601.465/665 — Natural Language Processing
Assignment 6: Finite-State Programming[*]

Prof. Kevin Duh and Jason Eisner — Fall 2019
Due date: Tuesday 3 December, 11:59pm

This assignment exposes you to finite-state programming. You will build finite-state machines automatically using open-source toolkits. You'll also get to work with word pronunciations and character-level edits.

Collaboration: ***You may work in pairs on this assignment,*** as it is fairly long and involved. That is, if you choose, you may collaborate with one partner from the class, handing in a single homework with multiple names on it. However:

1. You are expected to do the work *together*, not divide it up: your solutions should emerge from collaborative real-time discussions with the whole group present.
2. Your `README` file should describe at the top what each of you contributed, so that we know you shared the work fairly.
3. Your partner **must not be the same partner** as you had for HW5. Make new friends! :-)

In any case, observe academic integrity and never claim any work by third parties as your own.

Reading: There is no separate reading this time. Instead, we'll give you information and instructions as you work through the assignment.

What to hand in (via Gradescope): As usual, you should submit a `README.pdf` file with answers to all the questions in the text. We'll also ask you to submit a `.zip` archive of all the grammar files you create:

File	Questions	Should contain
`binary.grm`	1, 2, 6	`First, Second, Disagreements, Triplets, NotPillars, Oddlets, WFlip, WeightedMultipath`
`rewrite.grm`	3, 4	`Cross`, `BitFlip1`, `BitFlip2`, `Parity1`, `Parity2`, `Parity3`, `UnParity`, `Split`
`chunker.grm`	5a	`NP, MakeNmod, TransformNP, BracketTransform, BracketResults`
`noisy.grm`	7, 8, 9	`CompleteWord, DelSpaces, SpellText, Generate, RandomWord, Spell` (revised), `PrintText` (revised)

Software (not Python this time!):

- **OpenFST** is a very efficient C++ toolkit for building and manipulating semiring-weighted FSMs. You can use the C++ API to directly specify states, arcs, and weights, or to combine existing FSMs through operations like union and composition. You can also store these FSMs in `.fst` files and manipulate them with command-line utilities like `fstunion` and `fstcompose`.

  A symbol table (`.sym` file, or part of some `.fst` files) specifies the internal representation of the upper or lower alphabet. E.g., the integers 1, 2, 3, ... might internally represent the letters `a`, `b`, `c`, ... or perhaps the words `aardvark`, `aback`, `abacus`, ... OpenFST uses these integers to label arcs in its data structures and file format (ϵ arcs are labeled with 0). It is only the symbol table that tells what a given FSM's integer labels are supposed to *mean*.

However, we will not be using OpenFST directly (nor its Python interface, Pynini). Instead, we will use two packages that provide a fairly friendly interface to OpenFST:

[*]Many thanks to Frank Ferraro, who co-wrote this assignment and wrote the accompanying scripts, and to Jason Eisner.

- **Thrax** is an extended regular expression language that you can use to define collections of finite-state machines. A Thrax grammar can be created with any text editor and is stored in a `.grm` file. The Thax compiler *compiles* this into a `.far` file—an "**f**st **ar**chive" that contains multiple named OpenFST machines and symbol tables.

- Since regular expressions are not good at specifying the topology of n-gram models, there's also the **NGram** toolkit, which builds a n-gram backoff language model from a corpus. It supports many types of smoothing. The resulting language model is represented as a weighted FSA in OpenFST format.

First, get set up!

Optional: OpenFST, NGram, and Thrax are installed on the `ugrad` machines (as well as the graduate network). It's probably easiest to do the assignment there. But if you would prefer to install a copy on your own machine,

1. Download and install OpenFST:
   http://www.openfst.org/twiki/bin/view/FST/FstDownload

   - **Important:** Make sure you run `configure` with the flag `--enable-far=yes`. If you don't, Thrax won't work!
   - You should also use the flag `--enable-ngram-fsts=yes`.
   - Do **not** use `--enable-static=no`.
   - After installation, you may need to set the environment variable `LD_LIBRARY_PATH` to where the OpenFST libraries are.

2. Download and install Thrax:
   http://www.openfst.org/twiki/bin/view/GRM/ThraxDownload

3. Download and install NGram:
   http://www.openfst.org/twiki/bin/view/GRM/NGramDownload

4. To view drawings of FSMs, download and install graphviz:
   http://www.graphviz.org/Download.php.
   On Linux systems, you can just do `sudo apt-get install graphviz`.

5. Download a copy of the assignment directory `hw-ofst`, either from the ugrad network[1]

Look in the `hw-ofst` directory.[1] Our scripts are in the `bin` subdirectory, which you should probably add to your `PATH` so that you can execute these scripts without saying where they live. Run the following command (with the `bash` shell) (and maybe put it in your `~/.bashrc` so that it will be executed automatically next time you log in).

```
export PATH=${PATH}:/usr/local/data/cs465/hw-ofst/bin
```

We've given you a script `grmtest` to help streamline the compilation and testing of Thrax code. Its usage is:

```
grmtest <grm file> <transducer_name> [max number output lines]
```

[1] `/usr/local/data/cs465/hw-ofst/`. You can copy this directory to your local machine or symlink to it on `ugrad`.

This script compiles the specified `.grm` file into a `.far` file (using a makefile produced by `thraxmakedep`), and then passes the standard input through the input through the exported FST named by `<transducer_name>`. You'll get to try it out below.

Warning: If the output string is the empty string ϵ, then for some reason `grmtest` skips printing it. This seems to be a bug in `thraxrewritetester`, which `grmtest` calls. Just be aware of it.

1. Now get to know Thrax. We highly recommend looking through the online manual[2] and perhaps the commented examples that come with Thrax.[3] The following tutorial leads you through some of the basic FSM operations you can do in Thrax.

   (a) Let's first define some simple FSMs over a binary alphabet. Type the following declarations into a new file `binary.grm`.

   ```
   Zero = "0";
   One = "1";
   Bit = Zero | One;
   export First = Optimize[Zero Zero* Bit* One One One One?];
   ```

   This defines four named FSMs using Thrax's regular expression syntax (http://www.openfst.org/twiki/bin/view/GRM/ThraxQuickTour#Standard_Library_Functions_Opera).[4] Each definition ends in a semicolon. The first and second FSMs accept only the strings `0` and `1`, respectively. The third defines our entire alphabet, and hence accepts either `0` or `1`. The fourth accepts some subset of `Bit`$*$.

   We can compile this collection of named FSMs into a **f**st **ar**chive (a "`.far` file"). More precisely, the archive provides only the FSMs that have been marked with an `export` declaration; so here `Zero`, `One`, and `Bit` are just intermediate variables that help define the exported FSM `First`.

   i. Try compiling and running it using our `grmtest` script:

   ```
   $ grmtest binary.grm First
   [compiler messages appear here]
   Input string: [type your input here]
   ```

   The FSA `First` is interpreted as the identity FST on the corresponding language. So entering an input string will transduce it to itself if it is in that language, and otherwise will fail to transduce. Type `Ctrl-D` to quit.

   You'll get an error if you try running `grmtest binary.grm Zero`, because `Zero` wasn't exported.

1 ii. What language does `First` accept (describe it in English)? Why are `0` and `1` quoted in the `.grm` file?

   iii. Let's get information about `First`. First, we need to extract the FSA `First` from the FST archive:[5]

   ```
   $ far2fst binary.far First
   ```

   Now use the `fstinfo` shell command[6] to analyze `First.fst`:

   ```
   $ fstinfo First.fst
   ```

2 Look over this output: how many states are there? How many arcs?

[2] http://www.openfst.org/twiki/bin/view/GRM/ThraxQuickTour

[3] `/usr/local/share/thrax/grammars/` on the ugrad or grad machines.

[4] Whereas XFST defines many special infix operators, Thrax instead writes most operators (and all user-defined functions) using the standard form `Function[arguments]`. Thrax uses square brackets `[]` for these function calls, and parentheses `()` for grouping. Optionality is denoted with `?` and composition with `@`. There is apparently no way to write a wildcard that means "any symbol"—you need to define `Sigma = "a"|"b"|"c"|...` and then you can use `Sigma` within other regular expressions.

[5] Use `far2fst binary.far` to extract *all* exported FSTs, or `far2fst binary.far First Second` to extract multiple ones that you specify.

[6] http://man.sourcentral.org/f14/1+fstinfo

iv. Optionally, look at a drawing of First (as an identity transducer over the alphabet $\{0, 1\}$):

```
$ fstview First.fst
```

Note that `fstview` is a wrapper script that we are providing for you.[7] The picture will take a few seconds to appear if the graphics pixels are being sent over a remote X connection.[8]

(b) Now let's look at equivalent ways of describing the same language.

i. Can you find a more concise way of defining `First`'s language? Add it to `binary.grm` as a new regexp `Second`, of the form

```
export Second = Optimize[ ...fill something in here... ];
```

Run `grmtest` to check that `First` and `Second` seem to behave the same on some inputs.

ii. Here's how to check that `First` and `Second` really act the same on *all possible inputs*—that they define the same language:

```
export Disagreements = Optimize[ (First - Second) | (Second - First) ];
```

3 If `First` and `Second` are equivalent, then what strings should `Disagreements` accept?

To check that, run `fstinfo` on `Disagreements.fst`. From the output, can you conclude that
4 `First` and `Second` must be equivalent?

Note: The `fstequal` utility is another way to check:

```
if fstequal First.fst Second.fst; then echo same; else echo different; fi
```

One way to program `fstequal` would be to construct the `Disagreements` FSA.

(c) You might have wondered about those `Optimize[ ... ]` functions. The Thrax documentation notes that `Optimize`

> ...involves a combination of removing epsilon arcs, summing arc weights, and **determinizing and minimizing** the machine ... [Details are here.]

To find out what difference that made, make a new file `binary-unopt.grm` that is a copy of `binary.grm` with the `Optimize` functions *removed*. Then try:

```
grmtest binary-unopt.grm First   # and type Ctrl-D to exit
far2fst binary-unopt.far
fstview First.fst Second.fst Disagreements.fst
```

Questions:

i. Although `First` and `Second` may be equal, their unoptimized FSMs have different sizes and different
5 topologies, reflecting the different regular expressions that they were compiled from. How big is each one?

ii. The drawing of the unoptimized `Disagreements.fst` shows that it immediately branches at the
6 start state into two separate sub-FSAs. Why? (*Hint:* Look back at the regexp that defined `Disagreements`.)

iii. Now test some sample inputs with

```
grmtest binary-unopt.grm First
```

7 How are the results different from the optimized version? Why?

---

[7] After printing `fstinfo`, it calls `fstdraw` to produce a logical description of the drawing, then makes the drawing using the Graphviz package's `dot` command, and finally displays the drawing using `evince`. Each of these commands has many tweakable options. What if you're running on your own machine and don't have `evince`? Then edit the `fstview` script to use a different PDF viewer such as `xreader`, `atril`, `xpdf`, or `acroread`.

[8] If you start getting "can't open display" errors, then try connecting via `ssh -Y` instead of `ssh -X`. An alternative is to copy the (small) `.pdf` file to your local machine and use your local image viewer. Mac users might also like the free remote file browser Cyberduck.

(d) You may not want to call `Optimize` on every machine or regular sub-expression. The documentation offers the following warning:

> When using composition, it is often a good idea to call `Optimize[]` on the arguments; some compositions can be massively sped up via argument optimization. However, calling `Optimize[]` across the board (which one can do via the flag `--optimize_all_fsts`) often results in redundant work and can slow down compilation speeds on the whole. Judicious use of optimization is a bit of a black art.

8 If you optimize `Disagreements` *without* first optimizing `First` and `Second`, what do you get and why?

2. Now try some slightly harder examples. Extend your `binary.grm` to also export FSAs for the following languages. (You are welcome to define helper FSAs beyond these.)

   (a) `Triplets`: Binary strings where 1 only occurs in groups of three or more, such as `000000` and `0011100001110001111111`.

   (b) `NotPillars`: All binary strings *except* for even-length strings of 1's: ϵ, 11, 1111, 111111, 11111111, ... (These correspond to binary numbers of the form $2^{2k} - 1$ written in standard form.) Some strings that *are* in this language are `0`, 1, `000011`, 111, `0101`, `011110`.

   (c) `Oddlets`: Binary strings where 1's only appear in groups of odd length. Careful testing this one! (Note that `0000` is in this language, because 1's don't appear in it at all.)

You will extend `binary.grm` further in question 6.

3. So far we have only constructed FSAs. But Thrax also allows FSTs. Create a new file `rewrite.grm` in which you will define some FSTs for this question and the next question.

   Complicated FSTs can be built up from simpler ones by concatenation, union, composition, and so on. But where do you get some FSTs to start with? You need the built-in `:` operator:

```
input : output
```

   which gives the cross-product of the input and output languages.

   In addition, any FSA also behaves as the identity FST on its language.

   Place the following definition into `rewrite.grm`:

```
export Cross = "a" (("b":"x")* | ("c"+ : "y"*) | ("":"fric")) "a";
```

   Note that `""` denotes the empty string ϵ.

   9 (a) What is the input language of this relation (answer with an ordinary regexp)?

   10 (b) Give inputs that are mapped by `Cross` to 0 outputs, 1 output, 2 outputs, and more than 2 outputs.

   (c) How would you describe the `Cross` relation in English? (You do not have to hand in an answer for this, but at least think about it.)

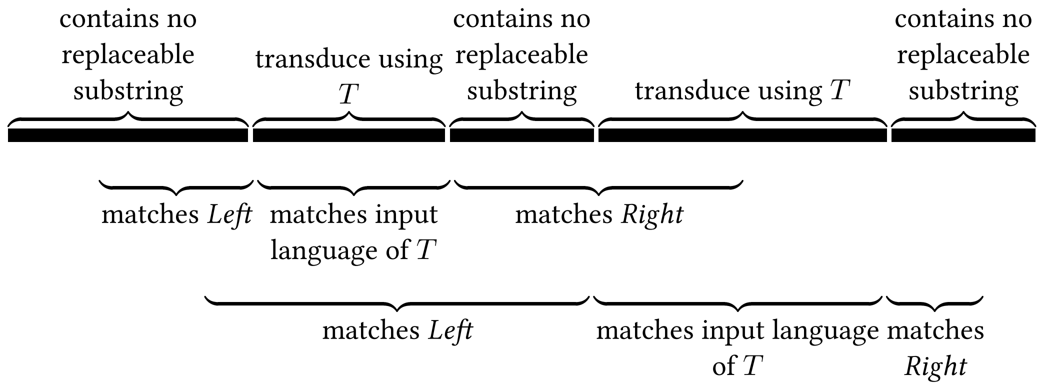
   11 (d) Make an `Optimized` version of `Cross` and look at it with `fstview`. Is it consistent with your answers above? How many states and arcs?

   Check your answers by transducing some inputs, using the following commands:

```
grmtest rewrite.grm Cross
grmtest rewrite.grm Cross 3
```

The second version of the command limits the number of outputs that are printed for each input that you enter.

4. If you have a simple FST, T, then you can make a more complicated one using **context-dependent rewriting**. Thrax's `CDRewrite` operator is similar to the `->` operator in XFST. It applies T "everywhere it can" within the input string, until there are no unreplaced substrings that could have been replaced. The following shows schematically how two substrings of ▬▬▬ might be replaced:



The braces underneath the string show that each of the 2 replaced substrings appears in an appropriate context—immediately between some substring that matches *Left* and some substring that matches *Right*. The 2 replaced substrings are not allowed to overlap, but the contexts can overlap with the replaced substrings and with one another, as shown in the picture.[9]

If you want to require that the *maximal* substring to the left matches *Left*, then start *Left* with the special symbol `[BOS]`, which can only match at the beginning of the string. Similarly for *Right* and `[EOS]` (end of string).

The above example shows only one way of dividing up the input string into regions that are transduced by T and regions that are left alone. If there are other ways of dividing up this input string, then the rewrite transducer will try them too—so it will map this input to multiple outputs.[10]

`CDRewrite[`*`T`*`, `*`Left`*`, `*`Right`*`, `*`Any`*`, `*`Dir`*`, `*`Force`*`]` specifies a rewrite transducer with arguments

- *`T`* : any FST
- *`Left`*, *`Right`* : unweighted FSAs describing the left and right contexts in which T should be applied. They may contain the special symbols `"[BOS]"` and `"[EOS]"`, respectively. (These symbols are only to be used when describing contexts, as in these arguments to `CDRewrite`, which interprets them specially. They do *not* appear in the symbol table.)
- *`Any`* : a minimized FSA for Σ^*, where Σ is the alphabet of input and output characters. The FST produced by CDRewrite will only allow input or output strings that are in *`Any`*, so be careful!

[9] If you are wondering how this is accomplished, have a look at Mohri & Sproat (1996), section 3.1.

[10] So there is no notion here of selecting the regions to transduce in some deterministic way, such as the left-to-right longest match used by the XFST `@->` operator.

- *`Dir`* : the direction of replacement.
  - `'sim'` specifies "simultaneous transduction": *`Left`* and *`Right`* are matched against the original input string. So all the substrings to replace are identified first, and then they are all transduced in parallel.
  - `'ltr'` says to perform replacements "in left-to-right order." A substring should be replaced if *`Left`* matches its left context *after* any replacements to the left have been done, and *`Right`* matches its right context *before* any replacements to the right have been done.
  - `'rtl'` uses "right-to-left" order, the other way around.
- *`Force`* : how aggressive to be in replacement?
  - `'obl'` ("obligatory," like `->` in XFST) says that the *unreplaced* regions may not contain any more replaceable substrings, as illustrated above. That is, they may not contain a substring that matches the input language of *`T`* and which falls between substrings that match *`Left`* and *`Right`*.
  - `'opt'` ("optional," like `(->)` in XFST) says it's okay to leave replaceable substrings unreplaced. Since the rewrite transducer has the freedom to replace them or not, it typically has even more outputs per input.

Define the following FSTs in your `rewrite.grm`, and test them out with `grmtest`:

(a) `BitFlip1`: Given a string of bits, changes every `1` to `0` and every `0` to `1`. This is called the "1's complement" of the input string. Define this *without* using `CDRewrite`.

(b) `BitFlip2`: Like `BitFlip1`, but now it should work on any string of digits (e.g., transducing `1123401` to `0023410`). Define this version using `CDRewrite`.

Hint: The *`Any`* argument in this case should be `Digit*` where

```
Bit = "0" | "1";
Digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9";
```

(c) `Parity1`: Transduces even binary numbers to `0` and odd binary numbers to `1`. Write this one *without* using `CDRewrite`.

It's always good to think through the unusual cases. Some people think the empty string ϵ is a valid
12 binary number (representing 0), while others don't. What does your transducer think?

(d) `Parity2`: Same thing as `Parity1`, but use the `Reverse` function in your definition. So start by writing a tranducer that keeps the *first* bit of its input instead of the *last* bit.

(e) `Parity3`: Same thing as `Parity1`, but this use `CDRewrite` in your definition.

13 What does this transducer think about ϵ?

Hint: You may find it helpful to use `[BOS]` and `[EOS]`, or the composition operator `@`.

14 (f) `UnParity`: Define this as `Invert[Parity3]`. What does it do?

(g) `Split`: Split up a binary string by nondeterministically inserting spaces into the middle, so that input `0011` maps to the eight outputs

{`0011`, `001 1`, `00 11`, `00 1 1` , `0 011`, `0 01 1`, `0 0 11`, `0 0 1 1`}

Hint: Use `CDRewrite["":" ",...]` and figure out the other arguments.

✌15 (h) **Extra credit:** `SplitThree`: Similar to `Split`, but always splits the input string into exactly three (nonempty) binary strings. This will produce multiple outputs for strings longer than 3 bits, and no outputs for strings shorter than 3 bits.

Hint: Compose `Split` with something else. The composition operator is `@`.

5. Bit strings are great, but let's move to natural language. You know from HW1 that precisely describing syntax can be challenging, and from HW4 that recovering the full parse of a sentence can be slow. So, what if you just want a quick finite-state method for finding simple NPs in a sentence? This could be useful for indexing text for search engines, or as a preprocessing step that speeds up subsequent parsing. Or it could be part of a cascade of FSTs that do information extraction (e.g., if you want to know who manufactures what in this world, you could scan the web for phrases like "X manufactures Y" where X and Y are NPs).

Identifying interesting substrings of a sentence is called "chunking." It is simpler than parsing because the "chunks" are not nested recursively. This tutorial question will lead you through building an FST that does simple NP chunking.

We will assume that the input sentence has already been tagged (perhaps by a tagging FST, which you could compose with this chunking FST). You'll build the following objects:

- An FSA that accepts simple noun phrases: an optional determiner, followed by zero or more adjectives `Adj`, followed by one or more nouns `Noun`. This will match a "base" NP such as `the ghastly orange tie`, or `Mexico`—though not the *recursive* NP `the ghastly orange tie from Mexico that I never wear`.

  The regexp defining this FSA is a kind of simple grammar. To make things slightly more interesting, we suppose that the input has two kinds of determiners: quantifiers (e.g., `every`) are tagged with `Quant` whereas articles (e.g., `the`) are tagged with `Art`
- A transducer that matches exactly the same input as the previous regular expression, and outputs a *transformed* version where non-final `Noun` tags are replaced by `Nmod` ("nominal modifier") tags. For example, it would map the input `Adj Noun Noun Noun` deterministically to `Adj Nmod Nmod Noun` (as in `delicious peanut butter filling`). It would map the input `Adj` to no outputs at all, since that input is not a noun phrase and therefore does not allow even one accepting path.
- A transducer that reads an arbitrary input string and outputs a single version where all the *maximal* noun phrases (chosen greedily from left to right) have been transformed as above and bracketed.

(a) You'll be editing the provided file `chunker.grm`:

```
import 'byte.grm' as bytelib;
import 'tags.grm' as tags;
Sigma = (tags.Tags) | (bytelib.kBytes);
SigmaStar = Optimize[Sigma*];
```

Copy this file along with `byte.grm` and `tags.grm` from the `grammars/` directory. Line 2 defines `tags` to be the collection of FSMs that are `export`ed by `tags.grm`. Expressions like `tags.Tags` in line 3 then refer to individual FSMs in that collection. You should look at these other files referenced by lines 1–2. Now:

i. Define an FSA `NP` that accepts an optional article (`Art`) or quantifier (`Quant`); followed by an arbitrary number of adjectives (`Adj`); followed by at least one noun (`Noun`). We would like to write:

```
export NP = Optimize[(Art|Quant)? Adj* Noun+];
```

16 What goes wrong? (*Hint:* look at importable FSMs from `tags`.) Fix the definition in `chunker.grm`,
17 and in your `README`, provide evidence of what you were able to accept.
You will use the fixed `chunker.grm` for the rest of this question.
(*Note*: Really we should be working over the alphabet of tags rather than the default alphabet of ASCII characters. Later in the assignment we'll see how to define other alphabets using *symbol tables*.)

ii. Have a look at `NP`:

```
far2fst chunker.far NP
fstview NP.fst
```

18 How many states and arcs are there? Comment on the structure of the machine.

(b) In a noun-noun compound, such as `the old meat packing district`, the nouns `meat` and `packing` act as *nominal modifiers*. Define and try out a transducer `MakeNmod`, using `CDRewrite`, that replaces `Noun` with `Nmod` immediately before any `Noun`. So `ArtAdjNounNounNoun` as in the example becomes `ArtAdjNmodNmodNoun`.

To define `MakeNmod`, you'll need to figure out what arguments to use to `CDRewrite`.

(c) Now define an FST

```
export TransformNP = Optimize[NP @ MakeNmod];
```

19 i. Describe in words what this composition is doing.
20 ii. What are the results on `ArtAdjNounNounNoun` and `AdjNounNounNounVerb`?
21 iii. What is the size of `TransformNP` compared to `MakeNmod`?
22 iv. How does the topology of `TransformNP` differ from that of `NP`?

(d) This FST transduces a noun phrase to one that has <angle brackets> around it:

```
export BracketNP = ("" : "<") NP ("" : ">");
```

Here the `NP` language is being interpreted as the identity relation on that language, and concatenated with two other simple regular relations. So `BracketNP` reads ϵ, any `NP`, ϵ and writes <, the same `NP`, >.

23 What, if anything, is the difference between the following?

```
export Brackets1 = Optimize[SigmaStar (BracketNP SigmaStar)*];
export Brackets2 = CDRewrite[BracketNP, "", "", SigmaStar,'sim','obl'];
```

Try them out on short and long strings, such as `ArtAdj`, `AdjNoun`, and `VerbArtAdjNounNounNounVerbPrepNoun`.

(e) Now define `BracketTransform` to be like `Brackets2`, except that it should not only bracket noun phrases but also apply the transformation defined by `TransformNP` within each noun phrase. This should be a fairly simple change.

(f) One interesting thing about FSTs is that you can pass many strings through an FST at once. Define `BracketResults` to be the regular language that you get by applying `BracketTransform` to *all* strings of the form `Quant Noun+ Verb` at once.

(*Hint:* Check out the `Project` operator in the Thrax documentation.[11] You may want to optimize the result.

[11] This operator is used to "project" a relation onto the upper or lower language, like the `.u` and `.l` operators in XFST. Why is that called projection? Consider a set of points on the plane: $\{(x_1, y_1), (x_2, y_2), \ldots\}$. The projection of this set onto the x axis is $\{x_1, x_2 \ldots\}$ and the projection onto the y axis is $\{y_1, y_2 \ldots\}$. Same thing when projecting a regular relation, except that each (x_i, y_i) pair is a pair of strings.

You can check the FSA by using `fstview` on `BracketResults` (note that it may be drawn as an identity FST). To print out the strings it accepts (up to a limit), run `grmtest` on the cross-product machine `"":BracketResults`, and enter an empty input string to see all the outputs.

(g) **Extra credit:** To get a sense of how `CDRewrite` might do its work, define your own version of `TransformNP` that does *not* use `CDRewrite`. It should be a composition of three FSTs:

- *Optionally* replace each `Noun` with `Nmod`, without using `CDRewrite`. This will transduce a single input to many outputs.
- *Check* that no `Noun` is followed by another `Noun` or `Nmod`. This filters outputs where you didn't replace enough nouns.
- *Check* that every `Nmod` is followed by a `Noun` or another `Nmod`. This filters outputs where you replaced too many nouns. (*Hint:* It is similar to the XFST "restrict" operator that we defined in class.)

24 Call your version `TransformNP2`.

6. In OpenFST, you can define weighted FSMs. By default, OpenFST, NGram and Thrax all use the tropical semiring, $\langle \mathbb{R} \cup \{\pm\infty\}, \oplus = \min, \otimes = +\rangle$.[12] Thus, weights can be interpreted as costs. Concatenating two paths or regexps will *sum* their costs, whereas if a string is accepted along two parallel paths or by a union of regexps, then it gets the *minimum* of their cost.

   Augment an FSM's definition by appending a weight w in angle brackets, < and >, and wrapping the entire definition in parentheses.

   25 (a) i. What is the minimum-weight string accepted by this FSA, and what is the weight of that string? (Remember that parentheses in Thrax just represent grouping, not optionality.)

   ```
   (Zero <1>) (Bit+ <0.2>) (One <0.5>)
   ```

   26 ii. What is the minimum-weight pair of strings accepted by this FST, and what is the weight of that pair?

   ```
   (Zero : One <1>) (Bit+ <0.2>) (One : One One <0.75>)
   ```

   (b) In your old `binary.grm` file, define a weighted transducer `WFlip` that accepts the language of the above FSA and, reading left to right:

   - Nondeterministically flips the leftmost bit. Flipping has weight 2, while not flipping has weight 1.
   - In the `Bit+` portion, replaces every `0` with `01` (at cost 0.5), and replaces every `1` with `0` (at cost 0.4).
   - Accepts the final `1` bit with weight 0.5.

   Don't use `CDRewrite` here.

   For example, `WFlip` should produce the following:

   ```
   Input string: 0011
   Output string: 00101 <cost=2.4>
   Output string: 10101 <cost=3.4>
   ```

   (c) Now let's consider cases where we aggregate the weights of multiple paths using $\oplus$.

   27 i. In your `README`, name any two binary strings x, y: for example, $(x, y) = (\texttt{00}, \texttt{1})$. In `binary.grm`, define `WeightedMultipath` to be a simple *weighted* FST of your choice, such that the particular pair (x, y) that you named will be accepted along at least two different paths, of different weights. To confirm that these two accepting paths exist, view a drawing of the machine, and use `grmtest`
   28 to find out what x maps to. What are the weights of these paths?

[12] And currently a portion of the Thrax we're using supports only this semiring.

29 ii. Now define `WeightedMultipathOpt = Optimize[WeightedMultipath]`. In this FST, how many paths accept (x, y) and what are their weights? Why?

iii. Suppose `T` is an FST with weights in some semiring, and `x` and `y` are strings. So `T` accepts the pair (`x`, `y`) along 0 or more weighted paths.

30 Describe, in English, what the following weighted languages or relations tell you about `T`:

```
T_out       = Project[     T,      'output'];  # erases input from T
xT_out      = Project[ x @ T,      'output'];  # erases input x from x @ T
Ty_in       = Project[     T @ y, 'input'];    # erases output y from T @ y
xTy         =        x @ T @ y;
exTye       = ("":x) @ T @ (y:"");  # erases input x & output y from x @ T @ y

xT_out_opt = Optimize[xT_out];
Ty_in_opt  = Optimize[Ty_in];
exTye_opt  = Optimize[exTye];
```

31 How big is the last FSM, in general? Why do the last three FSMs have practical importance?[13]

32 You can try these all out for the case where `T` is `WeightedMultipath` and `x` and `y` denote the strings (x, y) you named above.

33 (d) **Extra credit:** Define an FSM `NoDet` that has no deterministic equivalent. (Unweighted FSAs can always be determinized, but it turns out that either outputs (FSTs) or weights can make determinization impossible in some cases that have cycles.)

How will you know you've succeeded? Because the Thrax compiler will run forever on the line `Determinize[RmEps` the determinization step will be building up an infinitely large machine. (`RmEpsilon` eliminates ϵ arcs from the FSM, which is the first step of determinization. Then `Determinize` handles the rest of the construction.)

7. Throughout the remainder of this assignment, we'll be focused on building noisy-channel decoders, where weighted FSTs really shine. Your observed data $\mathbf{y}$ is assumed to be a distorted version of the "true" data $\mathbf{x}$. We would like to "invert" the distortion as best we can, using Bayes' Theorem. The most likely value of $\mathbf{x}$ is

$$\mathbf{x}^* \stackrel{\text{def}}{=} \operatorname*{argmax}_{\mathbf{x}} \Pr(\mathbf{x} \mid \mathbf{y}) = \operatorname*{argmax}_{\mathbf{x}} \Pr(\mathbf{x}, \mathbf{y}) = \operatorname*{argmax}_{\mathbf{x}} \underbrace{\Pr(\mathbf{x})}_{\text{"language model"}} \underbrace{\Pr(\mathbf{y} \mid \mathbf{x})}_{\text{"channel model"}} \tag{1}$$

In finite-state land, both language and channel models should be easy to represent. A channel is a weighted FST that can be defined with Thrax, while a language model is a weighted FSA that can be straightforwardly built with the NGram toolkit.[14] To make even easier to build a language model, we've given you a wrapper script, `make-lm`:

```
make-lm corpus.txt
```

[13] In general, one might want to do one of these computations for many different `x` (or `y`) values. Rather than compiling a new Thrax file for each `x` value, you could use other means to create `x` at runtime and combine it with `T`. For example, to work with `BracketTransform` from question 5e, try typing this pipeline at the command line: `echo "ArtNounNounNoun" | fstcompilestring | fstcompose - BracketTransform.fst | fstproject --project_output | fstoptimize | fstview`. (Other handy utilities discussed in this handout are `fstrandgen` for getting random paths, `fstshortestpath` for getting the lowest-cost paths, `farprintstrings` for printing strings from these paths, and our `grmfilter` script for transducing a file.) Or you could just use the C++ API to OpenFST.

[14] Language models can't be concisely described with regular expressions: at least, not the standard ones.

By default, this will create a Kneser–Ney back-off trigram language model called `corpus.fst`. Every sentence of `corpus.txt` should be on its own line.

(a) Create the default language model for the provided file `data/entrain.txt`. Each line of this file represents an observed sentence from the English training data from the HMM assignment.[15] Notice that the sentences have already been tokenized for you (this could be done by another FST, of course).

You should now have a language model `entrain.fst` in your working directory, along with two other files you'll need later: `entrain.alpha` is the alphabet of word types and `entrain.sym` is a symbol table that assigns internal numbers to them.

Look at the files, and try this:

```
wc -w entrain.txt     # number of training tokens
wc -l entrain.alpha   # number of training types
fstinfo entrain.fst   # size of the FSA language model
```

(b) The NGram package contains a number of useful shell commands for using FST-based n-gram models. Three that you may find particularly interesting are `ngramprint`, `ngramrandgen`, and `ngramperplexity`. You can read up on all three if you like (use the `--help` flag).

Sampling several sentences from the language model is easy:

```
ngramrandgen --max_sents=5 entrain.fst | farprintstrings
```

Each `<epsilon>` represents a backoff decision in the FSM. You can see that there is a *lot* of backoff from 2 to 1 to 0 words of context. That's because this corpus is very small by NLP standards: the smoothing method realizes that very few of the possible words in each context have been seen yet.

To read the sentence more easily, use the flag `--remove_epsilon` to `ngramrandgen` (this prevents `<epsilon>`s from being printed). Alternatively, just use our `fstprintstring` script to print a random output from any FST:

```
fstprintstring entrain.fst
```

If you want to know the most probable path in the language model, you can use `fstshortestpath`, which runs the Viterbi algorithm.[16]

```
fstshortestpath entrain.fst | fstprintstring
```

☞ 34 How long are the strings in both cases, and why? What do you notice about backoff?

(c) **Recommended:** Although it's a small language model, `entrain.fst` is far too large to view in its entirety. To see what a language model FSA looks like, try making a *tiny* corpus, `tiny-corpus.txt`: just 2–3 sentences of perhaps 5 words each. Try to reuse some words across sentences. Build a language model `tiny-corpus.fst` and then look at it with `fstview`. There is nothing to hand in for this question.

(d) Now, let's actually use the `entrain.fst` language model. Copy `noisy.grm` from the `grammars/` directory:

```
import 'byte.grm' as bytelib;         # load a simple grammar (.grm)
export LM = LoadFst['entrain.fst'];   # load trigram language model (.fst)
vocab = SymbolTable['entrain.sym'];   # load model's symbol table (.sym)
```

[15] Because the NGram toolkit assumes that sentence boundaries are marked by newlines, we've omitted `###`.

[16] In general `fstprintstring` will print a randomly chosen string, but in the output of `fstshortestpath`, there is only one string to choose from.

Each line loads a different kind of external resource. In particular, the second line loads the trigram language model FSA, and the third line loads the symbol table used by that FSA. The symbol table consists of the vocabulary of the language model, as well as the OOV symbol `<unk>` ("unknown").[17]

You can therefore use `LM` to transduce some strings:

```
grmtest-with-symbols noisy.grm LM entrain.sym entrain.sym
```

35 What is the result of transducing the following? Explain your answers. What are the domain and range of the relation `LM`?

- `Andy cherished the barrels each house made .`
- `If only the reporters had been nice .`
- `Thank you`

We now want to compose `LM` with a noisy channel FST. Because the language model is nothing more than an FSA, we can use it in Thrax. Of course, we're going to have to be careful about symbol tables: the noisy channel's input alphabet must be the same as `LM`'s output alphabet.

Remember to be careful when creating FSTs over a nonstandard alphabet. If you write

```
("barrels barrels" : ("" | "ship"))*;
```

then the input to this FST must be a multiple of 15 symbols in the default `byte` alphabet. But if you write

```
("barrels barrels".vocab : ("".vocab | "ship".vocab))*;
```

then Thrax will parse the quoted strings using the `vocab` symbol table. So here, the input must be an even number of symbols in the `vocab` alphabet. Writing `"Thank".vocab` will give an error because that word is not in the symbol table (it's not in the file `entrain.sym` from which `vocab` was loaded).

A noisy channel that "noises up" some text might modify the sequence of words (over the `vocab` alphabet) or the sequence of letters (over the `byte` alphabet). If you want to do the latter, you'll need to convert words to letters. Recall from question **??** that the `StringFile` function interprets its given tab-separated text file as an FST, with the domain and range as the second and third parameters, respectively. So we'll add the following line to `noisy.grm`:

```
Spell = Optimize[StringFile['entrain.alpha', vocab, byte]];
```

This maps a word to its spelling, just as `Pronounce` in **??** mapped a word to its pronunciation.

(e) In `noisy.grm`, define a transducer called `CompleteWord` that could be used to help people enter text more quickly. The input should be the first few characters in a word, such as `barr`. Each output should be a word that starts with those characters, such as `barrel` or `barrage`.

36 Use `LM` to assign a cost to each word, so that each completed word is printed together with its cost. Is a word's cost equal to the unigram probability of the word, or something else?

Hint: Be careful to think about the input and output alphabets, and to pass them as arguments to `grmtest-with-symbols`. The input alphabet should be `byte` (not `byte.sym`), as explained in question **??**.

37 (f) **Extra credit:** Now define `CompleteWordInContext` similarly. Here the input is a sequence—separated by spaces—of 0 or more complete words followed by a partial word. Each output is a single word that completes the partial word, as before. But this time the cost depends on the context: that's what language models are for.

38 Try it out and give a few example inputs that illustrate how the context can affect the ranking of the different completions.

Hint: You might not able to get away with exporting `CompleteWordInContext` as a single transducer—it's rather big because it spells out the words in the language model. It will be more efficient to use the pipelining trick from question ??. In your `README`, tell the graders what command to enter in order to try out your pipeline, and give the original `CompleteWordInContext` definition that your pipeline was derived from.

8. Question 7 defined our language model, $\Pr(\mathbf{x})$. Now let's compose it with some channel models $\Pr(\mathbf{y} \mid \mathbf{x})$ that we'll define. In this question, we'll practice by working through a simple deterministic noisy channel.

   (a) Still working in `noisy.grm`, define a deterministic transducer `DelSpaces` that deletes all spaces. Define this using `CDRewrite`, and use the alphabet `bytelib.kGraph | bytelib.kSpace`. Using `grmtest` you should be able to replicate the following:

```
Input string: If only the reporter had been nice .
Output string: Ifonlythereporterhadbeennice.
Input string: Thank you .
Output string: Thankyou.
Input string: The reporter said to the city that everyone is killed .
Output string: Thereportersaidtothecitythateveryoneiskilled.
```

   (b) `grmtest` will transduce each string that you type in, providing multiple outputs when they exist. To transduce a whole file to a single output, once you've tested your transducer, we've provided another wrapper script `grmfilter`:

```
$ grmfilter
Usage:
        cat input.txt | grmfilter [-opts] <grammar file> <name1>,<name2>,...
-r: select a random path
-s: find shortest path (default)
-h: print this help message (and exit)
```

   Just like `grmtest`, it takes two required arguments, a `.grm` file and a comma-separated list of FST names defined in that file. It reads strings from the standard input, one per line, and writes their transductions to the standard output. The output string comes from *one* of the paths that accept the input. The default (which can be signaled explicitly with the `-s` flag) is to choose a maximum-probability path. The alternative (the `-r` flag) is to select a path randomly in proportion to its probability. We know each path's probability because its total cost gives the negative log of its probability.

[17] The `make-lm` script takes the vocabulary to be all the words that appear in training data, except for a random subset of the singletons. These singletons are removed from the vocabulary to ensure that some training words will be OOV, allowing the smoother to estimate the probability of OOV in different contexts. Ideally the smoothing method would figure this out on its own.

Try running `DelSpaces` on the text file `data/entest.txt`, which contains the first 50 sentences of the English test data `entest` from the HMM assignment. Save the result as `entest-noisy.txt`. In general, you should use the `-r` flag to pass text through a noisy channel, so that it will randomly noise up the output (although in this introductory case the channel happens to be deterministic):

```
grmfilter -r noisy.grm DelSpaces < entest.txt > entest-noisy.txt
```

Uh-oh! Someone got into your files and used your own `DelSpaces` against you! Now how will you ever read any of your files?

After despairing for a while, you realize that you can just reverse `DelSpaces`'s actions. So you try `Invert[DelSpaces]`, but unfortunately that turns `Ifonlythereporterhadbeennice.` back into all kinds of things like

```
I fon lyt    he reporterh adbeenni  ce.
```

The correct solution is somewhere in that list of outputs, but you need to find it. What a perfect opportunity to use your language model `LM` and the Viterbi algorithm for finding the most probable path!

The idea is that the text actually came from the generative process (1), which can be represented as the composition

```
Generate = LM @ DelSpaces;   # almost right!
```

Unfortunately the output of `LM` is words, but the input to `DelSpaces` is characters. So they won't compose. You will need to stick a transducer `SpellText` in between. This transducer represents another deterministic step in the generative process that resulted in the noisy sequence of characters.

(c) Define `SpellText` in `noisy.grm`. It should spell the first input word, output a space, spell the second input word, output another space, and so on. This yields the kind of text that actually appeared in `entrain.txt` (there is a space after each word in a sentence, including the last).

Now revise your definition of `Generate` to use `SpellText`.

(d) Now you should be able to decode noisy text via

```
Decode = Invert[Generate];
```

Unfortunately, this machine will be too large (and slow to compile). So you should use the same approach as in question ??, and ask `grmtest` to pass the noisy text through a sequence of inverted machines.

Important: At the end of your sequence of machines, you should add `PrintText`, which you can define for now to be equal to `SpellText`. This has the effect of pretty-printing the decoded result. It will turn the recovered sequence of words back into characters, and put spaces between the words.

Using `grmtest` in this way, try decoding each of the following. Note that the lowest-cost results are
39 shown first. Discuss the pattern of results, and their costs, in your `README`:

- `Ifonlythereporterhadbeennice.`
- `If only.`
- `ThereportersaidtothecitythatEveryoneIskilled.`
- `Thankyou.`

(e) The reason Thankyou failed is because we didn't account for OOVs. The vocabulary has an OOV symbol <unk>, but it is treated like any other word in the vocabulary.[18] So `LM` will accept phrases like <unk> you, but not `Thank you`.

So just as we described how to spell in the above questions, we'll now describe how to spell `OOV` words. We'll say that <unk> can rewrite as an arbitrarily long sequence of non-space text characters (`bytelib.kGraph`):

```
RandomChar = bytelib.kGraph <4.54>;
RandomWord = Optimize[(RandomChar (RandomChar <w1>)* ) <w2>];
SpellOOV = "<unk>".vocab :  RandomWord;
```

The weight in `RandomChar` is saying that each of the 94 characters in `bytelib.kGraph` has the same probability, namely $\frac{1}{94}$, since $-\log\frac{1}{94} \approx 4.54$.

How about `RandomWord`? When you define it in `noisy.grm`, you'll have to give actual numbers for the numeric weights w_1 and w_2. Try setting $w_1 = 0.1$ and $w_2 = 2.3$. To check out the results, try these commands:

```
grmtest noisy.grm RandomWord   # evaluate cost of some strings
far2fst noisy.far RandomWord   # (get the FSA for commands below)
fstprintstring RandomWord.fst  # generate a random string
fstview RandomWord.fst         # look at the FSA
```

40 i. What do w_1 and w_2 represent? Hint: the costs 0.1 and 2.3 are the negative logs of 0.9 and 0.1.

41 ii. For each $n \geq 0$, what is the probability p_n that the string generated by `RandomWord` will have length n?

42 iii. What is the sum of those probabilities, $\sum_{n=0}^{\infty} p_n$?

43 iv. How would you change w_1 and w_2 to get longer random words on average?

44 v. If you decreased *both* w_1 and w_2, then what would happen to the probabilities of the random words? How would this affect the behavior of your decoder? Why?

45 vi. How could you improve the probability models `RandomChar` and `RandomWord`?

Once you've answered those questions, **reset $\mathbf{w_1 = 0.1}$ and $\mathbf{w_2 = 2.3}$ and proceed.**

(f) Now, revise `Spell` so that it is not limited to spelling words in the dictionary, but can also randomly spell <unk>. (*Hint:* Use `SpellOOV`.)

Also revise `PrintText` so that if your decoder finds an unknown word <unk>, you will be able to print that as the 5-character string "<unk>."

To check your updated decoder, try running the sentences from question 8d through it. Again discuss the pattern of results. Remember that if you want, you can add an extra argument to `grmtest` to limit the number of outputs printed per input.

(g) Remember that your goal was to de-noise your corrupted files, whose spaces were removed by `DelSpaces`. Just run `grmfilter` again, but with three differences:

- Before, you were converting `entest.txt` to `entest-noisy.txt`. Now you should convert `entest-noisy.txt` to `entest-recovered.txt`.
- Instead of running the noisy channel forward, run it backward, using your pipeline from 8d. You can leave out the `PrintText` step of the pipeline since `grmfilter` is a bit smarter than `grmtest` about how it prints outputs.

[18]Except by some of the `ngram` utilities that we're not using.

- Since you want the most likely decoding and not a random decoding, don't use the `-r` flag this time.

46 Look at the results in `entest-recovered.txt`. What kinds of errors can you spot? Does this *qualitative* error analysis give you any ideas how to improve your decoder?

(h) Suppose you'd like to *quantify* your performance. The metric we'll consider is the **edit distance** between `entest.txt` and `entest-recovered.txt`.

Edit distance counts the minimum number of edits needed to transduce one string (x) into another (y). The possible edits are

- **substitute** one letter for another;
- **insert** a letter;
- **delete** a letter;
- **copy** a letter unchanged.

Each of these operations has a cost associated with it. We'll stick with the standard unweighted edit distance metric in which substitions, insertions and deletions all have cost 1; copying a character unchanged has cost 0. For simplicity we will treat the unknown word symbol as if really were the 5-character word `<unk>`, which must be edited into the true word.

As you know, edit distance can easily be calculated using weighted finite-state machines:

```
Sigma = bytelib.kBytes;
export Edit = (Sigma | ((""|Sigma) : (""|Sigma)  <1>) )*;
```

The `Edit` machine transduces an input string x one byte at a time: at each step, it either passes an input character through with cost 0, or does an insert, delete or substitute with cost 1. That gives an edited version y. The cheapest way to get from x to a given y corresponds to the shortest path through x `@ Edit @` y. As we saw in class, that machine has the form of an $|x+1| \times |y+1|$ grid with horizontal, vertical, and diagonal transitions. It has exponentially many paths, of various total cost, that represent different sequences of edit operations for turning x into y.

We've given you an edit distance script to calculate the edit distances between the corresponding lines of two files:

```
editdist entest.txt entest-recovered.txt
```

This will compare each recovered sentence to the original. Do the scores match your intuitive, qualitative results from 8g?

Please look at `grammars/editdist.grm`, the Thrax grammar used by the `editdist` script. You'll see that it's more complicated than `Edit`, but this construction *reduces* the size of the overall machine by a couple of orders of magnitude. While it still computes x `Edit` y, it splits `Edit` up into two separate machines, `Edit1` and `Edit2`. We still find the shortest path, but now through

(x `@ Edit1) @ (Edit2 @` y`);`

By doing the composition this way, both x and y are able to impose their own constraints (what letters actually appear) on `Edit1` and `Edit2`, thus reducing the size of the intermediate machines. The resulting FST can be built quite quickly, though as mentioned before, it does have $|x+1| \times |y+1|$ states and a similar number of arcs.

✌47 (i) **Extra credit:** How can you modify your pipeline so that it recovers an appropriate spelling of each unknown word, rather than <unk>? For example, decoding `Thankyou` should give `Thank you` rather than <unk> `you`.[19]

✌48 9. **Extra credit (but maybe the real point of this assignment):** Finally, it's time to have some fun. We just set up a noisy-channel decoder to handle a simple deterministic noisy channel. Now try it for some other noisy channels! The framework is nearly identical—just replace `DelSpaces` with some other FST. For each type of noisy channel,

   i. Define your channel in `noisy.grm` as a weighted FST.
   ii. Explain in `README` what you implemented and why, and how you chose the weights.
   iii. Use your channel to corrupt `entest.txt` into `entest-noisy.txt`.
   iv. Use your inverted channel, the language model, and `SpellText` to decode `entest-noisy.text` back into `entest-recovered.txt`.
   v. Look at the files and describe in your `README` what happened, with some examples.
   vi. Report the edit distance between `entest.txt` and `entest-recovered.txt`.

   Have fun designing some of the channels below. Each converts a string of bytes into a string of bytes. In general make them non-deterministic (in contrast to `DelSpaces`), and play with the weights.

   (a) `DelSomeSpaces`: Nondeterministically delete none, some, or all spaces from an input string.
   (b) `DelSuffixes`: Delete various word endings. You may find `http://grammar.about.com/od/words/a/comsuffixes.htm` helpful.
   (c) `Typos`: Introduce common typos or misspellings. You may get some inspiration from `http://en.wikipedia.org/wiki/Wikipedia:Lists_of_common_misspellings` or `http://en.wikipedia.org/wiki/File:Qwerty.svg`
   Some real-world typos are due to the fact that some words *sound* similar, so if you're ambitious, you might be able to make some use of `data/cmudict.txt` or the `Pronounce` transducer.
   (d) `Telephone`: Deterministically convert (lower-case) letters to digits from the phone keypad. For example, `a` rewrites as `2`.
   (e) `Tinyphone`: Compose your `Telephone` FST with another FST that allows common cellphone keypad typos. For example, there should be a small chance of deleting a digit, doubling a digit, or substituting one of the adjacent digits for it.
   (f) Try composing some of these machines in various orders. As usual, give examples of what happens, and discuss the interactions.

   Feel free to try additional noisy channels for more extra credit. You could consider capitalization, punctuation, or something crazy out of your imagination.[20]

---

[19] The recovered spelling is determined by the language model and the channel model. It won't always match the noisy spelling. E.g., if the noisy channel tends to change letters into lowercase, then decoding `Thank you` might yield `THANK you`.

[20] It might be fun to replace each word deterministically with its rhyming ending, using your `WordEnding` FST from question ?? (composed with something that transduces ARPAbet characters to the byte alphabet). Then your noisy channel decoder will find the highest-probability string that rhymes word-by-word with your original input text. Should be amusing.