Assignment 2: Lemmatizing English Verbs with FSTs

In this assignment, you will build a basic lemmatizer for English verbs using OpenFST. A lemmatizer is a program that takes words and returns their lemma—roughly, the form of the word you would find in a dictionary. For English verbs, lemmas are the INFINITIVE form of the verb, the form that comes after *to* (as in *to have* or *to thrash*). You will be given a list of PARADIGMS—lists of the forms of each verb—for many known verbs, including most exceptional (or irregular) verbs. Your task will be to construct a finite-state transducer (FST), using OpenFST, which takes any of those in vocabulary forms, or any regular out-of-vocabulary (OOV) verb, and returns one or more possible lemmas.

If your solution is not implemented as a FST, using OpenFST, you will receive no credit. All string operations on the input must be done with a FST.

1 Learning objectives

1. Students will learn how finite-state transducers and acceptors work, and how they can be combined and modified with operations like composition, concatenation, union, and inversion.
2. Students will learn how to develop and test a practical rule-based NLP system.
3. Students will learn the issues involved in using computational tools to analyze linguistic morphology.

2 The Problem

Your goal is to map verb forms like *squigging* to lemmas like *squig.*

You will have to solve lemmatization for two cases:

1. In vocabulary lemmatization - based on the data in Section 3.
2. Out of vocabulary lemmatization

At first, lemmatization might sound trivial. If it's in vocabulary, look up the lemma. If it's out of vocabulary, chop off the affixes. In reality, lemmatization is more complicated, as your textbook points out. There are two issues:

1. How morphemes fit together, or MORPHOTACTICS
2. Spelling rules that account for differences in how a particular morpheme is realized, or ALLOMORPHY

2.1 Known cases (In vocabulary)

For most words that would be found in a dictionary, you will know the lemma because you will be provided with it in the data (in Section 3). For each verb, you will be given the five forms of English verbs shown in Table 1. With the data, you can create a transducer for these words that maps directly from the inflected forms to lemmas.

	Suffix	Example 1	Example 2	Example 3	Example 4
Infinitive	-ϵ	give	walk	fuss	slug
Gen. Present	-ϵ	give	walk	fuss	slug
3Sg Present	-s	gives	walks	fusses	slugs
Past	-ed	gave	walked	fussed	slugged
Pres Part.	-ing	giving	walking	fussing	slugging
Past Part.	-ed/-en	given	walked	fussed	slugged

Table 1: English verb inflection

There are five forms of English verbs you have to worry about, which are shown in Table 1 (the general present tense form is always the same as the infinitive, so we get that for free).

DO NOT hard code a FST for each verb by hand. Use code to generate FSTs for each verb in the dataset, then union together the FSTs for your in vocabulary transducer. It may be helpful to work out a few examples by hand to figure out how to programtically generate FSTs.

It will then be possible to combine this transducer, via union, with the out of vocabulary transducer. However, you must do something special at this point: indicate which forms are "known" and which are "guessed". You will do this by appending the string "+Known" via the transducer for in-vocabulary words. You should also append the string "+Guess" for out of vocabulary words. That way, when the whole FST produces an ambiguous output, a downstream disambiguator can know which lemma is more likely to be reliable.

For lemmatize, your in vocabulary FST should take in inputs like `giving<#>` and output `give+Known`

2.2 Morphotactics (Out of vocabulary)

The morphotactics of English verb inflection are very simple: a stem (a root plus any derivational affixes) is followed by a single inflectional suffix. Again, we care about the five forms of English verbs shown in Table 1.

To lemmatize a verb like *squigs/squigged/squigging*, we have to posit lemmas with shapes like *squig* which can be followed by suffixes like *-s*, *-ed*, and *-ing*. The FST that collects these statements about what morphemes can occur in what sequences is called a LEXICON FST.

Since we do not know the data for all verbs, we must make our best guess for what the suffixes are. For lemmatize, your general morphological FST should separate out the suffix with morpheme boundaries, taking in inputs like `squigging` and outputting `squigg`< ∧ >`ing<#>`.

2.3 Allomorphy (Out of vocabulary)

However, there is a problem. In *squigged* and *squigging*, there is a extra "g". Under certain circumstances, the last letter in a stem is doubled when *-ing* or *-ed* is added. This is an example of allomorphy: our stem has two forms, *squig* and *squigg*. This is a spelling rule or orthographic rule. There are a number of other important spelling rules listed in your textbook (p. 20). Allomorphic rules of this kind can also be modeled using an FST. These FSTs can be combined, via union, to produce a single FST that maps from the surface forms of words to a form in which all morphemes have only one representation. In fact, this FST can be composed with the lexicon FST to form a big FST that maps between inflected forms and lemmas.

Your lemmatizer is expected to handle the following orthographic rules:

1. **Consonant doubling**: one-letter consonants doubled after VC (where C is consonant and V is vowel) and before *-ing* or *-ed*. *beg* ↔ *begging* ("e" is V, "g" is C)
2. **E deletion**: silent *e* is deleted before *-ing* or *-ed*. *make* ↔ *making*
3. **E insertion**: *e* is inserted after *s*, *z*, *x*, *ch*, or *sh* before *-s*. *watch* ↔ *watches*
4. **Y replacement**: *y* changes to *ie* before *s* and *i* before *ed*. *try* ↔ *tries*
5. **K insertion**: after verbs ending in *c*, insert *k* before *-ed* and *-ing*. *panic* ↔ *panicking*

Your Orthographic FST should correct a morpheme separated verb by taking in inputs like `squigg`< ∧ >`ing<#>` and output `squig`< ∧ >`ing<#>`.

2.4 Putting it all together

The in-vocabulary FST and out-of-vocabulary FST combine via union to form one giant FST - your lemmatizer. We have already done the work of combining the various FSTs using union and composition for you (see the flowchart at the end to understand how it is put together). Your task is to implement the individual FSTs:

1. Set up your environment (see sections 4, 5, and 6). **Do this ASAP.** Many students report difficulties setting up OpenFST.
2. Implement `fsts/pre-process.txt`. It should accept an input like `going` and generate `going<#>`.
3. Implement your in-vocabulary FST in `lemmatizer.get_in_vocab_fst()`. Your code should generate an FST for each verb in the dataset, and then union them together. Your resulting FST should accept an input like `giving<#>` and generate `give+Known`.
4. Implement `fsts/general-morph.txt`. This is your general morphological FST. It should accept an input like `going<#>` and generate `go<` $\wedge$ `>ing<#>`. Your FST should detect the following suffixes: *-ed, -s, -en,* and *-ing.* It will be easier to generate it inverted, so we invert your FST in lemmatizer.py.
5. Implement your five orthographic rule FSTs from Section 2.3 in the appropriate files in `fsts/`. You may find it easier to write the inverted FSTs instead. If you do, be sure to uncomment the lines that perform the inversion. You can also check the `fststr` documentation on how to use the consonant and vowel symbols (see Section 5). An incomplete implementation of the e-insertion rule is provided for you: you will, at a minimum, need to modify it to account for *ch* and *sh.* Finally, if a rule doesn't apply in an FST, you have the option of accepting or rejecting the input, depending on your implementation. Remember that all of your orthographic rules are unioned together in the final FST.
6. Implement `fsts/post-process.txt`. It should accept an input like `go<` $\wedge$ `>ing<#>` and generate go+Guess.

Once you have implemented your FSTs, the provided code should properly merge them to form the lemmatizer. See Section 6 for details on how to test it.

3 Data

The data can be found in the handout as `in_vocab_dictionary_verbs.txt`. Each line is of the format: `lemma,verb_in_form,verb_form_name,`

For example the line, `give,gave,Past,` represents the lemma `give` and it's past tense `gave`. The data comes from the Unimorph english dataset.

4 OpenFST and Python Installation

`OpenFST` (`http://www.openfst.org/twiki/bin/view/FST/WebHome`) is a widely used library implementing FSTs. `OpenFST` is written in C++ and has a well-documented interface. OpenFST requires either OSX or Linux to install. Students have reported the latest version has issues with OSX, please use version 1.7.3 and follow the below instructions very carefully. Note that AFS does not have a sufficient gcc version for OpenFST.

NOTE: If you run into issues installing OpenFST, the easiest way is to install it on a fresh GCP Ubuntu VM. Please see the tutorial in the `google-cloud-installation.pdf` handout. It will help you create a VM and upload an installation script that installs everything for you. If you wish to manually install, you can follow the steps below.

We will be using Python3 to interface with OpenFST. The included python wrapper `pywrapfst` (`http://www.openfst.org/twiki/bin/view/FST/PythonExtension`) is quite low-level. So, we will be using our own interface `fststr`, more details on that later!

1. Download `OpenFST` (version 1.7.3) from `http://www.openfst.org/twiki/pub/FST/FstDownload/openfst-1.7.3.tar.gz`
2. Untar OpenFST folder: `tar -xvf openfst-1.7.3.tar.gz`
3. Make sure to have g++ $\geq$ 4.9 for Linux and XCode $\geq$ 5 for OSX. Check out how to update g++ for linux at `https://askubuntu.com/questions/466651/how-do-i-use-the-latest-gcc-on-ubuntu/581497`
4. Follow the instructions in `INSTALL`
   (a) `./configure --enable-far`
   (b) `make`
   (c) `make install`
5. Install pywrapfst via PyPI: `pip3 install openfst`

To provide an interface that makes `OpenFST` better suited for processing strings, we created `fststr` (pronouced "fast-strong"). With `fststr`, creating FSTs is much easier, just define a text file or string with each line representing an arc of: `previous_state next_state input_label output_label`.

We expect you to use `fststr` to implement FSTs. You can view its documentation and source code at `https://github.com/dmort27/fststr`. Check out the examples of mapping a diagram to fststr code.

1. Install fststr: `pip3 install fststr`

5 Creating FSTs in Python: fststr

Examples are in the `fststr` repo under `examples/FSTs`. We will examine `e-insertion.txt`, which is based off of the e-insertion FST from the textbook. (also in the slides). The FST takes in morphologically separated inputs like `fox`< ∧ >`s`<`#`> and outputs `foxes`<`#`>. Here's a quick run through:

- Each line of the file represents information about the FST.
- The first line `0` represents that q0 is a final state
- The second line `0 0` <`other`> <`other`> represents an arc from q0 to q0 with the value <`other`>`:`<`other`>
- The fifth line `0 1 z z` represents an arc from q0 to q1 with the value `z:z`
- Note: The textbook does not account for e-insertion rule with `ch` or `sh`, we also do not have it in our example. You must include it with `ch` and `sh` for this assignment.
- We have provided steps at the bottom of lemmatizer.py to demonstrate how to run the existing `e-insertion.txt` FST. You should comment it out while submitting to gradescope.

6 Testing

In `lemmatizer.py`, there exists a Python module called `Lemmatizer` exporting two functions: `lemmatize` and `delemmatize`. If you have implemented your FSTs correctly, `lemmatize` should apply the transducer "up," from the inflected form to the lemma. Correspondingly, `delemmatize` should apply the transducer "down," from lemma to inflected forms. Both functions take one argument, a string. Your FST may have ambiguous outputs, so the outputs of both functions will be a set of strings.

For `lemmatize`, an example input is "giving" and we expect the output set to contain "give+Known" and "give+Guess".

For `delemmatize`, an example input is "give+Guess" and we expect the output set to contain "give", "giving", "gived", "gives", "giveing", "giveen", and "giveed".

You are also encouraged to debug by testing your individual FSTs. You can do this using the available functions in the module. An example is given at the end of the file.

7 Submission

Run `make submit` with the given `Makefile`. This should zip up your `lemmatizer.py` and related FST files.

Do not include any print statements in your submission. Gradescope will throw an error.

8 Hints and Suggestions

- Read this handout carefully. This assignment is not easy, and we give you a ton of information on how to build your lemmatizer.
- You should draw out your FSTs before implementing them in OpenFST. Drawing out FSTs makes them much easier to debug.
- If you having failing test cases, start by testing your component FSTs. You can test each FST module and make sure it is functioning properly.
- You can union two compiled FSTs by `fst1.union(fst2)`.
- FSTs are invertible. Sometimes it is easier to define an FST mapping one direction (e.g. from the lemma to the surface form), then invert it (rather than writing it so that it maps from surface form to lemma in the first place).
- OpenFST does not provide a very rich language for talking about subsets of alphabets. `fststr` provides this in a limited way, with the `<other>, <C>, <V>` symbols and the associated functions. You may need to write code that generates arcs of your FSTs to avoid the tedious task of hand-coding transitions. You can do this by iterating over your list of symbols.
- See the last page to understand how all the component FSTs are merged together to form one large FST.

9 Evaluation Criteria

Note: For some tests, we will not be using real English verbs, but we are looking for lemmatization based on English grammatical rules we described above. You will be given details about the test cases you fail, and you will have infinite submissions.

1. Performance in 5 basic lemmatizing tests for allomorphic rules. (5 points)
2. Performance on a held-out set of 70 items (15 in-vocabulary items and 55 out-of-vocabulary items) measured with F2 (70 points).
3. Inverted performance (mapping lemma to other forms) on 25 items (5 in-vocabulary and 20 out of vocabulary) measured with F2 (25 points).

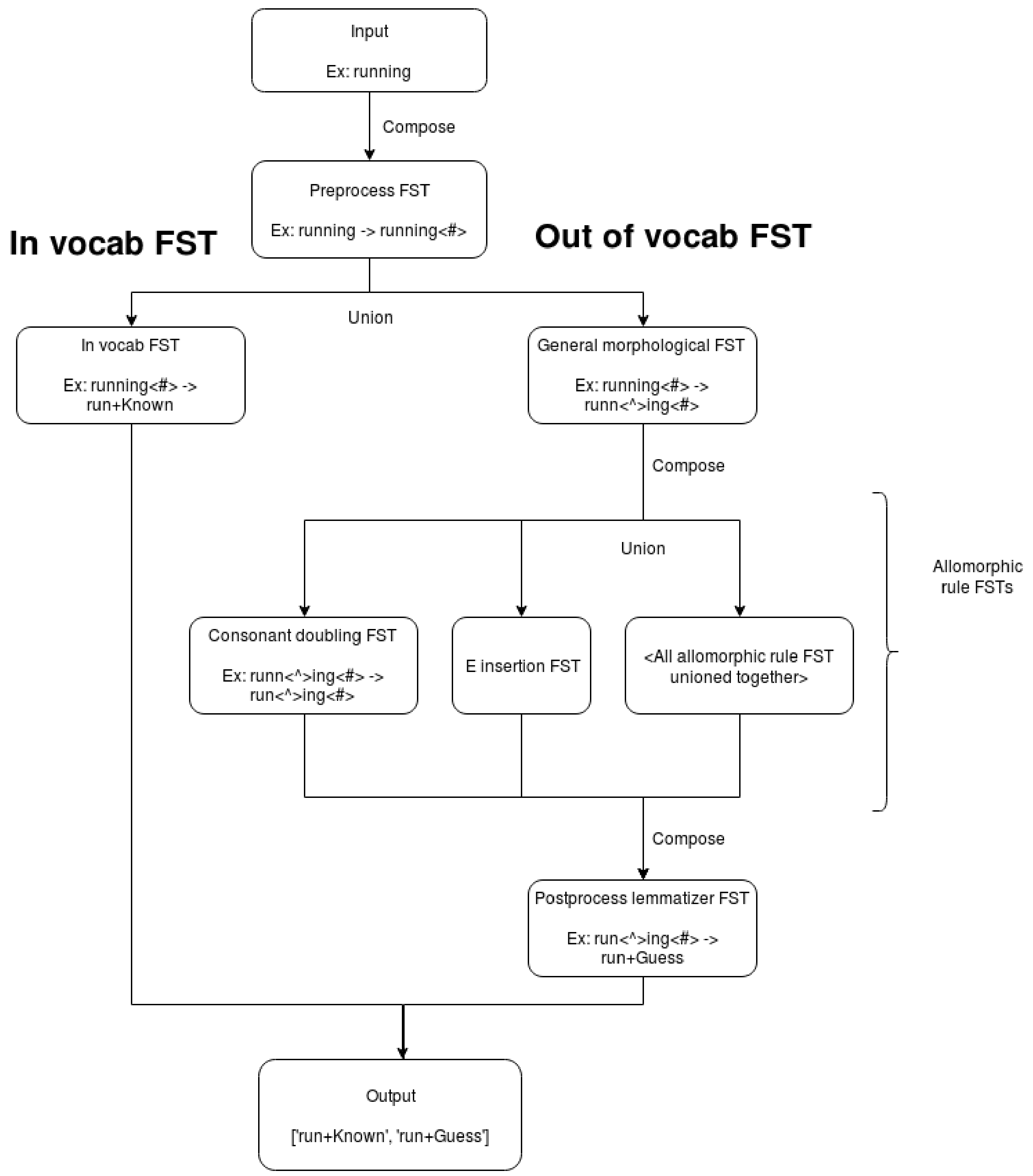
Input
Ex: running
Compose
Preprocess FST
Ex: running -> running<#>
In vocab FST
Out of vocab FST
Union
In vocab FST
Ex: running<#> -> run+Known
General morphological FST
Ex: running<#> -> runn<^>ing<#>
Compose
Union
Allomorphic rule FSTs
Consonant doubling FST
Ex: runn<^>ing<#> -> run<^>ing<#>
E insertion FST
<All allomorphic rule FST unioned together>
Compose
Postprocess lemmatizer FST
Ex: run<^>ing<#> -> run+Guess
Output
['run+Known', 'run+Guess']

Figure 1: Example FST flow for `lemmatize`