600.406 — Finite-State Methods in NLP, Part II
Assignment 4: Building Finite-State Operators

Solution Set
Prof. J. Eisner — Spring 2001

1. (a) $A \;\mathsf{xx}\; B \stackrel{\text{def}}{=} \{\langle a, b\rangle : a \in A, b \in B, |a| = |b|\}$

   (b) First eliminate ϵ's from A and B (by full determinization or just ϵ-closure). Now perform a cross-product construction much like the one used for intersection or composition. The key step is that if A has an arc $q \xrightarrow{a} q'$ and B has an arc $r \xrightarrow{b} r'$, then $A \;\mathsf{xx}\; B$ should have an arc $\langle q, r\rangle \xrightarrow{a:b} \langle q', r'\rangle$. Unlike intersection, any symbol in A can be matched with any symbol in B.

   (c) This question is harder than I intended. The relation $A \;\mathsf{xx}\; B$ is a *function* iff B contains at most one length-$|a|$ string for every $a \in A$. However, being a function is weaker than being sequential; accordingly, this condition is necessary but not sufficient for sequentiality.

   For a counterexample consider $A = \{u^m\}$, $B = \{v^{2n}\} \cup \{w^{2n+1}\}$. These satisfy the condition above (hence $A \;\mathsf{xx}\; B$ is a function), but $A \;\mathsf{xx}\; B$ is the classic nonsequential relation $\{\langle u, v\rangle^{2n}\} \cup \{\langle u, w\rangle^{2n+1}\}$.

   On the other hand, if we change A to $\{u^{2n}\} \cup \{x^{2n+1}\}$, then $A \;\mathsf{xx}\; B$ becomes sequential (even though we have not changed the lengths of strings in A). These two examples together suggest that in general, determining the (sub)sequentiality of $A \;\mathsf{xx}\; B$ may be no easier than determining the (sub)sequentiality of an arbitrary regular relation (e.g., by the twins property).

   (d) $E \circ \mathsf{?}^* \circ F$

2. (a) Skip step (G).

   (b) The intent of this question was that if the stochastic process declined (nondeterministically) to replace a longest match, then it should continue as usual at the next available point—skipping over just one character, not over the entire longest match. For example, `replace_nondeterm`$(aa : b, \epsilon, \epsilon)$ should transduce aaa to the set $\{aaa, ba, ab\}$, not just $\{aaa, ba\}$.

Your answers missed this point: they tried to modify (E) so that a substring y' surrounded by `<`$_1$ and `>`$_1$ would be nondeterministically replaced by $T(y')$ or left alone. This is equivalent to `replace`$(T \cup$ `domain`$(T), L, R)$, and does not have the intended effect.

The correct answer is to modify (B) so that before each `domain`(T)`>`$_2$, it inserts `<`$_2$ *with probability* p (and ϵ with probability $1-p$). It will then fail to see any matches to `domain`(T) starting at the points where it declined to insert `<`$_2$.

(c) In step (C), don't replace `<`$_2$`domain`(T)`>`$_2$ if it contains `>`$_2$ internally. Also get rid of step (D).

(d) Oops! My intended answer to this one doesn't quite work. Sometimes you have to start writing the solutions before realizing that. :-)

My idea was the same as in the answer to (b): randomly remove some of the matches to `domain`(T). After step (B), just stochastically delete some of the `>`$_2$ marks. Each `>`$_2$ mark should be retained with independent probability p (and replaced by ϵ with probability $1-p$). Then continue as in shortest-match replacement.

This can be accomplished with a simple one-state weighted transducer, described by the slightly less simple regexp

$$\backslash{>_2}^* \; (\; \{>_2 : >_2 : p \; , \; >_2 : \epsilon : (1-p)\} \; \backslash{>_2}^* \;)^*$$

Unfortunately, the probabilities now are not independent as requested. If the transducer declines to replace a match ending at position k, then it will later decline to replace any later-starting match that also ends at k. I doubt this can be fixed, although perhaps a useful variant is still possible.

(e) The idea was to stochastically delete some of the `>`$_2$ marks after step (B), as above, but to continue as in longest-match rather than shortest-match replacement. But again, this answer doesn't quite work.

3. For each tag pair (x, y), let R_{xy} be the sequential transducer `replace`$(\epsilon : \epsilon : p_{xy}, x, y)$, which leaves the input string alone but multiplies its weight by p_{xy} each time xy appears in the input.[1] Note that the first argument of `replace` transduces ϵ to ϵ

[1]There are several perfectly good ways to write R_{xy}.

but with weight p_{xy}. Now compose all the R_{xy} transducers together in any order to get the weighted transducer R. Since R is a weighted identity transducer, it is indistinguishable from a weighted acceptor as desired.

To handle the edges of the string correctly, the above construction must allow x and y to be the special symbols `^` and `$`, which match the start and end of the string respectively. In XFST, these symbols are called `.#.` and `.#.`. If they are not implemented at all, as in the FSA Utilities, one can add them and remove them before applying R: just write $E \circ R \circ E^{-1}$, where $E = (\epsilon : \hat{})\texttt{?}^*(\epsilon : \$)$.

4. (a) i. A binary constraint C_i (a regular language) can be equivalently implemented as a counting constraint (a regular relation) that acts as the identity on strings in C_i and inserts a single star into other strings. Specifically, the counting constraint may be written as $C_i \cup (\epsilon : \texttt{*})(\tilde{}C_i)$.

ii. Following Karttunen (1998), but using FSA Utilities notation,

```
:- op(402,yfx,'oo').  % declare oo as an infix operator
macro(punion(Q,R), {Q, ~domain(Q) o R}).
macro(T oo C, punion(T o C, T)).
```

iii. Define $V_i \stackrel{\text{def}}{=} \tilde{}(\texttt{?}^*(\star\texttt{?}^*)^i)$, the language of strings with fewer than i stars. Now put $C_i \stackrel{\text{def}}{=} \texttt{domain}(C \circ V_i)$ is the language of strings to which C assigns fewer than i stars. Now T `oo` C_1 `oo` C_2 `oo` C_3 gives T `o+` C as desired. (Note that $C_i = \texttt{?}^*$ for $i \geq 4$, since by assumption C always assigns fewer than 4 stars.)

(b) A completed version of `otdir.plg`, with the definitions filled in, is available on request. Here are the definitions. Remember that multiple correct answers are possible for `lang_one` through `lang_seven`; only one is given here.

i.
```
macro(constraint(Lif,Rif,Lthen,Rthen),
      addstarwhere(Lif,Rif) o delstarwhere(Lthen,Rthen)).
```

ii.
```
macro(surfconstraint(Lif,Rif,Lthen,Rthen),
      constraint(ignore(Lif,deep) & ~[? *, deep],
                 ignore(Rif,deep),
                 ignore(Lthen,deep) & ~[? *, deep],
                 ignore(Rthen,deep))).
```

The `~[? *, deep]` clauses are necessary to ensure one star per violation. If the constraint is supposed to put a star between `A` and `B` on the surface, then these clauses ensure that `AcccB` is transduced to `A*cccB` rather than `A*c*c*c*B`. Of the 4 positions that are between `A` and `B` if

deep characters are ignored, we only consider the leftmost one (the one not preceded by a deep character).
(Actually, the extra clause on `Lthen` looks unnecessary to me now, but I haven't tried removing it.)

iii.
```
macro(noins,
      constraint(surfseg,[],corrpair,[])).
```

iv.
```
macro(onset,
      surfconstraint(lsyl,[],[],surfcons)).
```
Every `[` must be immediately followed on the surface by a consonant.

v.
```
macro(nocomplex,
      surfconstraint(surfcons,surfcons,{},{})).
```
This states the constraint very directly: it says that two adjacent surface consonants always deserve a star, with no way out (since `Lthen` and `Rthen` are the empty language {}).

vi.
```
macro(singlenuc,
      surfconstraint(surfvowel,
                     ignore(surfvowel,surfcons),
                     {},{})).
```

vii.
```
macro(worsen_lr, [? *, ([]:star)+,
                  ['star, (star*):(star*) ]*]).
```

viii.
```
macro(prune_lr(TC),
      pragma([TC], TC o ~range(TC o elim(surf)
                                  o worsen_lr
                                  o intr(surf)))).
```

ix.
```
macro(T do C, reverse(reverse(T) od reverse(C))).
```

x.
```
macro(lang_one,   gen od nucleus od singlenuc
                      od syllabify od nodel od noins
                      od nocomplex od onset
                      o elim(deep)).
```

xi.
```
macro(lang_two,   gen od nucleus od singlenuc
                      od syllabify od nodel od noins
                      do nocomplex od onset
                      o elim(deep)).
```

xii.
```
macro(lang_three, gen od nucleus od singlenuc
                      od nocomplex od nodel od noins
                      od syllabify od onset
```

```
                                o elim(deep)).
xiii. macro(lang_four,  gen od nucleus od singlenuc
                            od nocomplex od syllabify
                            od noins od nodel od onset
                            o elim(deep)).
 xiv. macro(lang_five,  gen od nucleus od singlenuc
                            od nocomplex od syllabify
                            od noins do nodel od onset
                            o elim(deep)).
  xv. macro(lang_seven, gen od nucleus od singlenuc
                            od nocomplex od syllabify
                            od nodel do noins od onset
                            o elim(deep)).
```

(c) There are in fact quite a few possible answers for `lang_three`. It is instructive to look at the whole taxonomy. One must begin by requiring syllables to be well-formed:

```
gen od nucleus od singlenuc ...
```

One must end by asking that as much as possible be syllabified, and other things equal, that these syllables have onsets (e.g., to get `[DA][BEC]` rather than `[DAB][EC]`):

```
... od syllabify od onset
```

In between, the `nodel` and `noins` must dominate `syllabify`, because in `[AB]C[DE]`, we prefer letting the `C` go unsyllabified to deleting it or inserting a vowel:

```
... od nodel od noins ...
```

or

```
... od noins od nodel ...
```

The real question is the position of `nocomplex` with respect to all these constraints. Are we willing to insert or delete material (or syllable boundaries) to avoid `nocomplex`?

If `nocomplex` is ranked below `syllabify`, then we are willing to violate it in order to get everything satisfied. But it still matters whether we prefer to violate it late (`od`) or early (`do`). I'll use {} to indicate sets of constraints for which it doesn't matter in what order they're introduced or whether they're introduced with od or do (either because they are ranked too high to be violated, or they are ranked too low to have any choices left):

```
... {nodel,noins} {syllabify} od nocomplex ...: [AB][CCCCCDE]
... {nodel,noins} {syllabify} do nocomplex ...: [ABCCCCC][DE]
```

If nocomplex is more important than getting everything syllabified, but not important enough to justify insertion or deletion, then we will be able to avoid one more violation of it, at syllabify's expense:

```
... {nodel, noins} od nocomplex {syllabify} ...: [AB]C[CCCCDE]
... {nodel, noins} do nocomplex {syllabify} ...: [ABCCCC]C[DE]
```

If nocomplex is important enough, then we will be willing to insert or delete in order to avoid violating it. Here's what happens if noins is the least important of noins, nodel, nocomplex, and so gets violated (here V can be any vowel, so we get multiple outputs):

```
... {nodel,nocomplex} od noins {syllabify} ...: [AB]C[CVC]C[CV][DE]
... {nodel,nocomplex} do noins {syllabify} ...: [A][BVC]C[CVC]C[DE]
```

Finally, if nodel is low man on the totem pole, then it is the constraint that has to carry the violations. We end up deleting all but the first three or all but the last three consonants:

```
... {noins,nocomplex} od nodel {syllabify} ...: [AB]C[CE]
... {noins,nocomplex} do nodel {syllabify} ...: [AC]C[DE]
```

(d) The easiest solution is to add a constraint that prohibits E. IF this is placed at the bottom of the hierarchy, it is only used to break ties when there are multiple solutions. So it won't result in deleting an input E (as it would if ranked about nodel); it just prefers that A is inserted instead.

(e) One should ignore stars in substrings for purposes of matching them against the contexts L and R. Note that in the case of addstarwhere, we do not bother with ignore for the right context: that's because the input string is unstarred, and the right context has not yet been starred when it is checked during left-to-right directed replacement (replace).

(f) Since filtering by deep* o has no effect on strings that match deep*, of course the grammar will remain correct on such strings if we remove the filter. But the filter considerably reduces the size of the FST, because it spares the FST from having to deal with cases where the input contains surface characters. (Our macros define some behavior on such cases, but not a behavior that we planned or cared about when defining the macros.)

(g) i. 5 states in `syllabify` (Use the "Count FA" button at the bottom of the GUI window.)

ii. 4 states in `t_minimize(syllabify)`

iii. 3 states in `syllabify`, and 2 states in `t_minimize(syllabify)`, after changing the definition of `constraint` to

```
macro(constraint(Lif,Rif,Lthen,Rthen),
      range(gen) o addstarwhere(Lif,Rif)
        o delstarwhere(Lthen,Rthen)).
```

iv. The transducer for `syllabify` is quite easy to understand: it inserts a star after any surface segment (capital letter) that is not between brackets. The minimized version is more complicated because the minimization algorithm operates on sequential FSTs, so it has to start by determinizing the FST and in particular eliminating epsilons. In state 0 (outside brackets), it copies letters, inserting a star after each capital letter:

```
$@(A..E):[$@(a..e),*]   $@(A..E):[$@(a..e)]
```

In state 1, it does the same but never inserts stars. Open and close brackets allow it to switch between the states.

(h) `t_minimize(lang_five)` has 18 states and 98 arcs. Setting S to the disjunction `{a,b,'A','B','[',']'}`, we can define our smaller version as $S^* \circ$ `lang_five` $\circ S^*$, whose minimized version has only 5 states and 10 arcs.

(i) i. A reasonable strategy would be to encode input-only letters as `iA`, `iB`, `iC`,..., output-only letters as `Ao`, `Bo`, `Co`,..., and letters that are faithfully copied from input to output (so they appear in both) as `iAo`, `iBo`, `iCo`,.... Thus, Gen should insert `i` before each letter of the input string, freely insert `o` after some of those letters, and should also freely insert output-only letters such as `oC`. Now, for example, `nodel` checks that any letter preceded by `i` is followed by `o`: `constraint([i,?],[],[],o)`.

ii. The following relation will swap either `x` or `y` with the preceding character. For example, it will map `ax` to `xa`.

```
{[[]:x,?,x:[]],[[]:y,?,y:[]]}
```

This is tricky to determinize, because a deterministic version has to wait to see both characters before it can output either one. In other words, it has to remember `a` (or any character) while waiting to see whether the next character is `x` or `y`; output the `x` or `y`; and then output the remembered character. In the determinized machine, an arc `$@(?):[]` from state 0 to state 1 remembers and deletes a character `?`. Then an arc `x:[x,$@(?)]` or `y:[x,$@(?)]`, from state 1 to state 2, replaces `x` (or `y`) by `x` (or `y`) followed by the remembered character. Remember that `$@(···)` denotes a backreference to something matched in the input, and this example shows that the backreference can refer to a different arc.

As implemented, this technique cannot be used to swap arbitrary substrings, which would be beyond finite-state power. It is crucial that `x` and `y` are actually hardcoded into the machines above. For example, there is no way to give a regexp for a machine that swaps two *arbitrary* characters or substrings: backreferences will always output substrings in the same order as they were input. Moreover, backreferences must be to single characters, not to arbitrary substrings. Thus, determinizing the first expression below uses a sequence of two backreferences, but determinizing the second fails to terminate!

```
{[[]:x,?,?,x:[]],[[]:y,?,?,y:[]]}
{[[]:x,? *,x:[]],[[]:y,? *,y:[]]}
```