Natural Language Processing (JHU 601.465/665)
Answers to "Finite-State Machines" practice problems

1. (a) There are many possible solutions. A few are shown
below. If you want to test them out in xfst, make sure
to define Digit as %0|1|2|3|4|5|6|7|8|9.
%0 is needed since 0 by itself means epsilon.

Note: Common mistakes include transducing 123 to ,123.

ANSWER 1
Use the right-to-left replacement operator ->@
(instead of the left-to-right operator @->):

```
Digit Digit Digit ->@ %, ... // Digit _ ;
```

ANSWER 2
Direct description:

```
(%-) (Digit) (Digit) Digit [0:%, Digit Digit Digit]*
```

ANSWER 3
Describe what the desired output looks like:

```
define CommaNumber (%-) (Digit) (Digit) Digit [%, Digit Digit Digit]* ;
```

Now introduce a noisy channel:

```
define DelCommas %, -> 0
```

Reverse the noisy channel:

```
[CommaNumber .o. DelCommas].i
```

ANSWER 4
Generate too many solutions and filter the bad ones out:

```
Digit Digit Digit (->) %, ...    # optional comma before any group of 3 digits
.o. [ %, => Digit _ ]            # every comma must be preceded by a digit
.o. ~$[Digit Digit Digit Digit]  # don't allow 4 digits without any commas
```

(b) The game here is to take a solution to 23(a) that is only
guaranteed to work correctly on numbers, and adapt it to
work on arbitrary text. You don't know whether InsCommas
is defined as Answer 1 or Answer 2 above.

Let's define IC to be a version of InsCommas whose domain
is restricted to numbers, since we're not guaranteed that
InsCommas has this property:

```
[ (-) Digit+ ] .o. InsCommas
```

Now it is tempting to write

```
~$[IC.u] [ IC ~$[IC.u]]*
```

but it won't work since it allows the input "12345" to
be broken up into "123" and "45" and transduced separately.
(On the other hand, 123-45 *should* be broken up that way, so
we can't just test for adjacency.)

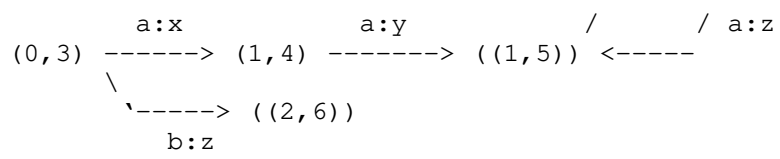
So we really need to replace longest matches. Here's a
general approach:

```
[ IC.u @-> %{ ... %} ]             # bracket what we should replace
.o. \%{* [ %{:0 IC %}:0 \%{* ]*    # replace all bracketed substrings
```

or perhaps more simply

```
[ IC.u @-> %{ ... %} ]          # bracket what we should replace
.o. ?* [ %{:0 IC %}:0 ?* ]*     # optionally replace some bracketed strings
.o. ~$%{                        # keep only solutions where they were all replaced
```

2.



3. 1. Let A,B be the two original FSAs and let C be their intersection.

   a. C has at most nm states, because each state of C corresponds to a different pair (q,r), where q is one of the n states of A and r is one of the n states of B.

   b. C could be as small as 1 state, namely (q0,r0) where q0 is A's initial state and r0 is B's initial state. This minimum is achieved iff it is impossible for C to get away from (q0,r0) -- that is, if there are arcs q0 -> q1 and r0 -> r1 that are labeled with the same symbol, then they are both self-loops, i.e., q1=q0 and r1=r0.

      (The answer gets more complicated if A and B are allowed to have epsilon arcs, or if you use a more complicated intersection algorithm that trims away useless states after intersection completes. However, we didn't discuss those cases in class.)

      Certainly the only way for C to have one state is if the intersection of the two languages (that are described by A and B) is a very simple language that can actually be described by an FSA with only a single state. Examples: a*, (a|b)*, the empty language, the language consisting only of epsilon.

      But even if the intersection language CAN be described by a single-state FSA, our intersection construction is not guaranteed to find that single-state FSA. Depending on how A and B are written, it might find some equivalent but larger FSA; in that case, one would have to use the FSA minimization algorithm (from 600.271) to shrink it down to the single-state version.

4. (a) One possible answer is

```
    [ ?* .x. ?* ] ? ? ?
```

   Another possible answer is

```
    [ ?* -> ?* ] ? ? ?
```

   Another possible answer is

```
    [ ?* -> ?* ] // ? ? ? .#.
```

   A few of you wrote [?:?][?:?][?:?] instead of ???. That's wrong, since [?:?] says to match any character to any (other) character. In that case you are not requiring the 3-letter endings to be identical.

   What you want is ???. This is the language that accepts any 3-character sequence. So if used in an FST context, it is coerced into the identity relation on 3-character sequences, which matches any 3-character sequence TO A COPY OF ITSELF.

   I'm afraid many of you wrote regular expressions that referred to mysterious undefined names like Input, Output, Pair, etc. This suggests that you don't understand how regular expressions work. A regular expression is not written like a Java function like

```
    transduce(String input)
```

   or

```
    check(String upper, String lower)
```

   In particular, it does not have named arguments that are single

strings that it can manipulate.

Rather, a regular expression defines a relation -- the set of ALL string pairs on which such a check function would return true. It is built up by combining other regular expressions that define simpler relations (simpler sets).

(b) Ending .o. Ending.i

Recall that the .i operation ("inversion") swaps upper and lower languages. So this composition says to first map the upper word to its ending, and then reverse-map that ending back to any other word that has the same ending.

Perhaps a better way of thinking about it is in terms of sets:

```
Ending = {(w,e): e is the ending of w}
Ending.i = {(e,w): e is the ending of w}
Ending .o. Ending.i = {(w1,w2): there exists e such that (w1,e) is in
                                Ending and (e,w2) is in Ending.i}
                    = {(w1,w2): there exists e such that e is the ending
                                of both w1 and w2}
```

A way of checking the answer is to observe that if you wanted to use the FST above to check whether two PARTICULAR strings (x,y) were Rhymes, you would find out whether it accepted the pair (x,y) by asking:

```
Is this nonempty?    x .o. [Ending .o. Ending.i] .o. y
```

That is equivalent to asking:

```
Is this nonempty?    [x .o. Ending] .o. [Ending.i .o. y]
Is this nonempty?    [x .o. Ending].l & [Ending.i .o. y].u
Is this nonempty?    [x .o. Ending].l & [y .o. Ending].u
```

in other words, do x and y have an ending in common?

(c) Most of you did pretty well on this one. Like your FST homework, it is kind of like procedural programming -- a composition of deterministic steps. (Problems (a)-(b) above ask you to produce FSTs that are strongly nondeterministic, requiring you to think more declaratively about what they are doing.)

Here is a solution:

```
Pronounce                          pronounce the word
   .o.
[ ?* @-> 0 // _ ' ]                delete the longest thing
                                   you can find that immediately precedes
                                   a stress mark (this will include earlier
                                   stress marks)
   .o.
[ Consonant* -> 0 // ' _ ]         delete initial Consonant* after the '
   .o.
[ [ ' | - ] -> 0 ]                 delete all remaining - or '
                                   (should really quote - since it is a
                                   special character, instead writing
                                    %- or "-")
```

There are other ways to solve parts of this. For example, you can replace the second step by this:

```
[ ? -> 0 // _ ?* ']      delete any character that has a
                         stress mark somewhere to its right
```

(Another option is offered in part (d).)

Or you can replace the second and third steps by this:

```
[ ?* ' @-> 0 ]           delete the longest thing you can find
                         that ends in a stress mark (gets
                         rid of the stress mark too)
```

```
        .o.
    [ Consonant:0 ]* ?*     delete initial string of consonants
```

It is worth remarking that Pronounce may be nondeterministic; some words have multiple pronunciations. As a result, Ending may be nondeterministic. A word could have multiple endings.

As a result, Rhyme would accept both
    (lead,feed)
    (lead,fed)
based on different pronunciations of lead, but it would not accept
    (feed,fed)
This shows that Rhyme does not have to be an equivalence relation.

(d) The version above will not delete anything at step 2 or 3, since the contexts don't match. Therefore, the ending will simply be the pronunciation with all - symbols removed.

For example, Ending will map
    (of, uv)
Since Ending also maps
    (love, uv)
despite the stress, Rhyme will accept the rhyme
    (of, love)

However, it is possible to write other answers to (c) that will simply not return any Ending for of. For example, suppose we had written the second step of the composition cascade above as

```
   [?:0]* ' [?-']*
```

which is arguably simpler. This says to delete everything for a while, then match a stress mark ', and then keep all remaining characters, which are NOT ALLOWED to be '. This final requirement means that the stress mark which is matched will be the rightmost one.

The unstressed pronunciation uv cannot match the upper language for this transducer, because it has no stress mark. As a result, no lower string will be produced. So "of" will not make it through the Ending cascade: we will find no Endings for it. This is the flip side of the "lead" example above, where we found > 1 Ending and hence found a lot of Rhymes; here we find < 1 ending and hence find no Rhymes.

In short, an unstressed word will be considered not to rhyme with anything, not even with itself!

A different problem would occur if the second and third steps in (c) had been replaced with this, which does both steps at once:

```
    [ ?* ' Consonant* @-> 0 ]
```

Here, if there is no stress mark, then the onset is left in place. So "of" will be allowed to rhyme with "love," but "the" will not rhyme with anything other than "the". (This is a bizarre definition of rhyme, in my opinion.)

(e) i. Make a straight-line automaton, Word, that accepts just the string "kangaroo meringue."
Compute the weighted intersection

```
   [ Word & Score1 ]
```

and use Dijkstra's algorithm or Viterbi's algorithm to find the maximum-weight path.

ii. Compute the weighted composition [Score1 .o. Rhyme .o. Score2] and use Dijkstra's algorithm to find the maximum-weight path.

(You can't use Viterbi's algorithm -- which is a special case of Dijkstra's algorithm for acyclic graphs -- because [Score1 .o. Rhyme .o. Score2] may accept an infinite number of rhyming string pairs, meaning that it has cycles.)

(f) The basic idea is to define LongEnding just as you defined Ending before, except that it doesn't strip the initial consonant cluster. Instead, it puts a "/" after the initial consonant cluster. I won't give the details since it should be a simple difference.

Now StrictRhyme can be defined as

```
LongEnding .o. ChangeOnset .o. LongEnding.i
```

We can define ChangeOnset like this:

```
Change "/" ?*
```

where Change has the behavior described in the question.

We can define Change as

```
?* [ [ ChangeLetter [?* .x. ?*] ] | [ 0 .x. ?+ ] | [ ?+ .x. 0 ] ]
```

which allows the strings to be equal for a while via ?*, but then insists that they must do something be different, which can happen in one of three ways:

* either the next letter is different for both of them
* the first string ends while the other keeps going
* the second string ends while the other keeps going

(A cleaner trick for handling the second two cases is to augment the alphabet by a new letter Z, and append a copy of Z to the end of both strings. Then if they are different, they must be different because some letter is different.)

Here you need to define ChangeLetter to be, in effect, ?:? - ?. You can do this by enumerating all pairs of unequal letters, e.g.,

```
a:b | a:c | a:d | b:a | b:c | b:d | c:a | c:b | c:d
```

Note that it would not work to define Change as

```
[?* .x. ?*] [ ChangeLetter | 0:? | ?:0 ] [?* .x. ?*]
```

because this actually would accept (money,money). Do you see why? The first subexpression could match (m,mon), the second could match (o,e), and the third could match (ney,y).