Pronunciation Modeling for LVCSR **Summer 2000**

Practical Session I: AT&T Finite State Tools

TeSTIA: The 8TH ELSNET European Summer School *Instructor: Eric Fosler-Lussier*

These exercises are intended to familiarize you with the AT&T Finite State modeling tools. This is a powerful toolkit that will allow you to manipulate finite state automata and transducers of both the weighted and unweighted variety.

Finite state machines (that is, automata and transducers) are compiled from a human-readable ascii format, specifying the transitions between states, into a binary format used by the FSM libraries. If you use labels on your transitions that are non-numeric, you must provide an alphabet key. For example, here is an (ascii) finite state automaton that recognizes English digits:

```
paprika$ cat digits.stxt
0 1 ONE
0 1 TWO
0 1 THREE
0 1 FOUR
0 1 FIVE
0 1 SIX
0 1 SEVEN
0 1 EIGHT
0 1 NINE
0 1 ZERO
1
```

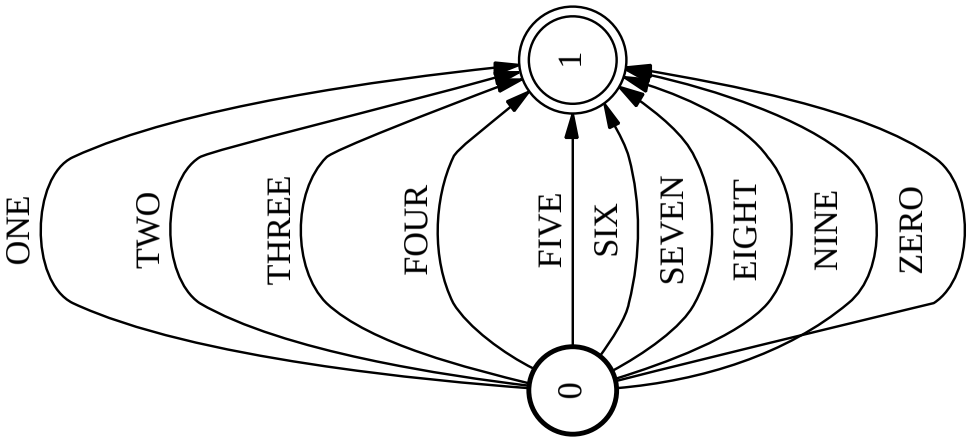
The "1" on a line by itself indicates that state 1 is a final state. This is the alphabet key corresponding to the digits:

```
paprika$ cat digits.alpha
epsilon 0
ONE     1
TWO     2
THREE   3
FOUR    4
FIVE    5
SIX     6
SEVEN   7
EIGHT   8
NINE    9
ZERO    10
```

The symbol number 0 in any alphabet file is interpreted as an *epsilon* transition marker – that is, when accepting this language, the machine can move between states without consuming any input.

To compile the ascii FSM into a machine usable format, the appropriate command is `fsmcompile`. You can decompile (i.e. print) a binary FSM using `fsmprint`, or draw a prettier version of it using the `fsmdraw` and `dot` commands.

```
prompt> fsmcompile -i digits.alpha digits.stxt > digits.fsm
prompt> fsmprint -i digits.alpha digits.fsm
0       1       ONE
0       1       TWO
0       1       THREE
0       1       FOUR
0       1       FIVE
0       1       SIX
0       1       SEVEN
0       1       EIGHT
0       1       NINE
0       1       ZERO
1
prompt> fsmdraw -i digits.alpha digits.fsm | dot -Tps > digits.ps
prompt> ghostview digits.ps
```

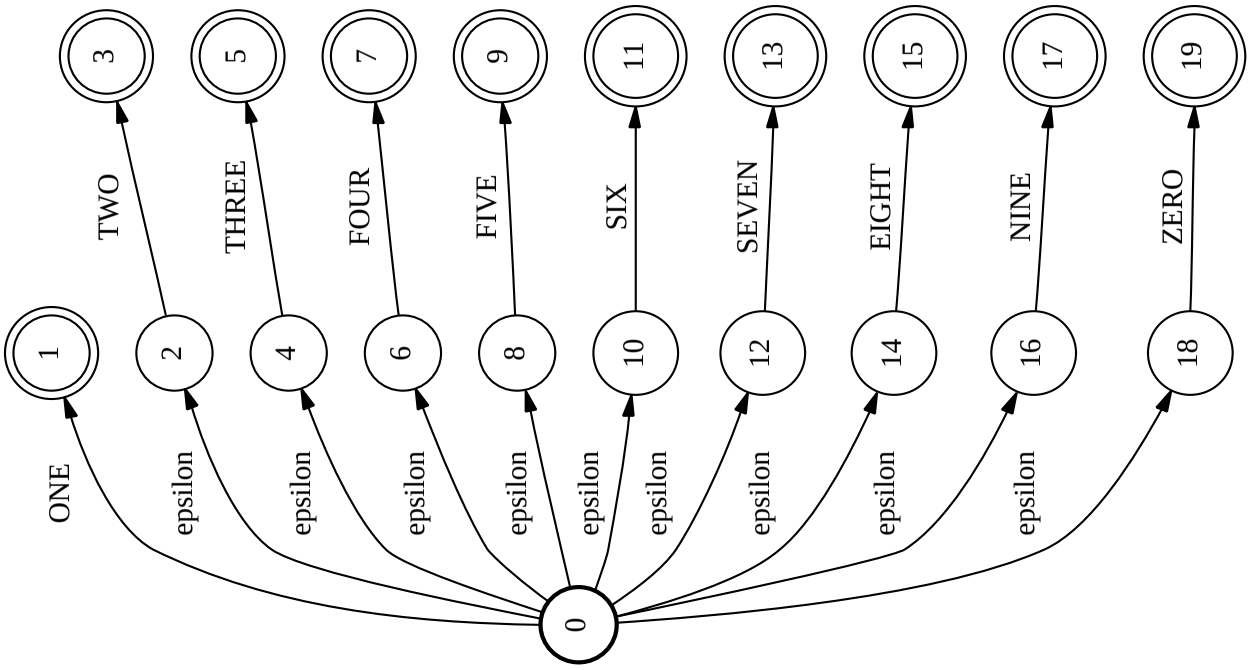


digits.fsm

I just typed in this FSM, but another way of constructing it is to consider it as a union of an FSM that accepts the word "ONE" with the FSM that accepts the word "TWO" and another that accepts "THREE" and so on. I've included a simple script that can generate *linear* FSMs, called `make_linear_fsm.pl`. To generate the digits fsm by forming the union of other FSMs, do this:

```
prompt> foreach i ( ONE TWO THREE FOUR FIVE SIX SEVEN EIGHT NINE ZERO )
foreach? make_linear_fsm.pl $i > $i.stxt
foreach? fsmcompile -i digits.alpha $i.stxt > $i.fsm
foreach? end
prompt> fsmunion ONE.fsm TWO.fsm THREE.fsm FOUR.fsm FIVE.fsm SIX.fsm \
SEVEN.fsm EIGHT.fsm NINE.fsm ZERO.fsm > digits2.fsm
```

The resulting FSM doesn't look quite the same:



digits2.fsm

However, you can probably tell that they are the same, and, in fact, you can determine that with `fsmequiv`, after removing the epsilon transitions and determinizing:

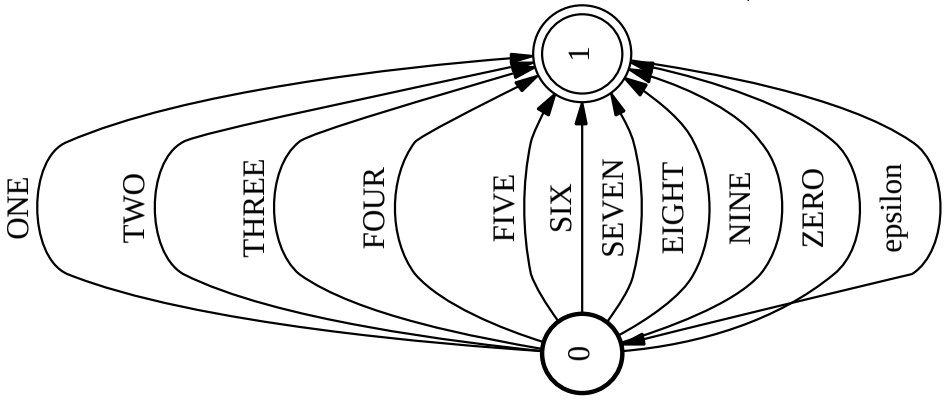
```
prompt> fsmrmepsilon digits2.fsm | fsmdeterminize | fsmequiv digits.fsm -
prompt> echo $?
0
```

`fsmequiv` will return a status of zero if the two FSMs are equivlaent, one otherwise.

Of course, you might want to have a string of digits, rather than just one. We can take the Kleene closure of the digits FSM to get a new FSM:

```
prompt> fsmclosure -p digits.fsm > digits+.fsm
```

This produces a new FSM that will accept one or more digits (note the epsilon from state 1 to state 0):

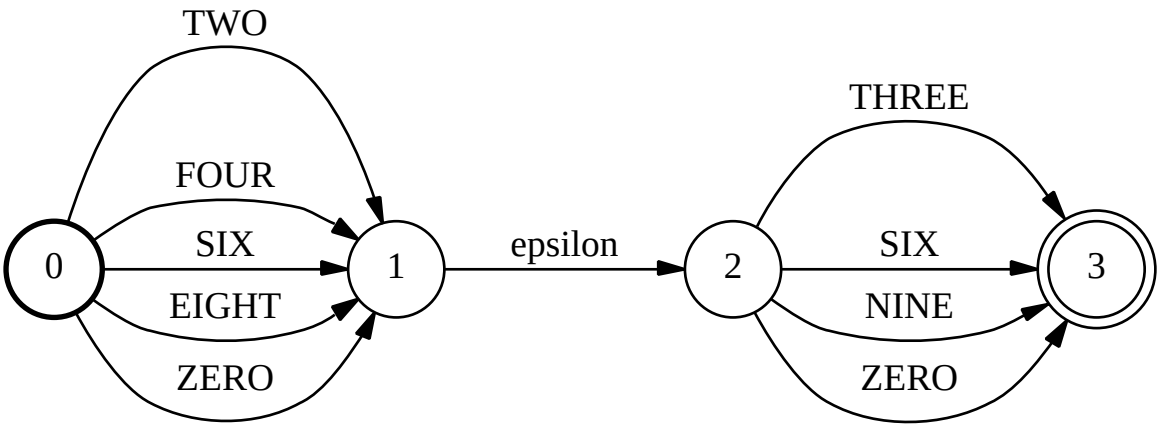


digits+.fsm

This type of Kleene closure is called the "Kleene plus", which means that there are 1 or more repetitions of the FSM. If you want zero or more repetitions (the "Kleene star"), then you just leave out the "-p" arguement in `fsmclosure`.

Let's now use two subsets of the digits: the even numbers, and the multiples of three. These are in the files `even.stxt` and `threes.stxt`. If, for some reason, we would want to build a FSM that accepted an even number followed by a multiple of three (perhaps as a code), we could build the FSM by concatenating the two FSMs together (using `fsmconcat`):

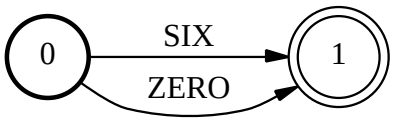
```
prompt> fsmconcat even.fsm threes.fsm > code.fsm
```



code.fsm

On a more interesting note, we can ask the question: what digits are even *and* a multiple of three? Of course, we know that the answers are zero and six, but we can prove that by *intersecting* the even and threes FSMs:

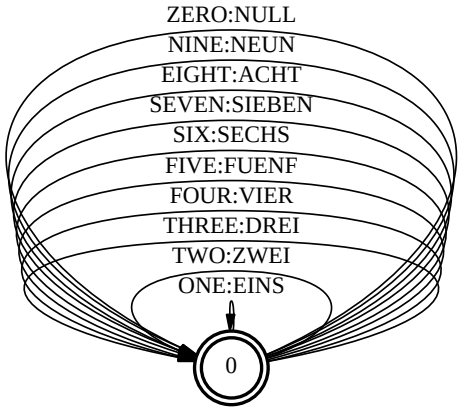
```
prompt> fsmintersect even.fsm threes.fsm > eventhrees.fsm
```



eventhrees.fsm

Intersection only allows paths that appear in *both* input FSMs to be present in the output FSM.

Another useful concept is *composition*, which is an operation defined on transducers. If we wanted to convert English digits to German digits, for example, we might have the following transducer:




```
0 0 ONE EINS
0 0 TWO ZWEI
0 0 THREE DREI
0 0 FOUR VIER
0 0 FIVE FUENF
0 0 SIX SECHS
0 0 SEVEN SIEBEN
0 0 EIGHT ACHT
0 0 NINE NEUN
0 0 ZERO NULL
0
```

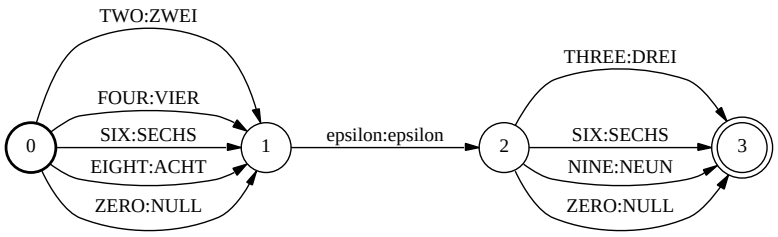
Eng2German.fsm

To compile this, we need to have a separate *output* alphabet (consisting of German numbers) from the input alphabet. `fsmcompile` takes the "-t" argument to indicate that the FSM is a transducer:

```
prompt> fsmcompile -t -i digits.alpha -o germandigits.alpha Eng2German.stxt > Eng2German.fsm
```

We can now compose this with, say, our code FSM to produce a German equivalent. The composition operation is a lot like intersection, except that we only match strings on the *input* side of the transducer:

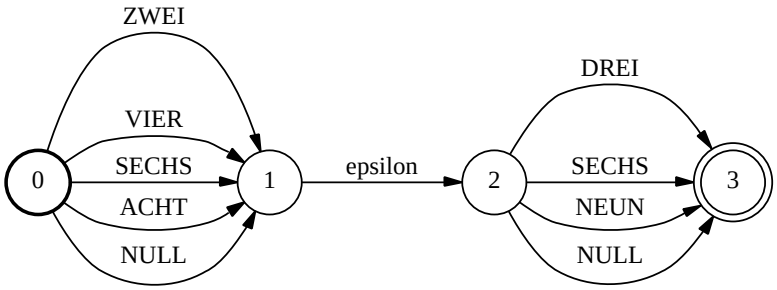
```
prompt> fsmcompose code.fsm Eng2German.fsm > codeGerman.fsm
```



codeGerman.fsm

Notice that the output of the composition operation is a transducer. If we only care about the German portion of the output, we can *project* to the output symbols using `fsmproject`:

```
prompt> fsmproject -o codeGerman.fsm > codeGermanOnly.fsm
```



codeGermanOnly.fsm

Here are some problems to try out on your own:

1. **The happy language**
   (a) How would you represent the string `There's a happy boy` ?
   (b) In English, we can modify (some) adjectives with the word "very". Therefore, we can imagine the following possible extensions to the above sentence

   ```
   There's a happy boy
   There's a very happy boy
   There's a very very happy boy
   ```

   ⋮

   Can you represent this with an FSM?
   (c) How would you allow girl, dog, rabbit, etc. instead of boy?
2. **Humans only, please!** What operation would you use to remove rabbits, dogs, and other non-humans from the above FSM, while still producing complete sentences?
3. **Change of emotion:** Can you define a transducer that will change:
   (a) happy to sad?
   (b) happy to either happy or sad?
4. How would you design a pronunciation dictionary for this small language? *Hint: you'll need to convert words to phones, and there's more phones than words.*