600.465 — Intro to NLP
Assignment 4: Finite-State Programming

Prof. J. Eisner — Fall 2004
Due date: Friday 19 November, 2pm

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, 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.[1] 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 `Adj`, followed by one or more nouns `Noun`.

     To make things slightly more interesting, determiners fall into two types, quantifiers ("every") and articles ("the"). These are assumed to have different tags `Quant` and `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 bracketed and transformed as above.

   (a) Make sure that `/usr/local/xerox/bin` is on your `PATH`. (It is by default.)

[1] It is a slightly more straightforward and self-contained version of the tutorial at `http://cs.jhu.edu/~jason/405/software.html#xfst`. (Ignore the backslash in that URL, it's a typesetting bug.)

(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.

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 words Nounphrase` and `print net Nounphrase`.

The former command appears to list all words that can be accepted along acyclic paths in the determinized machine.

The latter command 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.

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

```
define Input        Art Adj Adj Noun ;      defines a straight-line automaton
define Intersection Input & Nounphrase ;    intersects it with Nounphrase
push Intersection                           puts result on XFST's stack
                                              so we can work with it
test non-null                               is intersection empty set?
```

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:[2]

```
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`.)

[2]XFST has many other stack commands that let you manipulate and combine any number of machines 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.

(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 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 nounphrase, `>`. (Note that `%` is an escape character to ensure literal treatment of `<>`.) Try it on the same strings as before. (As before, it will have no outputs on `down VerbAdjNounNounNoun`, which does not match `Nounphrase` despite containing substrings that do.)

(k) The symbol `?` matches any character, so `?*` matches any string. If `?*` 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 `?* [BracketNP ?*]*` does. Apply it to the strings `VerbArtAdjNounNounNoun` and `ArtAdj`. Hand in your answers.

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

```
Nounphrase @-> %{ ... %}
```

`@->` calls for left-to-right longest-match replacement, and `...` stands for an output copy of whatever string was actually matched on the input side.

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

(m) Suppose you want like a transducer that "combines" the two previous answers, applying `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 `BracketNP` can match on the input side. This is the "upper language" or domain of `BracketNP`, which is denoted `BracketNP.u` and which happens to be equivalent to `Nounphrase` in this case.

Here's an attempt:

```
BracketNP.u @-> %{ ... %}         use {} to mark the substrings
                                  that BracketNP will replace
   .o.                            ...and then ...
?* [ %{:0 BracketNP %}:0 ?*]*     transduce marked strings with
                                  BracketNP, also deleting {}
```

Try this on `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 `?*` 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 `NoMarks` that matches strings that do not contain the character `{`. (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.

   This draft is online at `file:/usr/local/xerox/doc/xfst-book.ps` on both the undergrad and graduate networks. The book was finally published in 2003 (`www.fsmbook.com`) and is in the JHU library. However, the chapter, section and page numbers in this assignment refer to the *online*, pre-published draft. **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. `/usr/dict/words` is a standard file on Unix systems; it lists root forms for many English words, one per line.

`read text < /usr/dict/words`	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`.

   Warning: A regular expression that begins with `Words @->` will take an extremely long time to compile, for a large lexicon, but it's probably not what you want anyway. Remember that `@->` is greedy (left-to-right longest match) and therefore deterministic.

   (b) Testing `Segment` on `theprophetsaidtothecity` and other sentences of your choice, you will get a surprising number of segmentations. This is because `/usr/dict/words` contains many single-character words (e.g., initials).

   Modify your definition of `Segment` so that the only single-character words it allows are "a" and "i." You should now get only two segmentations for `theprophetsaidtothecity`.[3]

[3]Not including "the prophets aid to the city," since predictable plural forms such as `prophets` do not appear in `/usr/dict/words`. You might think about how to fix this.

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

Note: Your `Segment` transducer, with a bigger vocabulary, could have detected the embarrassing problems with these *real* domain names:[4] `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 `IntroToNLP` could be segmented as `IntRotOnLP`. So is this course about how data will degrade when stored on analog media?)

4. A solution to the Bambona exercise in section 3.3.3 (p. 103) is available in the script file `http://cs.jhu.edu/~jason/465/hw4/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 `source bambona.scr` at the XFST prompt; this will define all the machines.

   Here's a little lesson in linguistics. The `NounMorphology` transducer encodes knowledge of how morphemes in Bambona are pronounced and how they can be glued together in order to make nouns. The `Phonology` transducer encodes Bambona rules about how to fix up the joints after the morphemes are glued together. For example, Bambona speakers dislike pronouncing `pe`; if concatenating morphemes puts `p` and `e` next to each other, then the `Phonology` transducer changes the `pe` syllable to `po`, which is pronounced with the tongue farther back in the mouth.

   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. (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.

[4] Discussed in RISKS Digest mailing list, July 2003.

(b) Similarly, a lexical analysis transducer might map a surface form to *fewer* than one lexical form. Give an example of 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`, `natotópotulótol`. 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 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:* 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/hw4/porter:

- `porter-paper.txt`: Porter's very short paper that describes his stemming algorithm. You should start out by reading this.[5] 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 barley. 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/dict/words > words
  ```
- `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.[6] 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:

[5] *Warning:* Jurafsky & Martin's presentation has a few errors (contact me for details).

[6] It could be made more elegant if XFST had macros.

```
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
caress
apply down> cats
cat
Closing file Step1a.tst...
xfst[1]:
```

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:

  ```
  barley> xfst
  xfst[0]: source porter.scr
  xfst[0]: source makestems.scr
  ```

  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

  ```
  barley> 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,

  ```
  barley> comm -3 stems mystems
  barley> 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 stem as `joy` does (namely, `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`.