. Background

Since all three of us speak different languages, we wanted to work on a project in which we had an opportunity to merge them to create a new language.
The initial idea came from other conlangs that implement new alphabets (e.g. Lord of the Rings' various languages). We wanted to merge our languages and alphabets to get a new one.
The name, *ELFG*, comes from the initial letters of the aforementioned languages with the exception of the "i", which was changed into an "l" by mistake. We decided to keep the "wrong" name, since it matched our initial idea of an invented language related to the fantasy world.
Then, the idea evolved by searching for the actual utility of this code. We thought about it as a code that could provide the possibility to create a new language quickly using other languages as source material, relying mostly on google translations and not on large corpus which is difficult to find for every-day users. It could be a good program to use in role playing games (such as Dungeons & Dragons, Pathfinder, Shadowrun, etc), since it gives the possibility to create a new language with fixed words, which do not change as more inputs are added.

3. Software Design, Implementation

3.1. General Steps

Step 1 → Choosing a set of languages
Step 2 → Choosing the features we want to implement
Step 3 → Implementation

3.2 Choosing a set of languages

For the purposes of our task we created four text files in English, French, Italian, and Georgian, each containing different sentences. To ensure accurate translations, we used both our own skills and Google Translate. This approach helped us avoid translation errors, such as 'false friends', and allowed us to compare two different alphabets. However, the primary language is English, so we made a version of the code that uses only the English text file. This version lets hypothetical users add new

sentences directly through a command line, eliminating the need to update each file manually with translations.

3.3 Choosing the feature to implement

To integrate characteristics from each language, we used English for the grammatical structure, as it was the only language known to all group members. For word length, we calculated the average length of each four translated words starting from the English ones, rounding the result. For instance, the word *'small'* translates to '*piccolo*' in Italian, '*petit'* in French, and '*პატარა*' in Georgian. We averaged the lengths of these words with the English word, resulting in a mean length of 5.75, which we rounded to 6.

3.4. Implementation

FIRST PART

Main code:

We developed a code that processes four text files in different languages to create four lists of unique words. This ensures that repeated words or those added later won't have their translations recalculated. The primary list is in English; for each English word, the code generates translations in French, Georgian, and Italian, along with the original English.

The code also cross-references translations with other lists to identify any ambiguities, helping to spot potential Google Translate errors. The result is a json file with all the English words and their translations in each language.

Second version of the code:

The code has the same base idea, but it uses only the English text file and just creates a list of words for this one.

The main change here is that the code asks you if you wish to add a new sentences into the text file with a small system of yes/no answer before running:
- if yes press 1 and add a new sentence
- if no press 2 and it will run the rest of the code
This system is a loop therefore it will keep asking if you wish to add more sentences until you say no.

When running the code, the created json files for both the conditions is called `translations.json`, and it contains only the translations in the four languages of each word contained in the sentences.

SECOND PART

At this point we needed another part of the code to create the words in the new language. To do so we followed these steps:
1. For every English word in the json file, the code will compute the length of the newly created word by taking the average of the lengths of the same word in the four languages.

2. Also, it will take all the unique characters that exist in the four translations and put

them into a list to create a new alphabet for the new words, therefore, every new word will have its own alphabet based on the corresponding list.

3. Given the length and the alphabet, the characters for the new word are chosen randomly from the corresponding list.
Importantly, even if the list (alphabet) for each word contains unique characters, you can still have repetition of the same character inside the new word.

4. Each newly created word is stored inside a new json file, named elfg_translations.json and is saved, meaning that each time the code is run it will check if the word has already been created, if it has then it won't replace it with another translation. By implementing this, we allowed the created language to be fixed, even when modifying the initial texts or adding sentences.

Example of the described process:

Word: small

English: small
Italian: piccolo
French: petit
Georgian: პატარა

Mean length of the new word: 6 characters
Alphabet: {s, m, a, l, p, i, c, o, პ, ტ, ა, რ}

New word: miაპlpi

The code's results can differ depending on whether the two mentioned json files are present in the environment. Without these files, running the code will generate a new language each time. However, by sharing the json files, users can maintain and use the same language consistently.

This feature is crucial for the code's application, such as in creating languages for various purposes, including gaming. It allows for the continuous expansion and improvement of a language's vocabulary through shared dictionaries.
On the other hand, users have the option to create an entirely new language by simply removing the existing dictionaries.

4. ***Testing***

To ensure the code functions correctly under various conditions, we implemented several testing strategies:

1. We ran the code adding new words to see if it's working as intended.
2. We printed every single step to see if the code was working correctly.

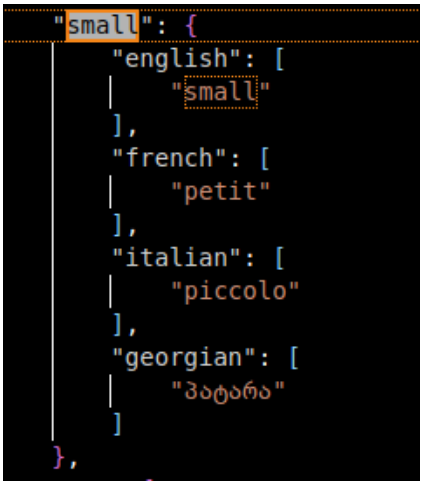
3. Finally, we tested the complete code on different computers, with different operating systems (Linux, MacOs).

5. ***Results:***

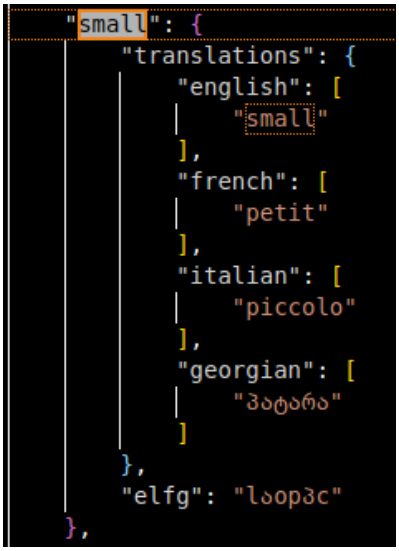
This code provides the possibility to create new words from a set of languages. You can observe its direct functioning by running it. Additionally, once the json files are created, you can modify the initial texts.
By adding the same translated sentence to each of the four texts or only to the English one in the alternative version and then running the code again, both json files will be modified, adding the new word or set of words to the end of the dictionary.
To generate brand new words, it will be sufficient to delete the json files and run the code again.
Alternatively, if you wish to retain your results for future use, you can download the dictionaries. Next time you'll need the language, you will simply need to re-upload the json files before running the code, and it will still work the same way.

The program's key feature is its versatility of usage and the persistence of results, which can be expanded by adding new inputs.
The advantage of our code is that even though it doesn't create a complex language with its own syntax, it is quite resilient to errors or problems related to grammar. For example, even if a language lacks a translation for a word like 'the' (as in Georgian), the code will continue to operate and create a new word using the present information it is presented with. This way any language could be potentially implemented, despite grammatical differences.

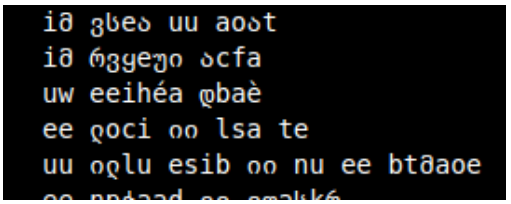
When running the code, you are going to get three different output:
1. A json file with the translation of every English word in English, French, Italian and Georgian.



2. A json file containing the same as above and also the translation of the word in the ELFG language



3. A text file containing the sentences translated in the ELFG language



6. **Conclusions and further implementations:**

6.1 Possible use

As proposed above, the present code could be a good starting point for a more complex code. The idea behind it is to provide a possibility for each user to choose any set of languages as a source.

In this case English was chosen as the chassis, since it's an international language; but potentially the base of the code could be the native language of the users, making the process easy.

Also, the dictionaries of a new language can be shared between users/players and improve the vocabulary of this language each time. The language is simple and relies on the grammar of the primary language of choice, therefore, everyone can use it without learning any rules.

6.2 Possible further implementations

Our initial idea was to also implement the IPA (International Phonetics Alphabet), providing the possibility to read the words with correct pronunciation, even when it contained letters from unknown languages, but this idea was called off because of the issues encountered *(look at 7.1 "Issues encountered")*.
Ideally the IPA would've been treated as an alphabet linked to each letter as their pronunciation in the language they come from. When the program randomized the letters inside the new word, it would've also retrieved the corresponding IPA sound, proposing the new word and along with it its sound structure.
If we could have been able to implement this feature, we also would have proposed a step further, considering the idea to use a voice synthesizer that enables the ones who do not know how to read IPA to read the new language out loud.

Moreover, we decided to take English as a source language in this case, borrowing its word order and other grammatical features but, as stated above, an interesting step further would be to allow users to choose a different language as the basis of the structure.

If we wanted a language completely independent from the given bases, we would need to add syntactic rules also.
Here we decided to keep the code as accessible to use as possible, adding only one morphological rule consisting in the length of words.
But from the perspective of creating a complete language, syntactic rules would certainly be required.

Lastly, it would be good to implement some kind of name-related feature, that would provide a different treatment for people's names. In our code the names behave like any other words, but since we created a new alphabet for each word, the names are not really different from the originals. Eg "John" can become "jʒɤɓn" which still look similar, just mixing alphabets.
It could be interesting to have two additional features:
- Either the possibility to specify people's names as different categories, keeping them equal as their initial form.
- Or the possibility of creating a brand new one that is substantially different.

7. **Critical evaluation:**

7.1 Issues encountered

There were a few issues encountered while working on the project.
The first and main issue was the use of the googletrans module. At first we just installed the most recent version of the module, but the module 'translate' couldn't be imported or found while running the code ,therefore, by checking across different forums many people had the same issues and found a solution to use the 3.1.0a0 version of the library, once installed the code worked perfectly fine.

Another issue occurred during one of the first testing phases. Google translator module worked only with a specific Python Version (8.11) for one of the users. However, by choosing this specific version, other parts of the code were not recognized.
This issue was solved by creating a brand new folder containing all the intended

material and a new Conda environment, containing only the specific googletrans version proposed above.

Lastly, as mentioned already, we had problems with adding the IPA to the code.
Even if the Googletrans library contains a Google IPA source, it is now a paid service,
At this point we tried different alternatives to get it, but each source encountered issues in recognizing all three languages (which have different alphabets but also share an alphabet using different sounds).
We ended up having json dictionaries for IPA that were empty.
Perhaps having more time/resources or trying different libraries this problem could be solved. Unfortunately, due to our deadline and our lack of expertise in coding this problem couldn't have been solved for now.

7.2. What we were not able to do

We weren't able to get conjugated words/verbs with googletrans module.
When a verb is conjugated (e.g. '*ho'* as *'I have'* present singular), the translation given by the module is in the infinitive form by default, and even when it checks with the other texts if it's correct, it adds the google translation word to the dictionary and not the word that exists in the other text files.
This problem might not be central in the code for the proposed use, since because of the variety across languages it would be difficult to find syntactic rules that can be applied to each one independently, for example building tenses in verbs, but it was still an issue we couldn't solve.

Moreover we found some limitations due to the lack of syntax base of the code:

1. **Plurals:** since our code is a direct translation of English words and doesn't have specific grammar rules, the plural of English typically is the word + 's'. The singular and plural words are thus considered as different. This way the plural form will have a brand new created word in ELFG and not a morpheme that is added to the word.
2. **Apostrophes**: in the French language "I have" is "J'ai". Since the computer takes every word as a set of strings separated by 'spaces', it considers "j'ai"as one word. To fix this issue we had to put " je ai" to separate the words. Anyway, this is not a big problem, since the apostrophe is just the silent 'e'.
3. **Function words**: we have encountered another small problem with some words that do not exist in a language. In this case some function words, like "the", do not exist in Georgian.

This can also be seen as a strong point: since the code uses the length and the available characters, the words can still be created, eliminating problems encountered when having radically different languages.

Again, these problems are related to the choice between usage simplicity and lexical complexity. Our goal was not to create a new language with fixed grammar, but a language flexible and easily edited over time and by different people.
Anyways, those problems might be encountered during usage and they would require further attention.

8. **Division of work**

   During the first phase of the work, the conceptual one, where we had to come up with an idea and think how to implement it, we had meetings on a weekly basis to discuss. The first meeting revolved around which kind of rules we wanted to implement, when we would be satisfied with the results, the possible usage and outcome of the work. A lot of time was spent on thinking about syntactic-morphological aspects of the program, since we wanted to create something that could pass as a real language. We also had to set boundaries and rules as to how the words would be formed.

   During the coding part, we were all in different places and could not arrange in-person meetings.
   During this time we still worked by ourselves trying to implement our ideas and seeing what would work.
   The spine of the project was proposed and implemented by Kassandra, which shared with Samuele and Otto each time the outcomes, trying to solve them during group calls or on our own.
   After the winter break we started meeting again in person and tested the code on different systems, searching for possible limits, problems or improvements.
   Once we were happy with the work we created a GitHub account to upload the code, then we wrote down the Readme and the report.
   In this last phase we spent our time working together, so again the work was evenly split.