

CMPE 561
NATURAL LANGUAGE PROCESSING
APPLICATION PROJECT #1

Students

Ali Yeşilkanat 2017700159

Çağatay Şahin 2015700192

Submission Date

14.11.2017

Introduction

Morphology is the study of words, how they are formed, and their relationship to other words in the same language. It analyzes the structure of words and parts of words, such as stems, root words, prefixes, and suffixes. Morphology also looks at parts of speech, intonation and stress, and the ways context can change a word's pronunciation and meaning.

Morphological parsing, in natural language processing, is the process of determining the morphemes from which a given word is constructed. The generally accepted approach to morphological parsing is through the use of a finite state transducer (FST), which inputs words and outputs their stem and modifiers. The FST is initially created through algorithmic parsing of some word source, such as a dictionary, complete with modifier markups.

A morphological parser must be able to analyze and parse rules for both morphotactics and orthographic rules. Morphotactics represent the ordering restrictions in place on the ordering of morphemes. Etymologically, it can be translated as "the set of rules that define how morphemes (morpho) can touch (tactics) each other". Orthographic rules represent set of conventions for writing a language. It includes norms of spelling, hyphenation, capitalization, word breaks, emphasis, and punctuation.

One of the common ways to designing a morphological analyzer is two level paradigm. Each level separates orthographic rules and morphotactics, allows us implement independent and individually coherent set of rules.

For this project, 4 different programs are created. 2 of them are for converting a given string in lexical form to intermediate form and inverse. Other 2 of them are for converting a given string in intermediate form to surface form and lexicon format to surface form. The last one, which is lexicon to surface converter uses lexicon-intermediate and intermediate-surface converters as middle converter in respect to two level morphology paradigm.

Program Design & Implementation

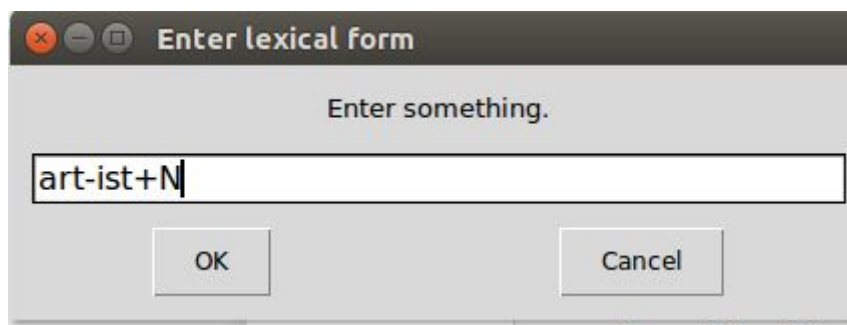
The application, is developed on Ubuntu environment using both Python 2.7 language and OpenFST bash commands. As Python IDE, Jupyter Notebook is used. For compiling FSTs `fstcompile` bash command is used, but for composing (running,transducing or executing) inputs and reverse operations, official Python Wrapper for OpenFST, `Pywrapfst` is used.

Also, for creating FST readable input files(AT&T FST Format) and symbol tables, some Python scripts are written. For user interface, Easy Gui library for Python is chosen. This section explains how the user communicates with the program.

For easily examining the results of two-level morphology paradigm, 4 different executables are created. This reason will be explained better in the Program Structure section.

Program Execution & Interface

All 4 programs have the same interface design which is just about a textbox and buttons. User is expected to enter form suitable to the program which he/she is executed. For example, lexical-intermediate form converter has this interface.



After entering input and pressing OK button, user may see the result of analysis.

In this example, given input **art-ist+N** is valid input (because **art** is stored in lexicon and **-ist** suffix, which is also stored in morphotactics, creates a noun. Expected output may be seen on below.



If multiple analysis' are valid, then output will have both results. For example intermediate-lexicon conversion, assuming input as **be#**, output may be seen on below.



It can be seen that, be# is parsed in different transducers, which one of them is noun and the other one is auxiliary verb FSTs. User may re-run the application and enter new inputs for analysis.

Input and Output

Each program will only show an output if given string a valid (matched in lexicon-morphotactics-morphophonemics) otherwise, they will show an error. On below each pair of converters explained.

- **Lexical - Intermediate Converters:**

The input string for this program is expected to be in lexical format. User is at least expected to give stem and pos tag of the final word. User may also add suffixes for stem, in that case given pos tag should represent the final form of the word (stem+suffixes)

For example, for creating “**artistic**” adjective, user is expected “**art-ist-ic+ADJ**”

Output of this converter, will be stem-suffixes(optional, and each of them separated with ‘^’(morpheme boundary)-part of speech tag-#(end of the word tag).

For previous example, output would be: “**art^ist^ic#**”

- **Intermediate-Morphophonemics Converters:**

The other pair of this converter to this job in reverse, by expecting an input string as in intermediate form and converts it into lexical form.

The format of the input and output will be explained in this section. If the program also gets some input from input files and produces some output on output files, then the exact format of the files must also be given. It is advisable to include some example inputs and outputs, and their explanations.

Program Structure

There are 3 main structures of the program. They are, lexicon & morphotactics, morphophonemic rules and UI & Analyzer and each of them explained separately on below.

1) Lexicon & Morphotactics: Lexicon is separated with respect to part of speech tags of the stems. These categories are auxiliary verbs, pronouns, propositions, nouns, adverbs, adjectives and nouns. Morphotactics also stored with lexicon in a special format.

1.1) Storage: For each stem in different category, some possible (most of) and suitable suffixes are gathered and stored in text files in a special format like LISP language. This storage format is like this:

stem (-suffix:+Pos of suffix1|-suffix2:+Pos of suffix2(-suffix3:+Pos of suffix3)|...)

With this representation, each possible suffix sequence is stored. By this it means that, some suffixes may append only after some suffixes. Pos of suffixes are stored also too, because of understanding the new pos tag of the word (may also some appended suffixes) would change after this new suffix addition.

For example, **accident** stem stored in noun lexicon ,because of having noun pos, have this representation:

accident (-al:+ADJ(-ism:+N|-ist:+N|-ity:+N|-ly:+ADV))

With representation, possible morphological derivations are,

accidental+ADJ, accidentalism+N, accidentalist+N, accidentality+N, accidentally+ADV

Plus, we also understand that accident+N is a possible form because of storing the stem in this category.

However, this representation is not enough to store irregular stems. Irregular stems are only found in noun and verb categories. In noun category, irregularity means irregular plural forms of nouns. And verb category, irregularity means irregular past forms of verbs. So, in this representation irregular forms of stems separated and stored with '/' character.

For example, **person** noun, has irregular plural form **people**, and they both stored as follows,

person/people

(-able:+ADJ(-ness:+N|-y:+ADV)|-al:+ADJ(-ity:+N|-ism:+N|-ly:+ADV)|-age:+N|-ate:+V|-ify:+V(-ation:+N|-ed:+ADJ)|-ize:+V(-ation:+N))

write verb, has irregular past forms as **wrote**, **written**, and these forms are stored as follow, **write/wrote/written (-ing:+N|-er:+N|-able:+ADJ)**

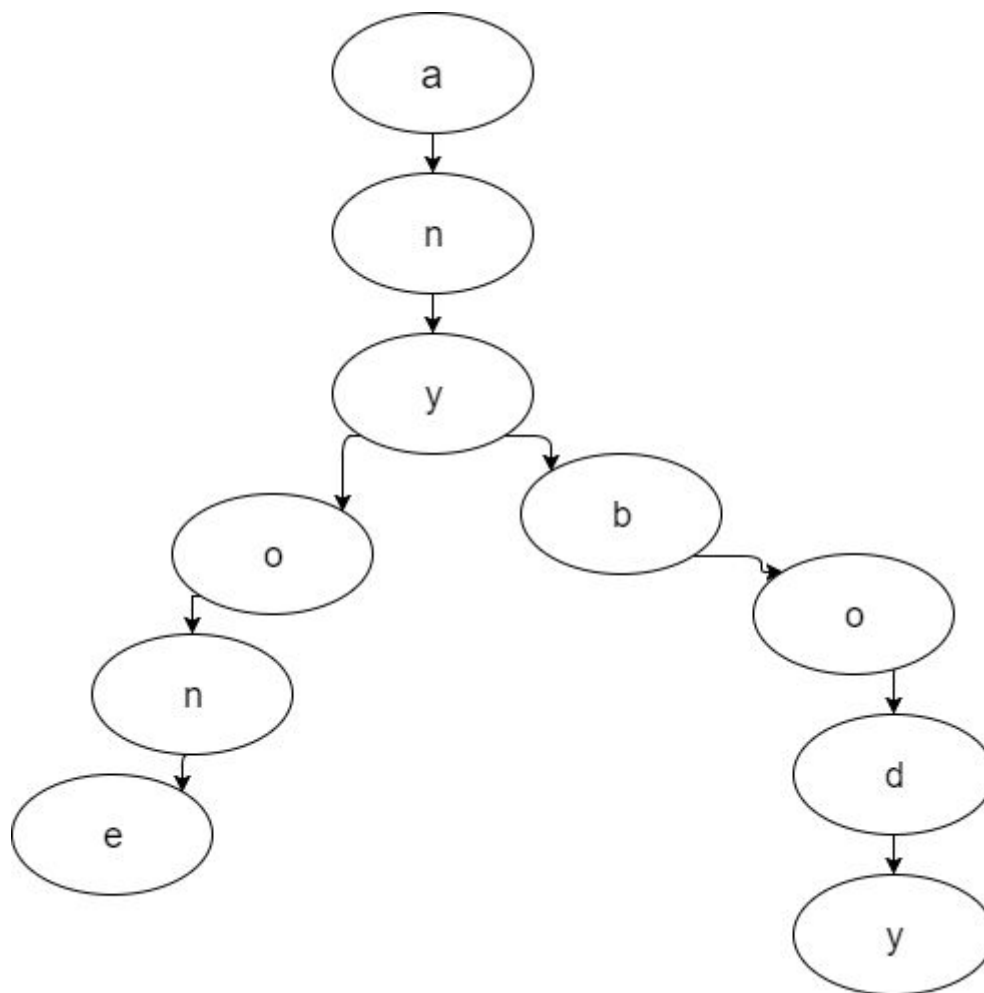
As result, all morphotactics of the stems are stored in this project are defined and ready to transformation to FSTs. It can be seen that, morphotactics are defined and written independently from programming language or any transducer library by storing them in a text file.

1.2) Parsing Morphotactics and Transform to Tree: As it can be seen from the previous chapter, defined morphotactics are independent from any FST library. But from this point, process is getting more dependent and some transformations are needed to convert this information to a suitable type which FST can understand.

For each defined category in previous part (nouns, verbs, etc.) a morphotactics parser and a tree is implemented. Tree is used because of efficiency and helps us to transform morphotactics. It is designed to store stems and their suffixes. Each character in stem, is a node in tree.

While parsing lexicon, texts stems and suffixes parsed differently. Stems can be seen easily that are lying down till the first space on the line. Parser traverses character iteratively from left to right and added to the tree. But the case is, having same preceding characters on different stems are represented as the same nodes in the tree. So, node count is minimized and in fst generation phase, prevents creating unnecessary states.

For example, in lexicon there are pronouns which two of them are **anybody** and **anyone**. While storing **a n** and **y** characters are stored in the same nodes, but **b o d y** and **o n e** characters are stored separately.

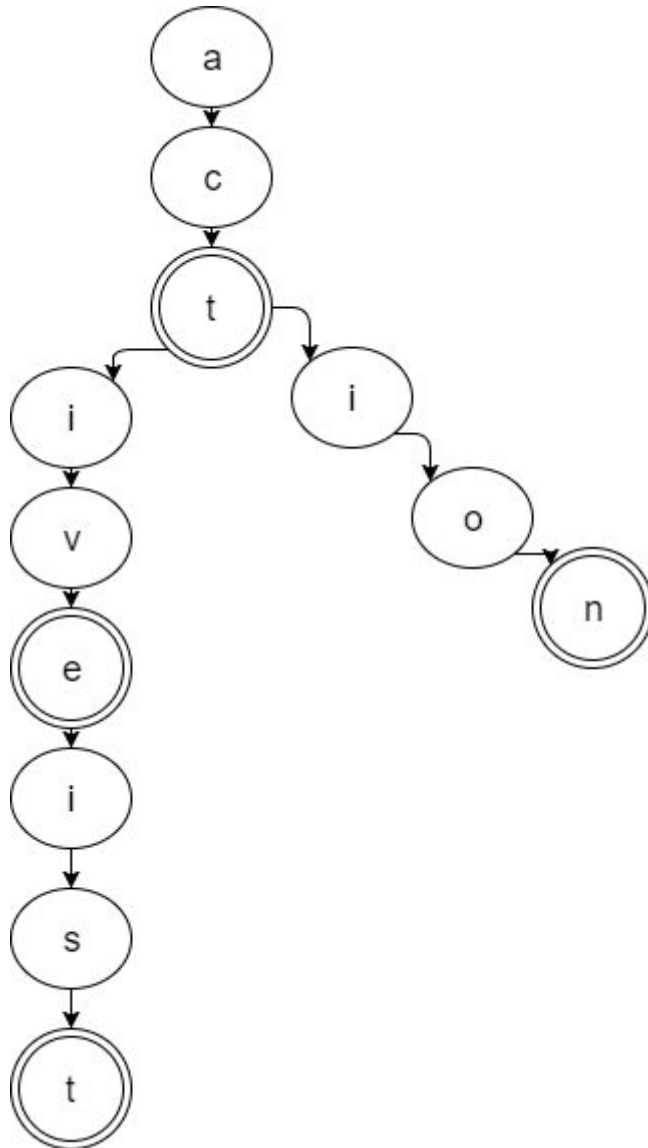


Example of the tree view for storing anyone and anybody.

After the stem insertion, suffixes are inserted one by one to the tree as each node represents a character in the suffix. In the language we defined, '(' character means insert

the following suffix on the previous root. Initially, root is the last character of the stem, so till the next '(' all suffixes which separated with '|' character are inserted to stem. When another '(' character has found, then the previously added suffix became root.

For this parsing, we defined a recursive algorithm. Base condition for this algorithm is facing ')' character, and root node changes as to previous root. Recursive step is facing '(' character, previously added node becomes root node.



Example of act (-ive:+ADJ(ist:+N))-ion:+N)

While adding suffixes to tree, end of the stem and suffixes are represented as final nodes which we need this information to the FSTs. Also, pos tags are stored on these nodes too.

1.3) FST Generation: After parsing morphotactics and stem from text files into tree, we can transform tree to FST readable text forms. For this project, OpenFST library is used for creating FSTs. Each final node in the previously created tree means accept state in fst.

Transitions represented as node characters in trees. By traversing tree, each node added to fst as transition. For example in tree a->b, in fst this means 0 state to 1 state/ a input - b output. So for this project, we created a script which makes this job.

0	1	<s>	<s>
1	0	<eps>	<eps>
0	2	</s>	</s>
2	0	<eps>	<eps>
0	3	<unk>	<unk>
3	0	<eps>	<eps>
0	4	a	a
4	5	n	n
5	6	y	y
6	7	b	b
7	8	o	o
8	9	d	d
9	10	y	y
10	0	+PRN	#
6	11	o	o
11	12	n	n
12	13	e	e
13	0	+PRN	#
0			

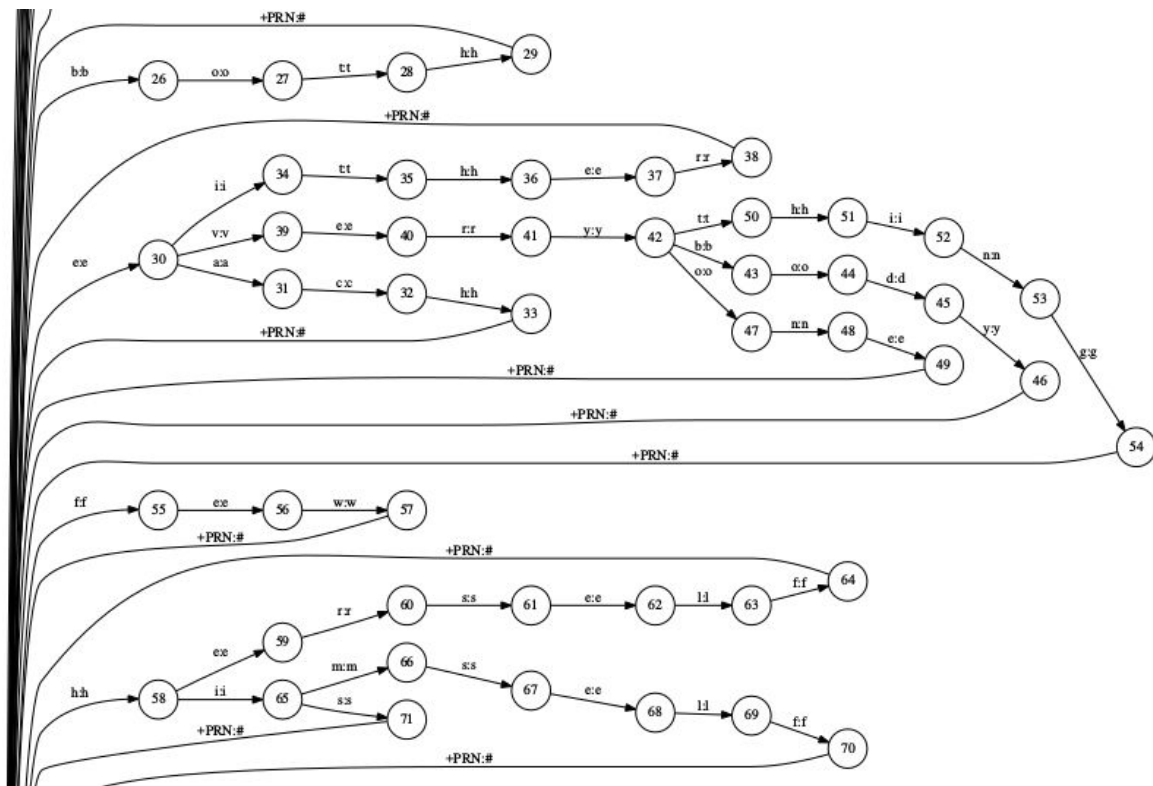
Just small part of pronouns FST shows a part for anybody - anyone. (Created by tree to fst transformation script)

0	4	a	a
4	5	c	c
5	6	c	c
6	7	e	e
7	8	p	p
8	9	t	t
9	10	+V	<eps>
10	0	<eps>	#
9	11	+3D	^s
11	0	<eps>	#
9	12	+ING	^ing
12	0	<eps>	#
9	13	-	<eps>
13	14	<eps>	^
14	15	e	e
15	16	e	e
16	17	+N	<eps>
17	0	<eps>	#

Some small part of verbs fst, showing accept stem with -ee and some inflectional suffixes.

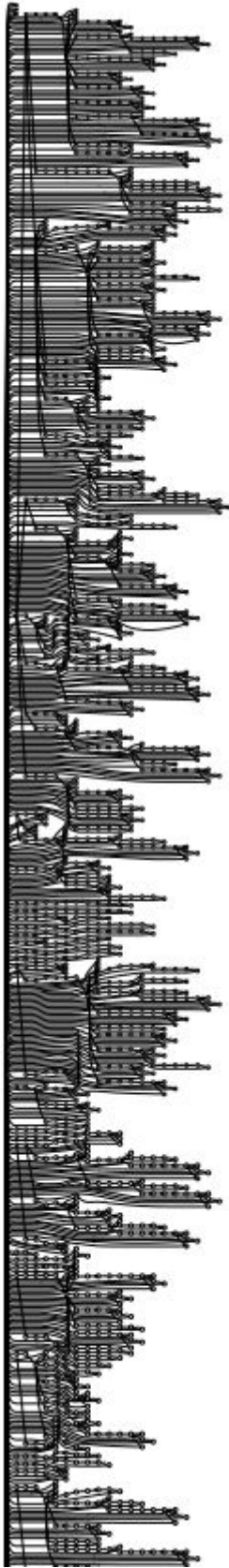
1.4) Compiling Morphotactics FSTs: Output of the previously mentioned script allows us to create fst for morphotactics. OpenFST has **fstcompile** shell command for compiling fst by given text representation. For transition symbols, --isymbols (input symbols) and --osymbols (output symbols) files are needed for fstcompile. For this issue, we also created a script to extract symbols from fst text file we generated.

After symbol tables generation, we compile FSTs for each category and obtain FST.



A small part of pronoun fst

While transducing, we compose all FSTs we have created with the input FST. If more than one match occurs from multiple FSTs, then we use them both (for printing output or sending as an input to the next level)



Verbs fst

2) Orthographic Rules (Morphophonemics): 3 different orthographic rules are covered in this application. And these rules are only programmed in one direction which is only to surface form. So, these rules can be examined in intermediate-surface conversion and lexicon-surface conversion applications.

These covered 3 rules are:

a) Dropping Silent E: If e occurs preceding a morpheme boundary ^ and this morpheme boundary is followed by a vowel, then that e is dropped.

For example, **make-ing -> making**

b) E Insertion: e added after s, z, x before ^s.

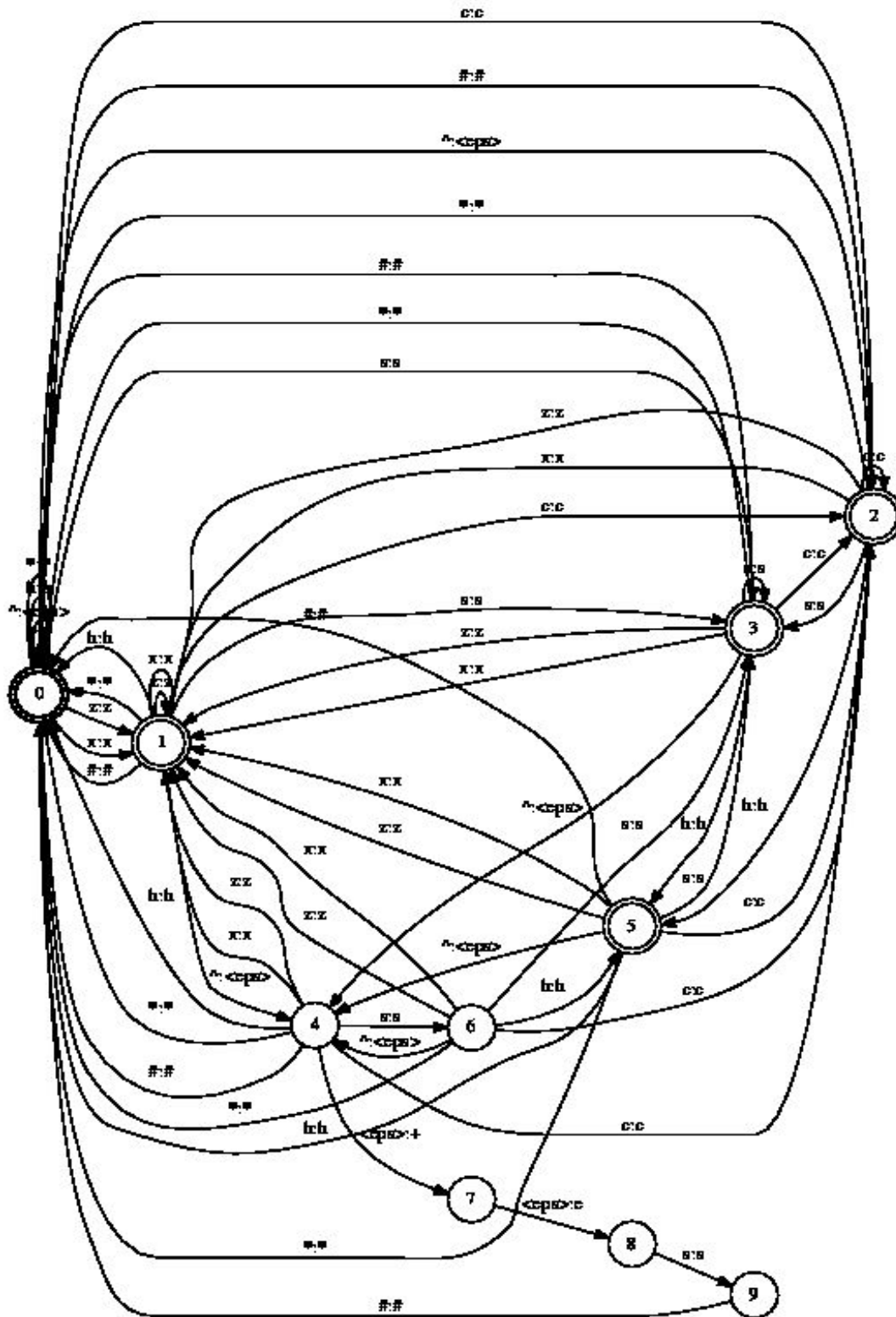
For example, **fox^s -> foxes**

c) Y Replacement: y changes to ie before ^s, i before ^ed.

For example, **try^s -> tries**

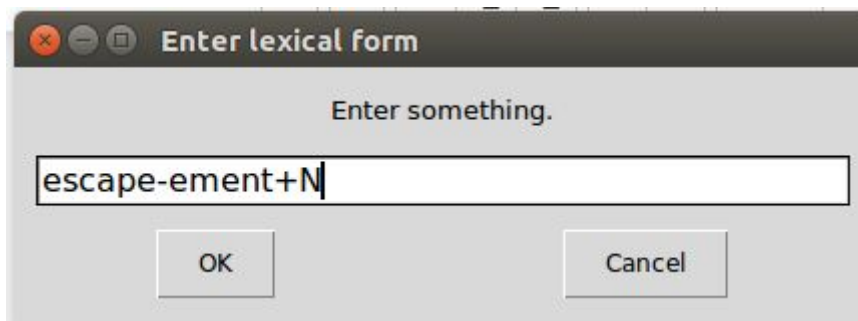
These rules are also programmed using FSTs. They are designed and executed separately on given input. If rule changes a morpheme (rule is executed) then a plus sign is added to that morpheme. For example, e-insertion FST converts **fish^s** to **fish+es**. If any rule adds + to the morpheme, then that conversion will be selected as the output of surface layer.

On below, e insertion rule FST can be seen. It is drawn differently than the programmed one, * represents other possible transitions on that arc. However, while implementing the real FST, all possible transactions are defined.



3) User Interface & Analyzer: Previously created FSTs are used in this part to analyze given inputs. This is made by a written program in Python language using official Python wrapper library for OpenFST. For lexicon to intermediate conversion, all FSTs for created all different categories read. Given input from user interface is composed with these FSTs. If any transducer produces an output, then its projected on user interface.

For example, for given input **escape-ement+N**,



A screenshot of a graphical user interface dialog box. The title bar at the top reads "Enter lexical form". Below the title bar, the text "Enter something." is displayed. A text input field contains the string "escape-ement+N". At the bottom of the dialog, there are two buttons: "OK" on the left and "Cancel" on the right.

results can be seen on below.



A screenshot of a graphical user interface window titled "Results". The window contains a large text area with the following text:
Intermediate form:
escape^ement#

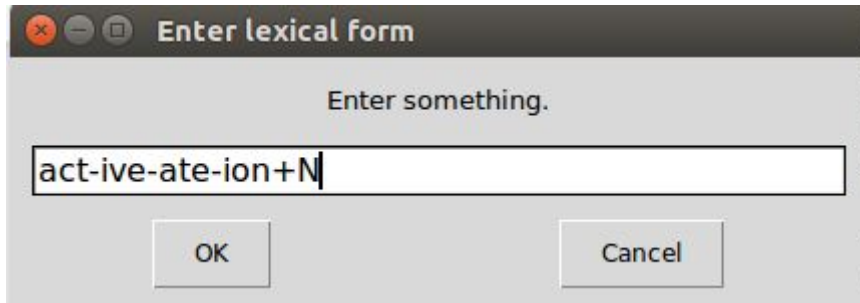
Surface form:
escapement#
The text is displayed in a monospaced font. There is a vertical scrollbar on the right side of the text area.

For reverse conversion (intermediate-lexicon) the given string is reversed, also previously created FSTs are reversed too. And for this time, these FSTs are composed with the FST which is created by user input.

Test Cases & Examples

- Lexical-Intermediate Analysis:

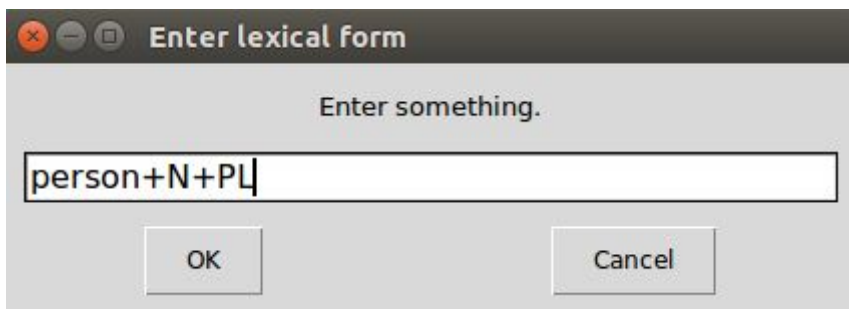
Multiple suffixes



A screenshot of a dialog box titled "Enter lexical form". It contains a text input field with the text "act-ive-ate-ion+N" and two buttons: "OK" and "Cancel".

```
act^ive^ate^ion#
```

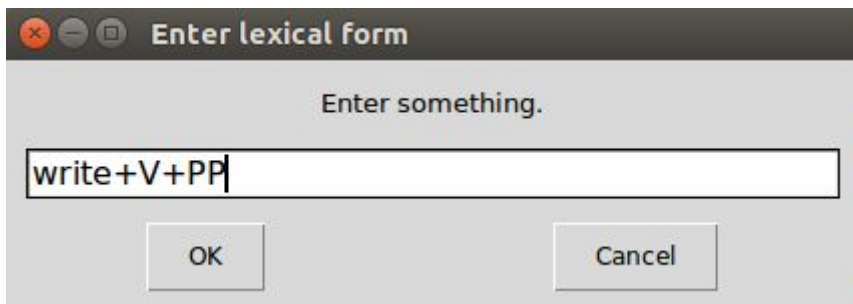
Plural nouns



A screenshot of a dialog box titled "Enter lexical form". It contains a text input field with the text "person+N+PL" and two buttons: "OK" and "Cancel".

```
people#
```

Irregular past form

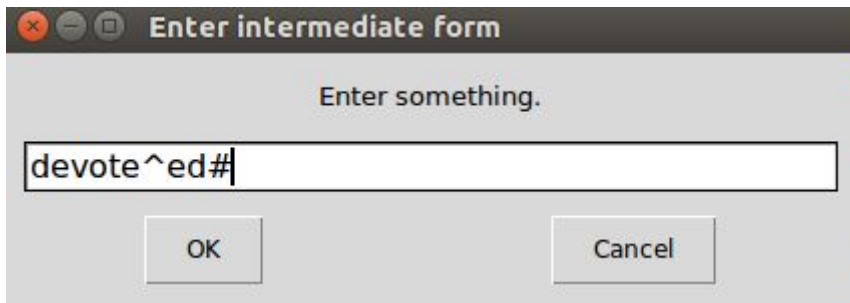


A screenshot of a dialog box titled "Enter lexical form". It contains a text input field with the text "write+V+PP" and two buttons: "OK" and "Cancel".

```
written#
```

- **Intermediate-Surface Analysis:**

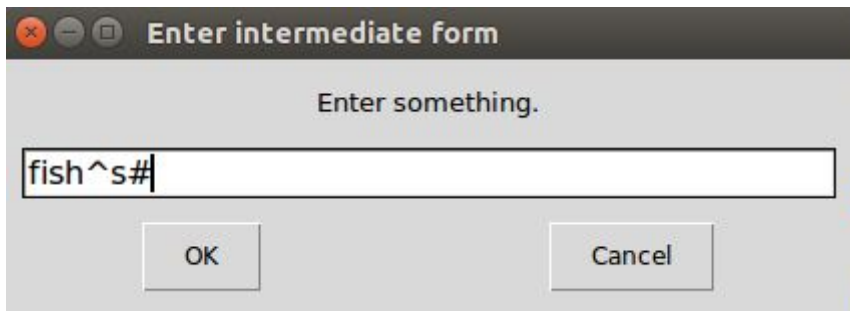
Silent e drop



A screenshot of a dialog box titled "Enter intermediate form". It contains a label "Enter something." and a text input field with the text "devote^ed#". Below the input field are two buttons: "OK" and "Cancel".

devoted#

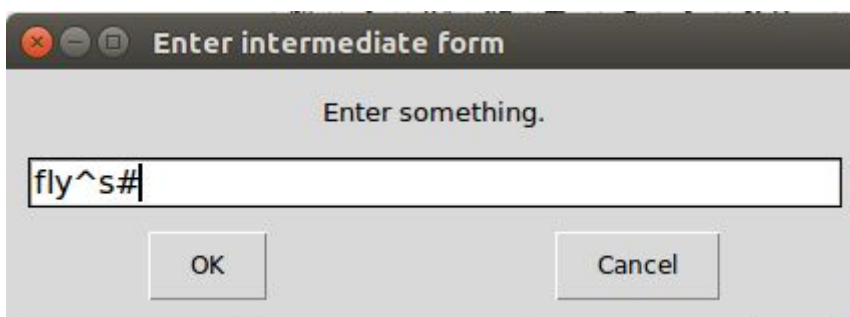
E insertion



A screenshot of a dialog box titled "Enter intermediate form". It contains a label "Enter something." and a text input field with the text "fish^s#". Below the input field are two buttons: "OK" and "Cancel".

fishes#

Y change



A screenshot of a dialog box titled "Enter intermediate form". It contains a label "Enter something." and a text input field with the text "fly^s#". Below the input field are two buttons: "OK" and "Cancel".

flies#

- Intermediate-Lexicon Analysis

A screenshot of a dialog box titled "Enter intermediate form". The dialog box has a title bar with standard window controls (close, minimize, maximize). Below the title bar, the text "Enter something." is displayed. A text input field contains the text "act#". At the bottom of the dialog box, there are two buttons: "OK" and "Cancel".

```
act+V
act+N+SG
|
```

- Lexicon-Surface Analysis

A screenshot of a dialog box titled "Enter lexical form". The dialog box has a title bar with standard window controls (close, minimize, maximize). Below the title bar, the text "Enter something." is displayed. A text input field contains the text "continue-ous+ADJ". At the bottom of the dialog box, there are two buttons: "OK" and "Cancel".

```
Intermediate form:
continue^ous#
```

```
Surface form:
continuous#
```

```
|
```

Improvements and Extensions

Lexicon which are covered in this project, may be improved. Size of stems and their suffixes could be increased. Also, which is also the weakest point of this project, orthographic rules may be increased, and surface to intermediate conversion could be developed. As for user interface, it can be designed better and more user friendly. Also, auto complete feature could be added to text box which user enters as input.

FSTs created for this project could be optimized and state number may be decreased. For example, Verbs FST have 1500 state counts. Because of this huge size of state numbers, graphical representations of FSTs are not represented in understandable format (see Verbs FST graphical model on the 1.4). Still, all graphical models are extracted in Lexicon/FSTs folder for each category (nouns,verbs,etc.) as both .ps and .pdf format.

Difficulties Encountered

Since we didn't have any previous hands-on-experience on OpenFst it took time for us to run it correctly. Having encountered Visual Studio C++ compiler issues on Windows, we realized that the OpenFst library works smoothly with GNU C++ compiler in Ubuntu.

Second challenge was determining stems that derived from the same root. For example, Which one is the stem: *enormity* or *enormous*? According to [1] both comes from “*Latin enormis, from e- (variant of ex-)‘out of’ + norma ‘pattern, standard’*”. How about *vivacity* and *vivacious*? Similarly, they are derived from “*Latin vivax, vivac- ‘lively, vigorous’*. [1]” In those cases we had to select both of the words as stems, with different POS tagging.

It took considerable time to develop morphophonemic rule FSTs, because we needed to meticulously transform intermediate forms to surface forms while not allowing sequences that doesn't follow the orthographic rules. It could be a lot easier If we were allowed to use some automated tools like such as Two-Level Rule Compiler [2]. Finally, when we parse surface string into intermediate form using orthographic rules, we encounter a lot of ambiguities. One example is e-insertion rule. While *fox-es* is derived by this rule, *axe-s* is not. Nevertheless, reversing the e-insertion FST would work on both words. Another example *foxes* could be both *fox+V+3SG* and *fox+N+PL* in those cases, we actually need some disambiguation algorithms to do the most appropriate parse [3]

Conclusion

We applied two-way morphological parsing approximately on two hundreds of English words and determined morphemes from which a given word is constructed. We parsed the words and generated rules for both morphotactics and orthographic rules. 4 different FSTs are created. Two of them converts a given string in lexical form to intermediate form and inverse. Other two transforms a given string in intermediate form to surface form and lexicon format to surface form. Finally, we combined lexicon-intermediate and intermediate-surface FSTs to build lexicon to surface converter in respect to two level morphology paradigm.

References

- [1] "enormity, n.1." *OED Online*. Oxford University Press, June 2017. Web. 14 November 2017.
- [2] Beesley, Kenneth R., and Lauri Karttunen. "Two-Level Rule Compiler." (2003).
- [3] Daniel Jurafsky and James F. Martin, *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition*, p.81, Upper Saddle River NJ: Prentice-Hall, 2000. ISBN: 0-13-095069-6
- [4] <http://www.openfst.org/>
- [5] http://www.umiacs.umd.edu/~hollingk/tutorials/JHU_tutorial.html
- [6] http://web.eecs.umich.edu/~radev/NLP-fall2015/resources/fsm_archive/fsm.5.html
- [7] <https://stackoverflow.com/questions/9390536/how-do-you-even-give-an-openfst-made-fst-input-where-does-the-output-go>
- [8] <http://www.isle.illinois.edu/sst/courses/minicourses/2009/lecture6.pdf>
- [9] <https://www.cl.cam.ac.uk/teaching/1213/L102/practicals//handout-2.pdf>
- [10] http://idiom.ucsd.edu/~rlevy/teaching/2015winter/lign165/lectures/lecture10/lecture10_working_with_OpenFst.pdf

Appendix

Lexicon2Surface.py

```
import pywrapfst as ofst
import easygui
LEX_PATH='Lexicon/FSTs/'
categories
=['Verbs','Nouns','Pronouns','Adjectives','Adverbs','Propositions'
,'Auxillary Verbs']
MORPH_PATH='Morphophonemics/'
rules = ['y-ie','silente','einsert']
LEX="LEX"
MORPH="MORPH"

# In[6]:

def set_input_fsm_int(input_string,ifst,isys):
    i=0
    while input_string[i]!='+':
        #print(str(i)+ " " + str(i+1)+ " " + input_string[i])
        print >> ifst, str(i)+ " " + str(i+1)+ " " +
input_string[i]
        i+=1
    props=input_string[i+1:].split('+')
    for p in props:
        #print(str(i) + " " + str(i+1)+" " + p)
        print >> ifst, str(i) + " " + str(i+1)+" " + p
        i+=1
    print >> ifst, str(i)

    #print(i)
    try:
        return ifst.compile()
```

```

    except:
        return None

def set_input_fsm_surf(input_string, ifst):
    for i in range(len(input_string)):
        #print ( ifst, str(i)+ " " + str(i+1)+ " " +
input_string[i])
        print >> ifst, str(i)+ " " + str(i+1)+ " " +
input_string[i]
        print >> ifst, str(len(input_string))
        #print(i)
    try:
        return ifst.compile()
    except:
        return None

def transduce((fst, ifst, isys), input_string, strategy):
    if strategy==LEX:
        a=set_input_fsm_int(input_string, ifst, isys)
    else:
        a=set_input_fsm_surf(input_string, ifst)
    if a==None:
        return ""
    else:
        b=ofst.compose(a, fst)
        b.set_input_symbols(isys)
        b.set_output_symbols(isys)
        b.project(project_output=True)
        lines=b.text(acceptor=True).split("\n")
        output=""
        for l in lines:
            sp=l.split("\t")
            if len(sp)==3 and sp[2]!='<eps>':
                output+=sp[2]
        return output

def get_lex_fst(name):
    syms=ofst.SymbolTable.read_text(LEX_PATH+name+'/symbols.txt')

```



```

        fst=ofst.Fst.read(LEX_PATH+name+'/'+name.lower()+'.fst')
    return
fst,ofst.Compiler(isymbols=syms,osymbols=syms,acceptor=True),syms
def get_morph_fst(name):
    syms=ofst.SymbolTable.read_text(MORPH_PATH+'symbols.txt')
    fst=ofst.Fst.read(MORPH_PATH+name.lower()+'.fst')
    return
fst,ofst.Compiler(isymbols=syms,osymbols=syms,acceptor=True),syms

# In[7]:

def run_lex2int(input_string):
    matches=""
    found=False
    for i in categories:
        match=transduce(get_lex_fst(i),input_string,LEX)
        if match!="":
            matches+=match+"\n"
            found=True
    if found==False:
        print("No match!")
    return matches
def run_int2surf(input_string):
    matches=""
    found=False
    for i in rules:
        match=transduce(get_morph_fst(i),input_string,MORPH)
        if '+' in match:
            return match.replace("^","").replace("+","")
    if found==False:
        return input_string

```

```

# In[9]:

a=easygui.enterbox(title="Enter lexical form")
if '+' not in a:
    easygui.textbox(text="No match, because you did not enter any
POS tag")
elif a!=None and a!="":
    results_int=run_lex2int(a)
    if len(results_int)>0:
        output="Intermediate form:\n"+results_int
        result_surface=""
        for form in results_int.split("\n"):
            result_surface+=run_int2surf(form)+"\n"
        easygui.textbox(title="Results",text=output + "\nSurface
form:\n" + result_surface)
    else:
        easygui.textbox(text="No match!")

```

Lexical2Intermediate.py

```

import pywrapfst as ofst
import easygui
FST_PATH='Lexicon/FSTs/'
categories
=['Verbs','Nouns','Pronouns','Adjectives','Adverbs','Propositions'
,'Auxillary Verbs']

def set_input_fsm(input_string,ifst,isys):
    i=0
    while input_string[i]!='+':
        #print(str(i)+ " " + str(i+1)+ " " + input_string[i])
        print >> ifst, str(i)+ " " + str(i+1)+ " " +
input_string[i]
        i+=1
    props=input_string[i+1:].split('+')

```

```

for p in props:
    #print(str(i) + " " + str(i+1)+" " + p)
    print >> ifst, str(i) + " " + str(i+1)+" " + p
    i+=1
print >> ifst, str(i)

#print(i)
try:
    return ifst.compile()
except:
    return None

def transduce((fst,ifst,isys),input_string):
    a=set_input_fsm(input_string,ifst,isys)
    if a==None:
        return ""
    else:
        b=ofst.compose(a,fst)
        b.set_input_symbols(isys)
        b.set_output_symbols(isys)
        b.project(project_output=True)
        lines=b.text(acceptor=True).split("\n")
        output=""
        for l in lines:
            sp=l.split("\t")
            if len(sp)==3 and sp[2]!='<eps>':
                output+=sp[2]
        return output

def get_fst(name):
    syms=ofst.SymbolTable.read_text(FST_PATH+name+'/symbols.txt')
    fst=ofst.Fst.read(FST_PATH+name+'/'+name.lower()+'.fst')
    return
fst,ofst.Compiler(isymbols=syms,osymbols=syms,acceptor=True),syms

```

```
# In[46]:
```

```
def run(input_string):
    matches=""
    found=False
    for i in categories:
        match=transduce(get_fst(i),input_string)
        if match!="":
            matches+=match+"\n"
            found=True
    if found==False:
        print("No match!")
    return matches
```

```
# In[56]:
```

```
a=easygui.enterbox(title="Enter lexical form")
if '+' not in a:
    easygui.textbox(text="No match, because you did not enter any
POS tag")
elif a!=None:
    results=run(a)
    if len(results)>0:
        easygui.textbox(text=results)
    else:
        easygui.textbox(text="No match!")
```

Intermediate2Surface.py

```
# coding: utf-8
```

```
# In[73]:
```

```

import pywrapfst as ofst
import sys
FST_PATH='Morphophonemics/'
rules = ['y-ie','silente','einsert']
import easygui

def set_input_fsm(input_string,ifst):
    for i in range(len(input_string)):
        #print ( ifst, str(i)+ " " + str(i+1)+ " " +
input_string[i])
        print >> ifst, str(i)+ " " + str(i+1)+ " " +
input_string[i]
        print >> ifst, str(len(input_string))
        #print(i)
    try:
        return ifst.compile()
    except:
        return None

def transduce((fst,ifst,isys),input_string):
    a=set_input_fsm(input_string,ifst)
    if a==None:
        return ""
    else:
        b=ofst.compose(a,fst)
        b.set_input_symbols(isys)
        b.set_output_symbols(isys)
        b.project(project_output=True)
        lines=b.text(acceptor=True).split("\n")
        output=""
        for l in lines:
            sp=l.split("\t")

```

```

        if len(sp)==3 and sp[2]!='<eps>':
            output+=sp[2]
        return output
def get_fst(name):
    syms=ofst.SymbolTable.read_text(FST_PATH+'symbols.txt')
    fst=ofst.Fst.read(FST_PATH+name.lower()+'.fst')
    return
fst,ofst.Compiler(isymbols=syms,osymbols=syms,acceptor=True),syms

# In[74]:

def run(input_string):
    matches=""
    found=False
    for i in rules:
        match=transduce(get_fst(i),input_string)
        if '+' in match:
            return match.replace("^","").replace("+","")
    if found==False:
        return input_string

# In[82]:

a=easygui.enterbox(title="Enter intermediate form")
if a!=None:
    results=run(a)
    if len(results)>0:
        easygui.textbox(text=results)
    else:
        easygui.textbox(text="No match!")

```

Intermediate2Lexical.py

```
# coding: utf-8

# In[58]:

import pywrapfst as ofst
import easygui
FST_PATH='Lexicon/FSTs/'
categories
=['Verbs','Nouns','Pronouns','Adjectives','Adverbs','Propositions'
,'Auxillary Verbs']

# In[59]:

def set_input_fsm(input_string,ifst):
    input_string=input_string[::-1]
    for i in range(len(input_string)):
        #print(str(i)+ " " + str(i+1)+ " " + input_string[i])
        print >> ifst, str(i)+ " " + str(i+1)+ " " +
input_string[i]
        #print (str(i+1))
        print >> ifst,str(i+1)
        try:
            return ifst.compile()
        except:
            return None
def transduce((fst,ifst,isys),input_string):
    a=set_input_fsm(input_string,ifst)
    if a==None:
        return "";
    else:
```

```

b=ofst.compose(fst,a)
b.set_input_symbols(isys)
lines=b.text(acceptor=True).split("\n")
output=""
for l in lines:
    sp=l.split("\t")
    if len(sp)==4 and sp[2]!='<eps>':
        if sp[2][0]=='+' :
            output+=sp[2][::-1]
        else:
            output+=sp[2]
    return output[::-1]
def get_fst(name):
    syms=ofst.SymbolTable.read_text(FST_PATH+name+'/symbols.txt')
    fst=ofst.Fst.read(FST_PATH+name+'/'+name.lower()+'.fst')
    return
ofst.reverse(fst),ofst.Compiler(isymbols=syms,osymbols=syms,accept
or=True),syms

# In[64]:

def run(input_string):
    matches=""
    found=False
    for i in categories:
        match=transduce(get_fst(i),input_string)
        if match!="":
            matches+=match+"\n"
            found=True
    if found==False:
        print("No match!")
    return matches

```



```
# In[68]:

a=easygui.enterbox(title="Enter intermediate form")
if a!=None and a!="":
    results=run(a)
    if len(results)>0:
        easygui.textbox(title="Results",text=results)
    else:
        easygui.textbox(text="No match!")
```