This short assignment exposes you to finite-state hacking. You will build finite-state
transducers by hand, using the extended regular expression language available in the Xerox
Finite-State Tool (XFST). XFST does not support probabilities or other weights, but it
supports both acceptors (FSAs) and transducers (FSTs).

1. First, get to know XFST. Here is a tutorial that walks you through an example.\footnote{It is a slightly more straightforward and self-contained version of the tutorial at \url{http://cs.jhu.edu/~jason/405/software.html#xfst}.} You
only have to hand in answers to 1k and 1n.

The tutorial shows you how to build the following objects:

- A regular expression over an alphabet of part-of-speech tags. The regexp is
intended to accept simple noun phrases: an optional determiner, followed by
zero or more adjectives \texttt{Adj}, followed by one or more nouns \texttt{Noun}.
To make things slightly more interesting, determiners fall into two types, quan-
tifiers (“every”) and articles (“the”). These are assumed to have different tags \texttt{Quant} and \texttt{Art}.
- A transducer that matches exactly the same input as the previous regular expres-
sion, and outputs a \textit{transformed} version where non-final \texttt{Noun} tags are replaced by
\texttt{Nmod} (“nominal modifier”) tags. For example, it would map the input \texttt{Adj Noun Noun Noun} deterministically to \texttt{Adj Nmod Nmod Noun} (as in “delicious peanut butter filling”). It would map the input \texttt{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 \textit{maximal} noun phrases (chosen greedily from left to right) have
been bracketed and transformed as above.

(a) \texttt{xfst} is installed on the \texttt{ugrad} machines (also on the \texttt{masters} machines and on
the Solaris machine \texttt{peregrine}).\footnote{Also installed are the other tools that come with \texttt{xfst}, namely \texttt{lexc}, \texttt{lookup}, \texttt{tokenize}, and \texttt{twolc}. These are described in the FSM book.} If you would prefer to install a copy on your
own machine (for non-commercial use), visit \url{www.fsmbook.com} and follow the
link in the first paragraph. Alternatively, try Foma, a new open-source clone of
XFST that uses OpenFST as a back end (thus, semiring weights may be added in a future version): http://foma.googlecode.com.

(b) To start XFST, type xfst. This gives you a command line. Useful commands are help, help command, and apropos topic. There are many commands but you can make do with only a few of them. Because XFST doesn’t have good command-line editing and recall facilities, you may want to start a shell in Emacs and run XFST in that shell. (ESC x shell starts the shell and C-h m tells you how it works.)

(c) Define a regular expression: define Nounphrase (Art|Quant) Adj* Noun+ ;

Note: Art, Quant, Adj and Noun are single symbols here, from an alphabet of part-of-speech tags. A sequence of symbols such as Quant Noun Noun is called a “word.”

Warning: Remember that parentheses () mean “optional”; XFST uses brackets [ ] for ordinary grouping. Regular expressions must be terminated by semicolon.

(d) Get information about the Nounphrase machine: print net Nounphrase and print words Nounphrase.

print net Nounphrase lists the transitions from each state. States are named sn or fsn depending on whether they are final; s0 or fs0 is the start state. The machine has been automatically determinized and minimized for us, since it is an acceptor rather than a transducer.

print words Nounphrase ought to print all words that can be accepted by the machine. However, there may be infinitely many such words. So the command only prints the words that can be accepted along acyclic paths in the machine (roughly speaking).

(Note: Each word is printed without spaces, e.g., QuantNounNoun. If you would prefer xfst to print spaces between the symbols of the word, first do set print-space ON.)

(e) Let’s see whether Art Adj Adj Noun is a noun phrase. Type the following:

```plaintext
define Input Art Adj Adj Noun ;
define Intersection Input & Nounphrase ;
push Intersection
```

```
test non-null
```

The intersection is not empty, so we conclude Art Adj Adj Noun is in the Nounphrase language.

(f) Shortcut: We could have put the intersection directly on XFST’s stack without naming it.3

---

3XFST has many other stack commands that let you manipulate and combine any number of machines
define Input  Art Adj Adj Noun ;  defines a straight-line automaton
regex Input & Nounphrase ;  puts intersection on stack
test non-null  is intersection empty set?
The regex command builds a machine and puts it on the stack in one step. You do have to use the stack here, because the command test non-null always applies to the machine on top of the stack. (So do the commands down and up that we'll see below.)

(g) Shortcut: We can also get away without building the straight-line automaton.
push Nounphrase  puts Nounphrase machine on stack
down ArtAdjAdjNoun  transduces ArtAdjAdjNoun through Nounphrase in the usual ("down") direction
Since the acceptor Nounphrase is interpreted as an identity transducer on the accepted strings, the output of the above is the same as the input. By contrast, down ArtAdjAdj has no output since ArtAdjAdj is not accepted. (Try it!)

Note: The down and up commands work on literal strings, not regular expressions, which is why we can't include space characters between the symbols. XFST does manage to interpret ArtAdjAdjNoun as a length-4 string over the tag alphabet. (It tokenizes by greedy left-to-right longest match; the capital letters are to help you read it, not XFST.)

(h) Define and try out a transducer that replaces Noun with Nmod immediately before any Noun:
define MakeNmod Noun -> Nmod || _ Noun ;
push MakeNmod
down FooBarNounBazNounNounBingNounNounNounNoun

(i) You can now do a composition:
define TransformNP Nounphrase .o. MakeNmod ;
push TransformNP
down ArtAdjNounNounNoun send string down through Nounphrase and then through MakeNmod
down VerbAdjNounNounNoun no outputs since Nounphrase won't let it through

(j) Let's build a machine that inserts angle brackets <> around the noun phrase in addition to otherwise transforming it:
define BracketNP 0:< TransformNP 0:<> ;
This machine reads 0, Nounphrase, 0 (where 0 denotes ε) and writes <, the transformed noun phrase, >. (Note that % is an escape character to ensure literal treatment of <>). Try it on the same strings as before. (As before, it will have without naming them, but this quickly gets confusing if you're not used to it. In this assignment, you never have to worry about machines that may be below the top of the stack.)
no outputs on \textit{down VerbAdjNounNounNoun}, which does not match \textit{Nounphrase} despite containing substrings that do.)

(k) The symbol \texttt{?} matches any character, so \texttt{?*} matches any string. If \texttt{?*} is used as a transduction, the usual rules mean it will be coerced to the transduction that maps any string to itself - i.e., leaves the input unchanged in the output.

Describe briefly but precisely what the transducer \texttt{?* [BracketNP ?*] *} does. Apply it to the strings \textit{VerbArtAdjNounNounNoun} and \textit{ArtAdj}. Hand in your answers.

(l) The following transducer greedily marks all noun phrases, using a left-to-right longest-match strategy:

\begin{align*}
\text{Nounphrase} & \rightarrow \%\{ \ldots \% \\
\emptyset & \rightarrow \text{calls for left-to-right longest-match replacement, and } \ldots \text{ stands for an output copy of whatever string was actually matched on the input side.}
\end{align*}

Try it on \textit{VerbArtAdjNounNounPrepArtAdjNoun}. Note that it only marks the NPs, without transforming \textit{Noun} to \textit{Nmod}. Its marks \texttt{\{\}} are intended to be an intermediate result, whereas the permanent brackets \texttt{<>} added by \textit{BracketNP} are intended to appear in the final output.

(m) Suppose you want like a transducer that “combines” the two previous answers, applying \textit{BracketNP} to bracket-and-transform NPs using a left-to-right longest match strategy.

To do this in a general way, we want to replace whatever it is that \textit{BracketNP} can match on the input side. This is the “upper language” or domain of \textit{BracketNP}, which is denoted \textit{BracketNP.u} and which happens to be equivalent to \textit{Nounphrase} in this case.

Here’s an attempt:

\begin{align*}
\text{BracketNP.u} & \rightarrow \%\{ \ldots \% \\
\mathrm{o.} & \text{ use } \%\{ \text{ to mark the substrings that } \text{BracketNP will replace } \}
\mathrm{.o.} & \text{ ... and then } \\
\texttt{?* [ %{:0 BracketNP %}:0 ?*] *} & \text{ transduce marked strings with } \\
& \text{BracketNP, also deleting } \%\{\}
\end{align*}

Try this on \textit{VerbArtAdjNounNounNounPrepArtAdjNoun}. It is still not quite right. It is better than question 1k in that it only ever replaces the two NPs marked by greedy left-to-right maximal matching—but because \texttt{?*} can match one or more of those marked NPs, our transducer can nondeterministically skip over some of the NPs without replacing them. You will fix that in the next question.

(n) Of all the nondeterministic results, the only one we want to keep is the one in which no marked NPs are left over. Define a regular expression \texttt{NoMarks} that matches strings that do not contain the character \texttt{\{}. (You may want to use
one or more of the operators ~, \, or $ — see the quick reference at http://cs.jhu.edu/~jason/405/software.html#comparison.)

Now a full solution is as follows:

```
BracketNP.u @-> %{ ... %}
.o.
?* [ %{:0 BracketNP %}:0 ?*]
.o.
NoMarks
```

Apply this to VerbArtAdjNounNounNounPrepArtAdjNoun. Hand in (i) the result, (ii) your definition of NoMarks, and (iii) a brief but precise explanation of why adding .o. NoMarks to the definition made it work (hint: see 1g).

Again, for the tutorial you only have to hand in answers to sections 1k and 1n.

2. Skim chapters 2–3 of the XFST book draft. (Even chapter 1 if you want.) You don’t have to read it carefully unless you get stuck; it is mostly a review of material we’ve covered in class.

The book draft can be found on the ugrad machines in the directory /usr/local/data/cs465/hw-xfst, as xfst-book.ps or xfst-book.pdf. (You could use the evince command to view either of these.) The book was published in 2003 (www.fsmbook.com) and is in the JHU library. However, the chapter, section and page numbers in this assignment refer to a pre-published draft that is online. **Our license does not allow you to make electronic copies of it - please respect this.**

Feel free to try out examples from the book. I rather like the Cola Machine example in section 2.6 (pp. 71) and its solution in Appendix H (p. 367). You don’t have to hand anything in.

3. You can build a non-probabilistic word segmenter very quickly. Download the file http://cs.jhu.edu/~jason/465/hw-xfst/words.txt, which contains the root forms of 25,143 English words, one per line.

```
read text < words.txt make an FSA that matches just those words, and push it on the stack
define Words assign it the name Words
```

(a) Construct a transducer, Segment, that reads a sequence of words without spaces and writes the words separated by spaces. Hand in the regular expression you used to construct Segment; it should refer to Words.

---

4A larger file that is not limited to root forms is the /usr/share/dict/words, standardly available on Linux systems. It contains 479,829 English words on the ugrad systems.
Warning: A regular expression that begins with \texttt{Words @->} will take an extremely long time to compile, for a large lexicon, but it’s probably not what you want anyway. Remember that \texttt{@->} is greedy (left-to-right longest match) and therefore deterministic.

(b) Testing \texttt{Segment} on \texttt{theprophetsaidtothecity} and other sentences of your choice, you will get a surprising number of segmentations. This is because \texttt{words.txt} contains many single-character words (e.g., initials).

Modify your definition of \texttt{Segment} so that the only single-character words it allows are “a” and “i.” You should now get only two segmentations for \texttt{theprophetsaidtothecity}.\footnote{Not including “the prophets aid to the city,” since predictable plural forms such as \texttt{prophets} do not appear in \texttt{words.txt}. You might think about how to fix this.}

Hand in the revised definition (and any interestingly ambiguous sentences you tested on!).

\begin{quote}
\textit{Note:} Your \texttt{Segment} transducer, with a bigger vocabulary, could have detected the embarrassing problems with these real domain names: PowergenItalia.com, ExpertsExchange.com, and WhoRepresents.com. I’m showing the intended segmentation with capital letters, but domain names are usually written in all lowercase.

(With the right vocabulary, even \texttt{IntroToNLP} could be segmented as \texttt{IntRotOnLP}. So is this course about how data will degrade when stored on analog media?)
\end{quote}

4. A solution to the Bambona exercise in section 3.3.3 (p. 103) is available in the script file \url{http://cs.jhu.edu/~jason/465/hw-xfst/bambona.scr}. It is a straightforward encoding of the grammatical principles given in the XFST book. (Figuring out such principles from raw data is the difficult job of linguists; writing them in XFST notation is the easy part.)

Download the script file and study it. You can run it by typing \texttt{source bambona.scr} at the XFST prompt; this will define all the machines.

Here’s a little lesson in linguistics. The \texttt{NounMorphology} transducer encodes knowledge of how morphemes in Bambona are pronounced and how they can be glued together in order to make nouns. The \texttt{Phonology} transducer encodes Bambona rules about how to fix up the joints after the morphemes are glued together. For example, Bambona speakers dislike pronouncing \texttt{pe}; if concatenating morphemes puts \texttt{p} and \texttt{e} next to each other, then the \texttt{Phonology} transducer changes the \texttt{pe} syllable to \texttt{po}, which is pronounced with the tongue farther back in the mouth.\footnote{Discussed in RISK Digest mailing list, July 2003.}
Thus, there are three levels of representation, as shown at the bottom of p. 107: lexical, intermediate, and surface. The lexical level is connected to syntax, and the surface level is connected to speech.

(a) A lexical analysis transducer will in general be nondeterministic. It might map a surface form to more than one lexical form.

Think of a surface representation with at least two lexical representations in English. Give the surface form and the two different lexical forms that you claim for it. You can represent the lexical forms in any reasonable way.

*Hint:* One solution is to think of a word like severer that is morphologically ambiguous in an interesting way.\(^7\) (Don’t use this example—think of your own!)

*Hint:* You could also try using homophones or homographs. But that requires you to think a little more about what the two lexical forms look like and why they are different from each other.

(b) Similarly, a lexical analysis transducer might map a surface form to fewer than one lexical form. Given an example of an input string where the FST would find no analyses (produce no output). That is, give a surface representation with no lexical representations in English.

*Hint:* This is a somewhat artificial question. If you get frustrated, try squeezing your eyes closed and banging on the keyboard for inspiration.

(c) Using XFST’s `up` command and the definitions from the script file, parse the following Bambona surface representations into lexical representations: ripotulozkon, kópuzmepog, poskugizmilek, natotópotólótol. Hand in your answers, which should be in the same style as the lexical forms on p. 107.

(d) According to the meanings on pp. 104–106, what do those lexical representations mean? (You don’t need xfst to answer this part!) Hand in your answers, which should be in the same style as the quoted translations on p. 107.

(e) Using XFST’s `up` command and the definitions from the script file, find the possible intermediate representations that correspond to the Bambona surface word ripotulozkon. Why aren’t there as many lexical representations for this word?

(f) What is a regular expression for the FSA that accepts exactly the set of surface nouns defined by the script file? (*Hint:* Use the domain (“upper”) or range (“lower”) operators.)

(g) Find a Bambona morpheme that can be pronounced in at least two ways depending on its context. Support your answer by giving two nouns in which that

\(^7\)severer could mean “someone who severs” (e.g., an executioner) or “more severe.”
morpheme is pronounced differently. Both nouns should be genuine Bambona nouns, i.e., they should be accepted by the regular expression in the previous question.

(h) The word “lexical” is just the adjective for “lexicon.” A language’s lexicon is a list of its morphemes or words—perhaps with some information about each one (e.g., semantics, part of speech, probabilities . . . ).

The lexicon is usually defined to hold the “arbitrary” facts of the language that must be listed. Hence it does not contain each word’s surface pronunciation, as that can be partly derived by rule. Bloomfield (1933) wrote: ”The lexicon is really an appendix of the grammar, a list of basic irregularities.” The morphologists DiSciullo & Williams (1987) famously characterized the lexicon as ”a prison—it contains only the lawless, and the only thing that its inmates have in common is lawlessness.”

Which definitions in bambona.scr define the Bambona lexicon?

5. **Extra credit (encouraged—think of it as practice for the finite-state problems on the final):** In this mission, should you choose to accept it, you will use XFST to implement Martin Porter’s popular heuristic algorithm for stripping suffixes off English words. Just as for the semantics questions in Assignment 3, you will not have to do the whole implementation, only fill parts in.

The files you will need for this extra-credit problem are in [http://cs.jhu.edu/~jason/465/hw-xfst/porter](http://cs.jhu.edu/~jason/465/hw-xfst/porter), or on the ugrad machines in /usr/local/data/cs465/hw-xfst.

- **porter-paper.txt**: Porter’s very short paper that describes his stemming algorithm. You should start out by reading this.⁸ The motivation for the algorithm is that when a search engine indexes documents, it should index the words CONNECTED and CONNECTIONS as if they were just CONNECT. This requires discarding a word’s suffixes to leave its stem.

  Note: “IR” stands for Information Retrieval.

- **realporter**: A correct implementation of the algorithm (written by Porter himself). You should be able to run it on the ugrad machines. Create a small text file foo.txt, and run `realporter foo.txt` to see the words together with their stems.

- **words**: A collection of lowercase English words. It was produced by the command `grep -v '^[^a-z]' /usr/share/dict/words > words`

⁸**Warning**: Jurafsky & Martin’s presentation has a few errors (contact me for details).
• **stems**: A collection of Porter stems of those words. It was produced by the command

```
realporter words | gawk '{print $2}' | sort | uniq > stems
```

You will use the **words** and **stems** files to test your implementation of the Porter stemmer.

(a) Stemming is a finite-state task. **porter.scr** is a mostly-complete XFST script that implements the stemmer.\(^9\) Remember that you can execute this script within XFST by typing `source porter.scr`. Alternatively, you can cut-and-paste portions of it into XFST.

Study **porter.scr**; figure out what is going on, perhaps by trying pieces out in XFST. Fill in the five missing expressions, which are marked with `?????`.

Testing the pieces of your implementation:

- You can test the smaller transducers Step1a, Step1b, etc. by using the convenient files **Step1a.tst**, **Step1b.tst**, etc. These contain the example words from Porter’s paper. Here is a sample dialogue:

```
xfst[0]: push Step1a
xfst[1]: down < Step1a.tst
Opening file Step1a.tst...
apply down> caresses
caress
apply down> ponies
poni
apply down> ties
ti
apply down> caress
cress
apply down> cats
ca
Closing file Step1a.tst...
```

To test all 5 steps separately, just type `source porter-test.scr`.

- When you think all 5 steps are working separately, run a full test as follows:

```
$ xfst
xfst[0]: source porter.scr
xfst[0]: source makestems.scr
```

\(^9\)It could be made more elegant if XFST had macros.
This will run your Stemmer transducer over the file words, producing a file of stems called mystems. (Look at the file makestems.scr for fun.)
If your transducer is working correctly, then mystems should be identical to stems, which Porter’s own stemmer produces. You can check this with
$ diff stems mystems
which will list the differences. To see the differences more clearly, if there are any, use the comm command. (Do man comm for documentation.) For example,
$ comm -3 stems mystems
$ comm stems mystems       # also shows similarities
Hand in your finished porter.scr. How many arcs and states does your Stemmer have?

(b) Now to amuse ourselves with the power of finite-state machines! Unlike the original Porter stemmer, transducers can be run backwards. You’ll use both down and up for this question.
   i. How many words have the same Porter stem as joy does (namely, the stem joi)? (These might not be “real” English words, but they resemble them.)
   ii. How many words have the same stem as sadness does?
   iii. How many words have the same stem as happiness does?
   iv. How many words have the same stem as unhappiness does?
   v. How many words have the same stem as gleefulness does? (And which one is your favorite?)

(c) Instead of using both down and up, build a single FST that will transduce a word to all the words that share its stem. Use it to check your answers to question 5b.
   Hand in the short expression you used to build the FST, as well as the number of arcs and states in the FST.

By the way, it is interesting to compare Porter’s stemmer with yours. Your code should be about 50 lines of XFST definitions; Porter’s is about 170 lines of procedures (if we count just the actual stemming code), most of which are really multiple lines without a line break.

The downside is that your porter.scr is, surprisingly, a little slower. Mainly, this is because the machine it builds must still be interpreted by xfst. But there are finite-state packages that will turn an FST into a piece of C code that can be fully compiled.

You may be interested to see the official Porter stemmer homepage, http://www.tartarus.org/~martin/PorterStemmer, and Porter’s work on extending it to other languages, http://snowball.tartarus.org.