CS187 Assignment 4

Finite-State Transducers

Lucas Freitas

March 6, 2013

Problem 1

For all the items in the next pages, I used a Python script to generate a text file, which I then converted to `FST` format using:

```
$fstcompile --isymbols=test.sym --osymbols=test.sym --keep_isymbols --keep_osymbols
--acceptor file.txt file.fst
```

Where `file` is the name of each of the `FST` or text files. I then generated the PDF images of the automatons using the command:

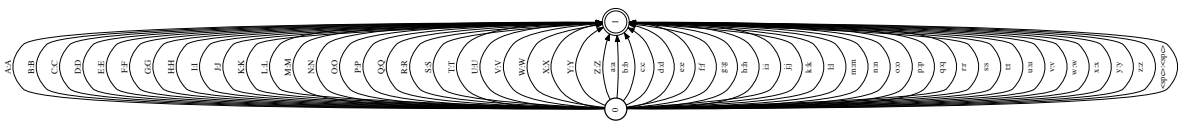
```
$ fstdraw --isymbols=test.sym --osymbols=test.sym file.fst | dot -Tpdf > file.pdf
```

(1) Accept any letter in L (including space)

To accept any letter in L, including space, we just need to create a Python script that outputs lines containing `0 1 letter` for each `letter` in L, so that any letter takes us to state `1`, which we will be our final state.

```
# accept all alphabetical letters
for letter in string.ascii_uppercase + string.ascii_lowercase:
    file.write("0 1 %c\n" %letter)

# accept space
file.write("0 1 <spc>\n")
file.write("1")
```

After compiling the text file (`p1a.txt`) with Open-FST, we generate the automaton below:



Problem 1.1 (Continued)

Testing our automaton using `test-string.py`, we notice that the automaton seems to work properly (notice that the blank space below in the second test is actually a `<spc>`):

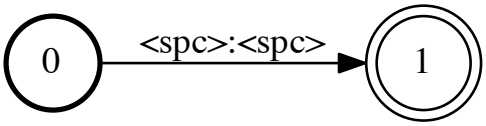
```
Please type a string to test: a
String is accepted
a

Please type a string to test:
String is accepted


Please type a string to test: z
String is accepted
z

Please type a string to test: this shouldnt work
String is rejected
```

(2) Accept a single space

This automaton is even simpler. We just have to do the same as below, but only accepting the `<spc>` letter instead of all possible letters in L.

```
# accept space
file.write("0 1 <spc>\n")
file.write("1")
```

After compiling the text file (`p1b.txt`) with Open-FST, we generate the automaton below (a blank area of the PDF was cropped for better visualization):



Testing our automaton using `test-string.py`, we notice that the automaton seems to work properly (notice that the blank space below in the first test is actually a `<spc>`):

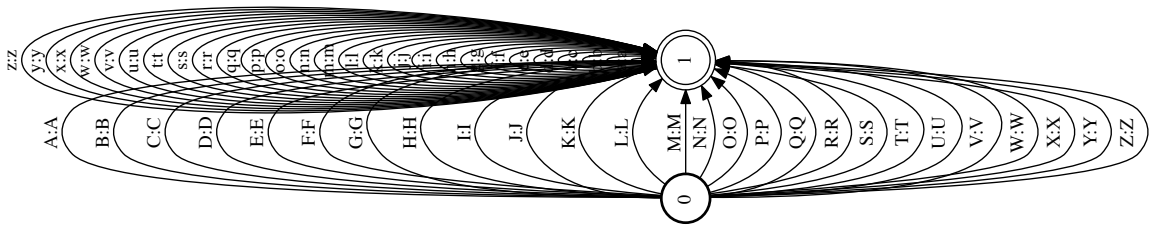
```
Please type a string to test:
String is accepted


Please type a string to test: z
String is rejected

Please type a string to test: p
String is rejected

Please type a string to test: this shouldnt work
String is rejected
```

(3) Accept a capitalized word

To accept a capitalized word, state `0` can only accept capital letters (`<spc>` not included), and should direct those to state `1`. No other letter should be accepted in state `0`. State `1` (the final state), on the other hand, should accept any lowercase letter (`<spc>` not included, as a word is defined in the specification as not having any space characters). The Python code to achieve that would be:

```
# accept first uppercase letter
    for letter in string.ascii_uppercase:
        file.write("0 1 %c\n" %letter)

# accepts following letters as lowercase
    for letter in string.ascii_lowercase:
        file.write("1 1 %c\n" %letter)
file.write("1")
```

After compiling the text file (`p1c.txt`) with Open-FST, we generate the automaton below (a blank area of the PDF was cropped for better visualization):



Testing our automaton using `test-string.py`, we notice that the automaton seems to work properly:

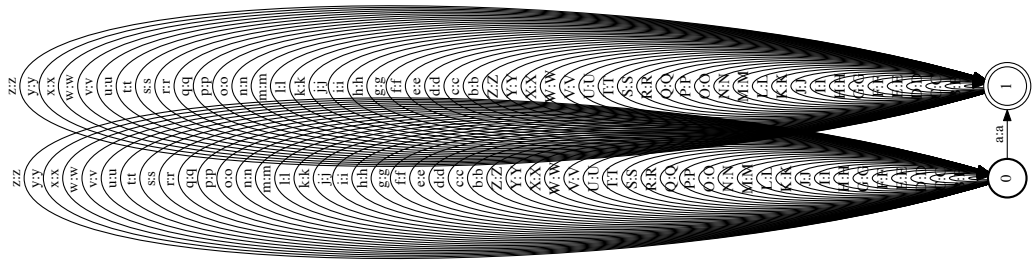
```
Please type a string to test: Schieber
String is accepted
Schieber

Please type a string to test: Csoneeightysevenisawesome
String is accepted
Csoneeightysevenisawesome

Please type a string to test: lucas
String is rejected

Please type a string to test: Lucas Freitas
String is rejected

Please type a string to test: turinG
String is rejected
```

(4) Accept a word containing the letter `a`

To accept a word containing `a`, just accept any letter (except for `<spc>`) in state `0`, directing all letters back to state `0` except for `a`, which should be directed to the final state `1`. State `1` would then accept any letter except for `<spc>`. The Python code to achieve that would be:

```
# loop non-a letters back to their current state
for letter in string.ascii_uppercase[1:] + string.ascii_lowercase[1:]:
    file.write("1 1 %c\n" %letter)
    file.write("0 0 %c\n" %letter)

# move a from state 0 to 1
file.write("0 1 a\n")
file.write("1")
file.write("1")
```

After compiling the text file (`p1d.txt`) with Open-FST, we generate the automaton below (a blank area of the PDF was cropped for better visualization):



Testing our automaton using `test-string.py`, we notice that the automaton seems to work properly:

```
Please type a string to test: lucas
String is accepted
lucas

Please type a string to test: lucAs
String is rejected

Please type a string to test: lucas freitas
String is rejected

Please type a string to test: csoneeightysevenisawesome
String is accepted
csoneeightysevenisawesome

Please type a string to test: csonesixtyonetoo
String is rejected
```

Problem 2

For all the items in the next pages, I used `FST` operations on the automatons from Problem 1 to generate a final automaton that achieves the function requested in the item. I then generated the PDF images of the automatons using the command:

```
$ fstdraw --isymbols=test.sym --osymbols=test.sym file.fst | dot -Tpdf > file.pdf
```

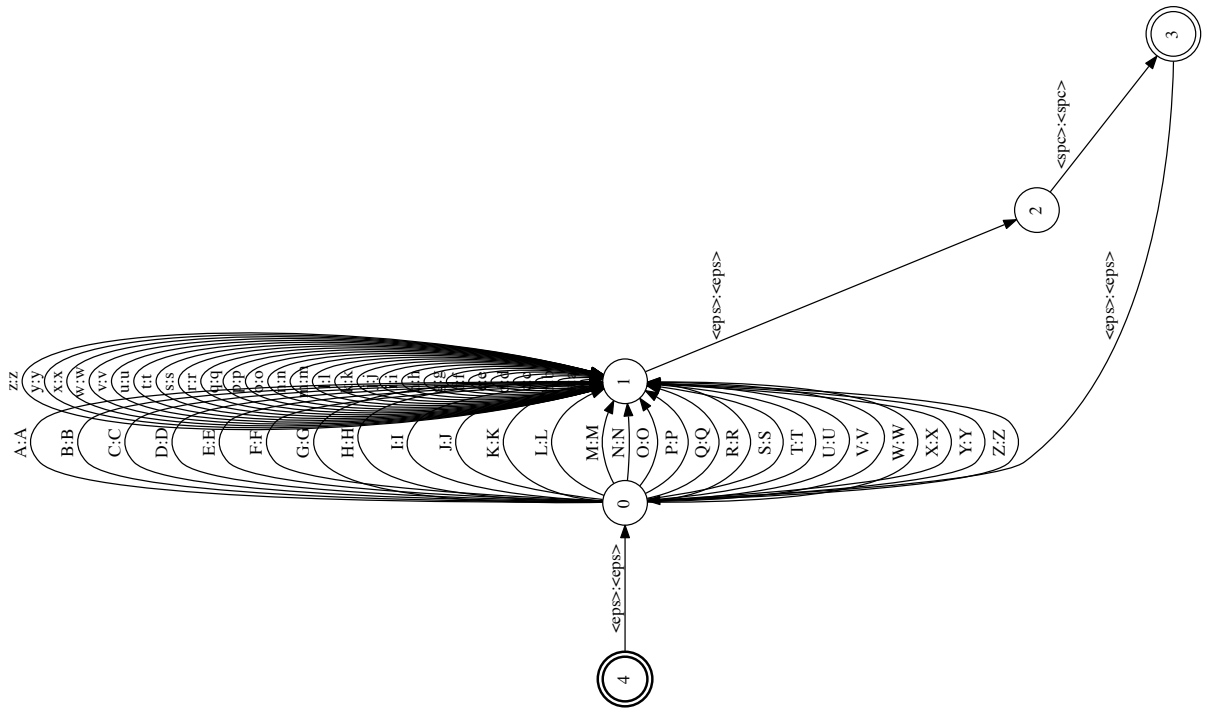
(1) Accept zero or more capitalized words followed by spaces

For this case, we just need to accept both capitalized words or spaces. As said in Piazza, every accepted string should include a space as its last character, so the automaton should accept `Lucas<spc>Freitas<spc>`, but not `Lucas<spc>Freitas`.

To achieve that, we first need to concatenate the automatons from items `(3)` and `(2)` above, so that the resulting automaton will accept strings of the form `Word<spc>`, where `Word` is a capitalized string. Finally, we take the closure of the automaton, which will get us to accept a sequence of zero or more `Word<spc>`, i.e., capitalized words followed by spaces. The list of Open-FST operations to generate the final automaton is:

```
$ fstconcat p1c.fst p1b.fst > p2a_concat.fst
$ fstclosure p2a_concat.fst > p2a.fst
```

The generated automaton can be seen below (a blank area of the PDF was cropped for better visualization):



Problem 2.1 (Continued)

Testing our automaton using `test-string.py`, we notice that the automaton seems to work properly (notice that in all examples besides the first, the string is ended with a space):

```
Please type a string to test: No Space At The End
String is rejected

Please type a string to test: Non capital Letter Somewhere
String is rejected

Please type a string to test: Lucas
String is accepted
<eps>Lucas<eps>

Please type a string to test: I Like This Class
String is accepted
<eps>I<eps> <eps>Like<eps> <eps>This<eps> <eps>Class<eps>
```

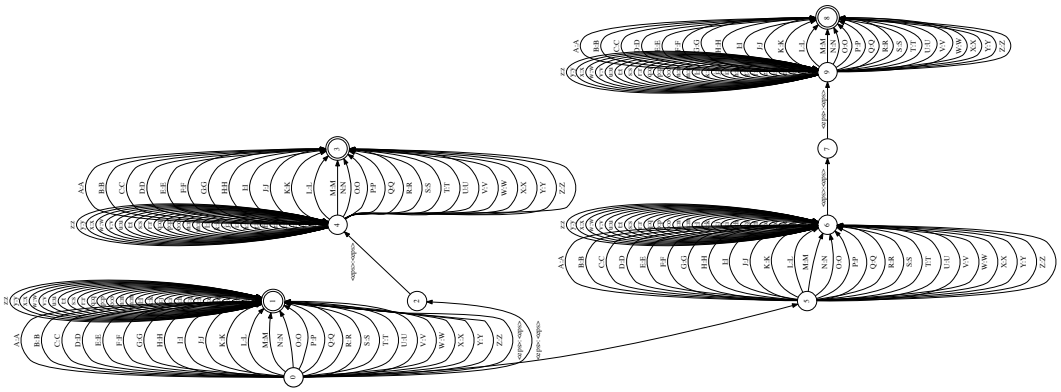
(2) Accept a word beginning or ending in a capitalized letter

As noted in Piazza, words might have capitalized letters at its beginning or end, but not in its middle. Therefore, we just have to realize that our final automaton should accept words that only start in a capitalized letter (capitalized word automaton from item `(3)` of Problem 1), words that only end in a capitalized letter (`fstreverse` of the capitalized word automaton), or both starts and ends with capitalized letter (`fstconcat` of the two previous automatons).

To achieve that, we just need to take the `fstunion` of the three automatons (take the `fstunion` of the first two, and then take the `fstunion` of the resulting automaton and of the third), since we want to accept any of the three kinds of strings. The list of Open-FST operations to generate the final automaton is:

```
$ fstreverse p1c.fst > p2b_reverse.fst
$ fstconcat p1c.fst p2b_reverse.fst > p2b_concat.fst
$ fstunion p1c.fst p2b_reverse.fst > p2b_temp.fst
$ fstunion p2b_temp.fst p2b_concat.fst > p2b.fst
```

The generated automaton can be seen below (a blank area of the PDF was cropped for better visualization):



Problem 2.2 (Continued)

Testing our automaton using `test-string.py`, we notice that the automaton seems to work properly (notice that for all examples besides the first, a space was added at the end of the string):

```
Please type a string to test: Chinese
String is accepted
Chinese

Please type a string to test: persiaN
String is accepted
<eps><eps>persiaN

Please type a string to test: JapanesE
String is accepted
<eps>J<eps><eps>apanesE

Please type a string to test: PorTuguesE
String is rejected
```

Note (additional automaton)

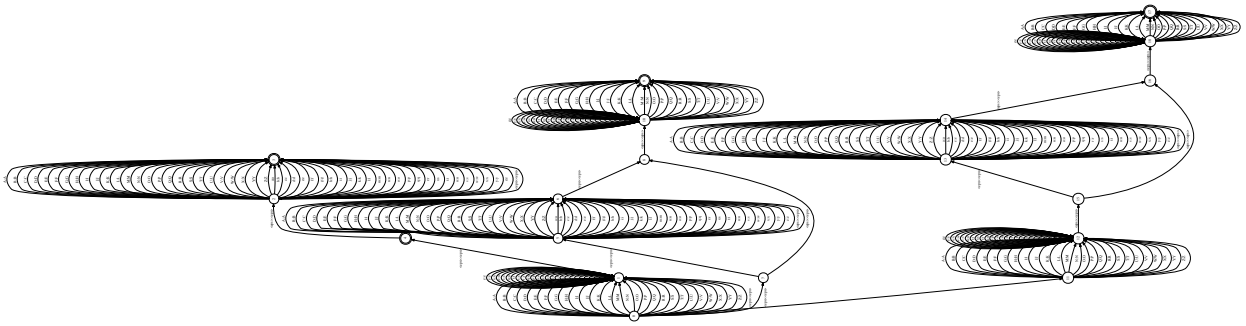
If we want to also accept words that begin or end in a capitalized letter, and might also have capitalized letter in their middle, one thing that we can is generate an automaton (let's call is `A`) that represents a generic word with any letters in L (`<spc>` not included), and make concatenations of `A` and the capitalized word automaton, the reverse capitalized word automaton and `A`, and the capitalized word automaton, `A`, and the reverse capitalized word automaton.

Taking the union of the three concatenations will generate the automaton we are looking for. To generate `A`, we just need to take the difference of the automatons in items `(1)` and `(2)` from Problem 1, and then take the `fstclosure` of the generated automaton. The list of Open-FST operations to generate the final automaton is:

```
$ fstdifference p1a.fst p1b.fst > p2b_letters.fst
$ fstclosure p2b_letters.fst > p2b_words.fst
$ fstreverse p1c.fst > p1c_reverse.fst
$ fstconcat p1c.fst p2b_words.fst > p2b_begincap.fst
$ fstconcat p2b_words.fst p1c_reverse.fst > p2b_endcap.fst
$ fstconcat p2b_begincap.fst p1c_reverse.fst > p2b_bothcap.fst
$ fstunion p2b_begincap.fst p2b_endcap.fst > p2b_tempunion.fst
$ fstunion p2b_tempunion.fst p2b_bothcap.fst > p2b_note.fst
```

Problem 2.2 (Continued)

The generated automaton can be seen below (a blank area of the PDF was cropped for better visualization):



Testing our automaton using `test-string.py`, we notice that the automaton seems to work properly (notice that for all examples besides the first, a space was added at the end of the string):

```
Please type a string to test: Lucas
String is accepted
L<eps><eps>u<eps>c<eps>a<eps>s

Please type a string to test: lucaD
String is accepted
<eps><eps><eps>lucaD

Please type a string to test: LucaS
String is accepted
<eps>L<eps><eps><eps>ucaS

Please type a string to test: LuCaS
String is accepted
<eps>L<eps><eps>u<eps>C<eps><eps>aS

Please type a string to test: lucAs
String is rejected

Please type a string to test: lucas
String is rejected
```

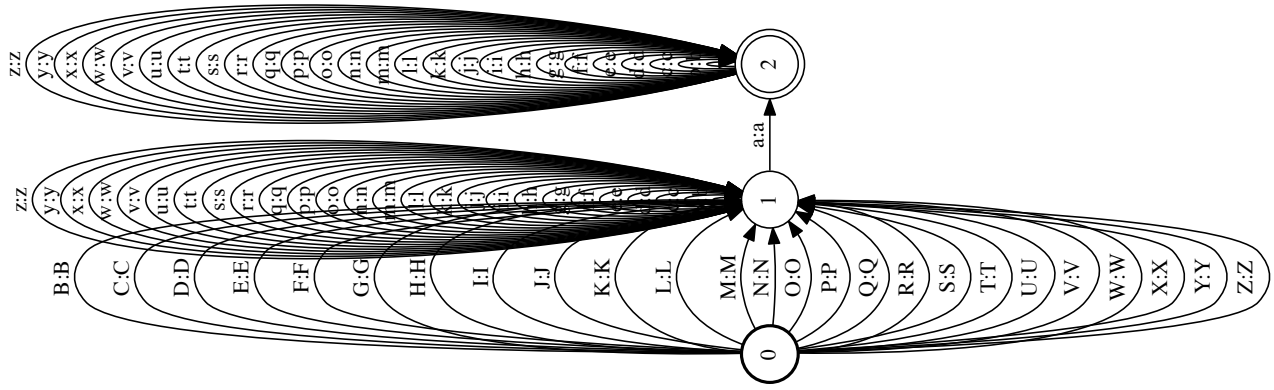
(3) Accept a word that is capitalized and contains the letter `a`

For this item, we just need to take the intersection of the automatons from items `(3)` and `(4)` in Problem 1, since we want the word to be both capitalized and to contain the letter `a`. The Open-FST operation to generate the final automaton is:

```
$ fstintersect p1c.fst p1d.fst > p2c.fst
```

Problem 2.3 (Continued)

The generated automaton can be seen below (a blank area of the PDF was cropped for better visualization):



Testing our automaton using `test-string.py`, we notice that the automaton seems to work properly (notice that for all examples besides the first, a space was added at the end of the string):

```
Please type a string to test: Lucas
String is accepted
Lucas

Please type a string to test: Scheiber
String is rejected

Please type a string to test: allen
String is rejected

Please type a string to test: Harry Potter
String is rejected
```

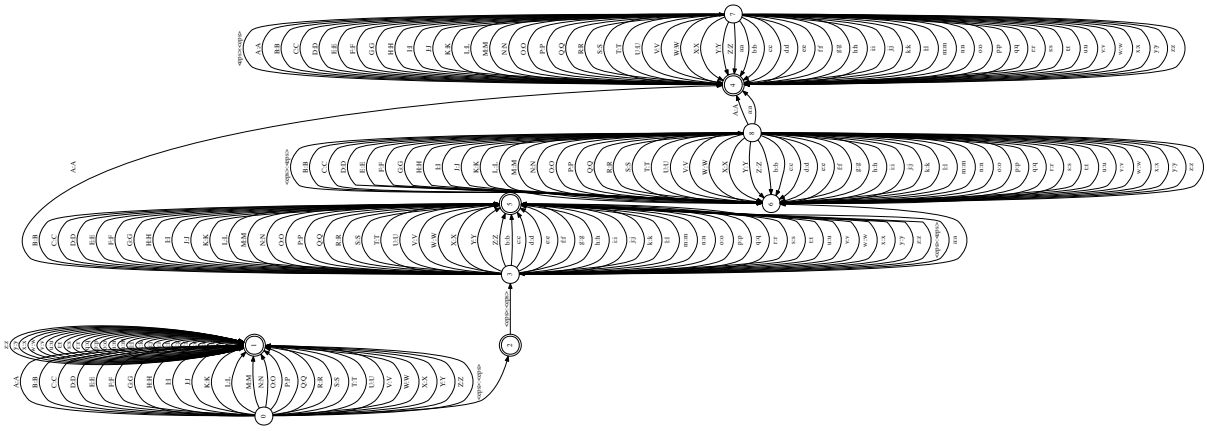
(4) Accept a word that is capitalized or does not contain an `a`

For this last item, we just need to take the union of the automaton from item `(3)` of Problem 1 and the negation of item `(4)` of Problem 1, which can be calculated from the `fstdifference` of all the words with letters in L (`<spc>` not included) and item `(4)`, since we want the word to be capitalized or to not contain the letter `a` (negation of containing letter `a`). The OpenFST operation to generate the final automaton is (notice that we had to sort arcs to fulfill the requirements of `fstdifference`):

```
$ fstdifference p1a.fst p1b.fst > p2b_letters.fst
$ fstclosure p2b_letters.fst > p2b_words.fst
$ fstarcsort p1d.fst > sorted_p1d.fst
$ fstdifference p2b_words.fst sorted_p1d.fst > not_a.fst
$ fstunion p1c.fst not_a.fst > p2d.fst
```

Problem 2.4 (Continued)

The generated automaton can be seen below (a blank area of the PDF was cropped for better visualization):



Testing our automaton using `test-string.py`, we notice that the automaton seems to work properly (notice that for all examples besides the first, a space was added at the end of the string):

```
Please type a string to test: Lucas
String is accepted
Lucas

Please type a string to test: Schieber
String is accepted
<eps><eps>S<eps>c<eps>h<eps>i<eps>e<eps>b<eps>e<eps>r

Please type a string to test: lucas
String is rejected

Please type a string to test: schieber
String is accepted
<eps><eps>s<eps>c<eps>h<eps>i<eps>e<eps>b<eps>e<eps>r
```

(5) Solve the previous item without using `fstunion`

Accepting a word that is capitalized or does not contain an `a` is the same as accepting all words with any letters in L (except for `<spc>`) except for those that are both not capitalized and contain the letter `a`.

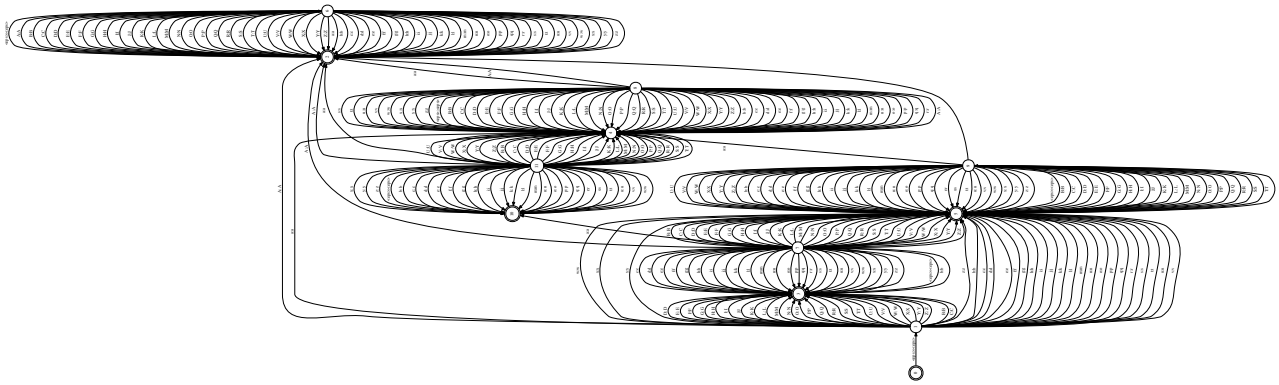
To accept words that are both uncapitalized and contain the letter `a`, first we have to take the `fstdifference` of all possible words and capitalized words (generating automaton `X`, which accepts uncapitalized words), and then take the `fstdifference` of all possible words and all words containing `a` (generating automaton `Y`, which accepts words without `a`).

We then take the `fstdifference` of `X` and `Y`, generating an automaton `Z` that accepts uncapitalized words that contain the letter `a`. Finally, we take the `fstdifference` of all possible words and `Z`, and our desired automaton should be generated.

Luckily enough, we have already generated an automaton for all possible words in the note for item 2, so the list of Open-FST operations we need to perform is:

```
$ fstdifference p1a.fst p1b.fst > p2b_letters.fst
$ fstclosure p2b_letters.fst > p2b_words.fst
$ fstdifference p2b_words.fst p1c.fst > uncap.fst
$ fstarcsort p1d.fst > sorted_p1d.fst
$ fstdifference p2b_words.fst sorted_p1d.fst > not_a.fst
$ fstrmepsilon not_a.fst > no_epsilon_or_a.fst
$ fstarcsort no_epsilon_or_a.fst > sorted_no_a.fst
$ fstdifference uncap.fst sorted_no_a.fst > almost_there.fst
$ fstrmepsilon almost_there.fst > no_epsilon_almost_there.fst
$ fstarcsort no_epsilon_almost_there.fst > arcsort_there.fst
$ fstdifference p2b_words.fst arcsort_there.fst > p2e.fst
```

Notice that we had to remove epsilons and sort arcs to fulfill the requirement of some of the methods. Let's see what the automaton looks like! The generated (and super cool) automaton can be seen below (a blank area of the PDF was cropped for better visualization):



Testing our automaton using `test-string.py`, we notice that the automaton seems to work properly (notice that for all examples besides the first, a space was added at the end of the string):

```
Please type a string to test: Lucas
String is accepted
<eps>L<eps>u<eps>c<eps>a<eps>s

Please type a string to test: Schieber
String is accepted
<eps>S<eps>c<eps>h<eps>i<eps>e<eps>b<eps>e<eps>r
```

Problem 2.5 (Continued)

```
Please type a string to test: lucas
String is rejected

Please type a string to test: schieber
String is accepted
<eps>s<eps>c<eps>h<eps>i<eps>e<eps>b<eps>e<eps>r
```

Problem 3

(1) How many states cat be reached from the initial state?

Only two states can be reached: `1` and `2`.

(2) How many states can reach a final state?

States `0`, `3`, and `4` can reach a final state. As an example, the state `0` can reach the final state `2` by getting `2` as input, while the state `3` can reach the final state `4` if it gets the input `4`. Finally, the final state `4` can reach the final state `4` (itself) by receiving the string `34` as input.

(3) Compile the automaton and use OpenFTS to remove all useless states

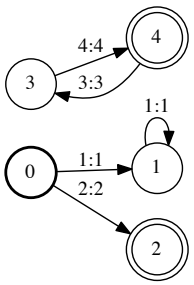
We can compile the automaton by creating a `.sym` file for `isymbols` and `osymbols` as the one below (the file can be found in the submission folder as `p3.sym`):

```
1 1
2 2
3 3
4 4
```

Using the automaton given and running the commands below, we obtain the following automaton:

```
$ fstcompile --isymbols=p3.sym --osymbols=p3.sym --keep_isymbols --keep_osymbols
--acceptor p3.txt p3.fst
```

```
$ fstdraw --isymbols=test.sym --osymbols=test.sym file.fst | dot -Tpdf > file.pdf
```



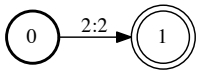
We can remove the useless states by running:

```
$ fstconnect p3.fst p3_new.fst
```

And then draw the new automaton by running:

```
$ fstdraw --isymbols=p3.sym --osymbols=p3.sym p3_new.fst | dot -Tpdf > p3_new.pdf
```

The resultant automaton can be seen below:



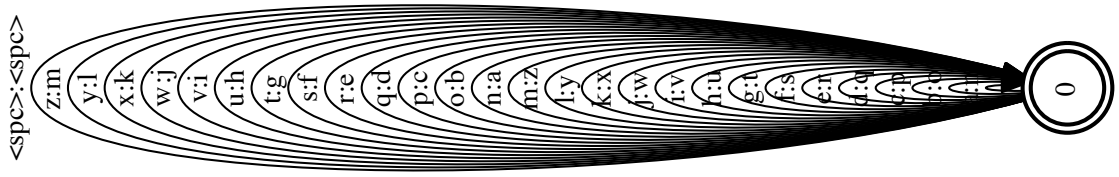
Problem 4

(1) Create a transducer that implements the `rot13` cipher

To implement the `rot13` cipher, all we have to do is rotate lowercase letters 13 positions in the alphabet, while leaving the space unchanged. That way, the transducer will only have one state, and will accept any lowercase letter or space. The text file (`p4.txt`) can be generated from the Python script below:

```
letters = string.ascii_lowercase

# encode message
for i in range(26):
    file.write("0 0 %c %c\n" %(letters[i], letters[(i + 13) % 26]))

# space encodes to space
file.write("0 0 <spc> <spc>\n")
file.write("0")
```

After compiling the text file (`p4.txt`) with Open-FST, we generate the automaton below (a blank area of the PDF was cropped for better visualization):



(2) Encode and decode the message given

To encode and decode the message, we can just run `$ python test-string.py` inputting each of the messages, as in the items below:

- Encode `how many yale students does it take to change a lightbulb` (the encoded message is the last line of output):

  ```
  Please type a string to test: how many yale students does it take to change a lightbulb
  String is accepted

  ubj znal lnyr fghqragf qbrf vg gnxr gb punatr n yvtugohyo
  ```

- Decode `abar arj unira ybbxf orggre va gur qnex` (the decoded message is the last line of output):

  ```
  Please type a string to test: abar arj unira ybbxf orggre va gur qnex
  String is accepted

  none new haven looks better in the dark
  ```